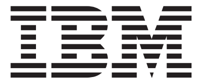
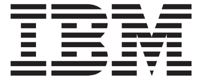


IBM i

ILE C/C++ ランタイム・ライブラリー関数

7.1

SC88-4701-01
(英文原典：SC41-5607-04)



IBM i

ILE C/C++ ランタイム・ライブラリー関数

7.1

SC88-4701-01
(英文原典：SC41-5607-04)

お願い

本書および本書で紹介する製品をご使用になる前に、 603 ページの『付録 B. 特記事項』に記載されている情報をお読みください。

本書は、IBM i 7.1 (製品番号 5770-SS1) に適用されます。また、改訂版で断りがない限り、それ以降のすべてのリリースおよびモディフィケーションに適用されます。このバージョンは、すべての RISC モデルで稼働するとは限りません。また CISC モデルでは稼働しません。

お客様の環境によっては、資料中の円記号がバックスラッシュと表示されたり、バックスラッシュが円記号と表示されたりする場合があります。

原典： SC41-5607-04
IBM i
ILE C/C++ Runtime Library Functions
7.1

発行： 日本アイ・ビー・エム株式会社

担当： トランスレーション・サービス・センター

第1刷 2010.4

© Copyright International Business Machines Corporation 1999, 2010.

目次

表 ix

本書について xi
本書の対象読者 xi
例についての注記 xi
前提条件および関連情報 xii

変更内容の要約 xiii

第 1 部 ランタイム・ライブラリー関数 1

第 1 章 インクルード・ファイル 3
<assert.h> 3
<ctype.h>. 3
<decimal.h> 4
<errno.h>. 4
<except.h> 4
<float.h> 7
<inttypes.h> 7
<langinfo.h> 8
<limits.h> 8
<locale.h> 8
<math.h> 9
<mallocinfo.h> 9
<monetary.h> 9
<nl_types.h> 9
<pointer.h> 10
<recio.h> 10
<regex.h> 13
<setjmp.h> 14
<signal.h> 15
<stdarg.h> 15
<stddef.h> 15
<stdint.h> 15
<stdio.h> 17
<stdlib.h> 18
<string.h> 19
<strings.h> 19
<time.h> 19
<wchar.h> 20
<wctype.h>. 20
<xxcvt.h> 20
<xxdtaa.h> 21
<xxenv.h> 21
<xxfdbk.h>. 21
マシン・インターフェース (MI) インクルード・ファイル 22

第 2 章 ライブラリー関数 23
C/C++ ライブラリー 23
エラー処理 23
検索およびソート 24
数学関数 24
時間処理 25
型変換 26
変換 27
レコード入出力 28
ストリーム入出力 29
引数リストの処理 32
疑似乱数 32
動的メモリー管理 33
メモリー・オブジェクト 33
環境の対話 34
ストリング操作 34
文字テスト 35
マルチバイト文字のテスト 36
文字ケースのマッピング 36
マルチバイト文字の操作 36
データ域 38
メッセージ・カタログ 38
正規表現 38
abort() ― プログラムの停止 38
abs() ― 整数の絶対値の計算 39
acos() ― 逆余弦の計算 40
asctime() ― 時間から文字ストリングへの変換 . . . 41
asctime_r() ― 時間から文字ストリングへの変換 (再始動可能) 43
asin() ― 逆正弦の計算 45
assert() ― 条件の検証 46
atan() - atan2() ― 逆正接の計算 47
atexit() ― プログラム終了の記録関数. 48
atof() ― 文字ストリングから浮動小数点への変換. . 49
atoi() ― 文字ストリングから整数への変換 50
atol() ― atoll() ― 文字ストリングの long 型整数または long long 型整数への変換 52
ベッセル関数 53
bsearch() ― 配列の検索 54
btowc() ― 1 バイト文字のワイド文字への変換 . . 56
_C_Get_Ssn_Handle() ― C セッションへのハンドル 57
calloc() ― ストレージの予約と初期化 58
catclose() ―メッセージ・カタログのクローズ . . . 60
catgets() ― メッセージ・カタログからのメッセージの検索 61
catopen() ―メッセージ・カタログのオープン . . . 63
ceil() ― 整数の検索 >= 引数 64
clearerr() ― エラー標識のリセット 65
clock() ― プロセッサー時間の判別 67
cos() ― 余弦の計算 68
cosh() ― 双曲線余弦の計算 69

© Copyright IBM Corp. 1999, 2010

| _C_Quickpool_Debug() — 高速プール・メモリー・マネージャー特性の変更 69
| _C_Quickpool_Init() — 高速プール・メモリー・マネージャーの初期化 71
| _C_Quickpool_Report() — 高速プール・メモリー・マネージャー・レポートの生成 73
ctime() — 時間から文字ストリングへの変換 . . . 75
ctime64() — 時間から文字ストリングへの変換. . . 77
ctime_r() — 時間から文字ストリングへの変換 (再始動可能) 78
ctime64_r() — 時間から文字ストリングへの変換 (再始動可能) 80
_C_TS_malloc_debug() — 使用されるテラスペース・メモリー量の判別 (オプションのダンプおよび検査を使用) 82
_C_TS_malloc_info() — 使用されるテラスペース・メモリー量の判別 84
difftime() — 時差の計算 86
difftime64() — 時差の計算 88
div() — 商および剰余の計算 90
erf() - erfc() — 誤差関数の計算 91
exit() — プログラムの終了 92
exp() — 指数関数の計算 93
fabs() — 浮動小数点絶対値の計算. 94
fclose() — ストリームのクローズ 95
fdopen() — ストリームとファイル記述子との関連付け 96
feof() — ファイル終了標識のテスト 99
ferror() — 読み取り/書き込みエラーのテスト. . . 100
fflush() — ファイルへのバッファーの書き込み . . 101
fgetc() — 文字の読み取り 102
fgetpos() — ファイル位置の取得 104
fgets() — ストリングの読み取り 105
fgetwc() — ストリームのワイド文字の読み取り 107
fgetws() — ストリームのワイド文字のストリングの読み取り 109
fileno() — ファイル・ハンドルの判別 111
floor() — 整数の検索 <=引数 112
fmod() — 浮動小数点の剰余の計算 113
fopen() — ファイルのオープン 114
fprintf() — フォーマット済みデータのストリームへの書き込み 122
fputc() — 文字の書き込み 124
_fputchar - 文字の書き込み 126
fputs() — ストリングの書き込み 127
fputwc() — 文字の書き込み 128
fputws() — ワイド文字ストリングの書き込み. . . 130
fread() — 項目の読み取り 132
free() — ストレージ・ブロックの解放 134
freopen() — オープン・ファイルのリダイレクト 136
frexp() — 浮動小数点値の分離 137
fscanf() — フォーマット済みデータの読み取り . . 138
fseek() — fseeko() — ファイル位置の位置変更 . . 140
fsetpos() — ファイル位置の設定 142
ftell() — ftello() — 現在位置の取得 144
fwide() — ストリーム指向の決定. 146
fwprintf() — ワイド文字としてのデータのフォーマット設定とストリームへの書き込み 149
fwrite() — 項目の書き込み 152
fwscanf() — ワイド文字を使用したストリームからのデータの読み取り 153
gamma() — ガンマ関数 156
_gcvt - 浮動小数点からストリングへの変換 . . . 157
getc() - getchar() — 文字の読み取り 158
getenv() — 環境変数の検索. 160
_GetExcData() — 例外データの取得 161
gets() — 行の読み取り 162
getwc() — ストリームからのワイド文字の読み取り 163
getwchar() — STDIN からのワイド文字の取得 . . 165
gmtime() — 時間の変換 167
gmtime64() — 時間の変換 169
gmtime_r() — 時間の変換 (再始動可能). 171
gmtime64_r() — 時間の変換 (再始動可能) 173
hypot() — 斜辺の計算 175
isalnum() - isxdigit() — 整数値のテスト 176
isascii() — 表示可能文字の ASCII 値としてのテスト 177
isblank() —ブランクまたはタブ文字のテスト . . . 179
iswalnum() から iswxdigit() — ワイド整数値のテスト 180
iswctype() — 文字プロパティーのテスト 182
_itoa - 整数からストリングへの変換. 183
labs() — llabs() — long 型および long long 型整数の絶対値の計算. 184
ldexp() — 2 のべき乗の乗算 185
ldiv() — lldiv() — long 型整数および long long 型整数の除算の実行 186
localeconv() — 環境からの情報の取得 187
localtime() — 時間の変換 192
localtime64() — 時間の変換. 194
localtime_r() — 時間の変換 (再始動可能) 195
localtime64_r() — 時間の変換 (再始動可能) . . . 197
log() — 自然対数の計算. 198
log10() — 基数 10 の対数の計算. 199
_ltoa - long 型整数からストリングへの変換 . . . 200
longjmp() — スタック環境の復元. 201
malloc() — ストレージ・ブロックの予約 203
mblen() — マルチバイト文字の長さの計算 . . . 205
mbrlen() — マルチバイト文字の長さの計算 (再始動可能) 207
mbrtowc() — マルチバイト文字からワイド文字への変換 (再始動可能) 210
mbsinit() — 状態オブジェクトが初期状態であるかどうかのテスト. 213
mbsrtowcs() — マルチバイト・ストリングからワイド文字ストリングへの変換 (再始動可能) 214
mbstowcs() — マルチバイト・ストリングからワイド文字ストリングへの変換 216
mbtowc() — マルチバイト文字からワイド文字への変換 220
memchr() — バッファーの検索 221
memcmp() — バッファーの比較 222

memcpy() — バイトのコピー 223
memicmp() - バイトの比較 224
memmove() — バイトのコピー 226
memset() — 値へのバイトの設定 227
mktime() — 地方時の変換 228
mktime64() — 地方時の変換 230
modf() — 浮動小数点値の分離 232
nextafter() — nextafterl()— nexttoward() — nexttowardl() — 表現可能な次の浮動小数点値の計算 232
nl_langinfo() — ロケール情報の検索. 234
perror() — エラー・メッセージの出力 236
pow() — 累乗の計算 237
printf() — 定様式の文字の出力 238
putc() - putchar() — 文字の書き込み. 249
putenv() — 環境変数の変更/追加 250
puts() — ストリングの書き込み 251
putwc() — ワイド文字の書き込み 252
putwchar() — ワイド文字の stdout への書き込み 254
qsort() — 配列のソート 256
QXXCHGDA() — データ域の変更 258
QXXDTOP() — double からパック 10 進数への変換 259
QXXDTOZ() — double からゾーン 10 進数への変換 260
QXXITOP() — 整数からパック 10 進数への変換 261
QXXITOZ() — 整数からゾーン 10 進数への変換 262
QXXPTOD() — パック 10 進数から double への変換 262
QXXPTOI() — パック 10 進数から整数への変換 263
QXXRTVDA() — データ域の検索 264
QXXZTOD() — ゾーン 10 進数から double への変換 265
QXXZTOI() — ゾーン 10 進数から整数への変換 266
raise() — シグナルの送信 267
rand()、rand_r() — 乱数の生成. 268
_Racquire() — プログラム装置の獲得 269
_Rclose() — ファイルのクローズ. 270
_Rcommit() — 現行レコードのコミット 271
_Rdelete() — レコードの削除 273
_Rdevatr() — 装置属性の取得 275
realloc() — 予約ストレージ・ブロック・サイズの変更 276
regcomp() — 正規表現のコンパイル 279
regerror() — 正規表現のエラー・メッセージの戻し 281
regexec() — コンパイル済み正規表現の実行 . . . 283
regfree() — 正規表現のメモリーの解放 285
remove() — ファイルの削除 286
rename() — ファイルの名前変更 287
rewind() — 現在のファイル位置の調整 288
_Rfeod() — データの終わりの強制 290
_Rfeov() — ファイルの終わりの強制 291
_Rformat() — レコード・フォーマット名の設定 292
_Rindara() — 分離標識域の設定 294
_Riofbk() — 入出力フィードバック情報の取得 . . 296
_Rlocate() — レコードの位置指定 298
_Ropen() — レコード・ファイルをオープンして入出力操作を行う. 301
_Ropnfbk() — オープン・フィードバック情報の取得 305
_Rpgmdev() — デフォルトのプログラム装置の設定 306
_Rreadd() — 相対レコード番号によるレコードの読み取り. 308
_Rreadf() — 最初のレコードの読み取り 310
_Rreadindv() — 送信勧誘された装置からの読み取り 312
_Rreadk() — キーによるレコードの読み取り . . . 314
_Rreadl() — 最終レコードの読み取り 318
_Rreadn() — 次のレコードの読み取り 319
_Rreadnc() — サブファイル内の次の変更済みレコードの読み取り. 322
_Rreadp() — 前のレコードの読み取り 324
_Rreads() — 同じレコードの読み取り 326
_Rrelease() — プログラム装置の解除 328
_Rrlslck() — レコード・ロックの解除 329
_Rrollbck() — コミットメント制御の変更のロールバック. 331
_Rupdate() — レコードの更新 332
_Rupfb() — 最終入出力操作についての情報の説明 334
_Rwrite() — 次のレコードの書き込み 336
_Rwrited() — レコード・ディレクトリーの書き込み 338
_Rwriterd() — レコードの書き込みと読み取り . . 341
_Rwrread() — レコードの書き込みと読み取り (分離バッファー) 342
scanf() — データの読み取り 344
setbuf() — バッファリングの制御 351
setjmp() — 環境の保存 352
setlocale() — ロケールの設定 354
setvbuf() — バッファリングの制御 360
signal() — 割り込みシグナルの処理 362
sin() — 正弦の計算 365
sinh() — 双曲線正弦の計算. 366
snprintf() — フォーマット設定データのバッファーへの出力 367
sprintf() — フォーマット設定データのバッファーへの出力. 368
sqrt() — 平方根の計算 369
srand() — rand() 関数の seed の設定 370
sscanf() — データの読み取り 371
strcasecmp() — 大/小文字を区別しないストリングの比較. 373
strcat() — ストリングの連結 374
strchr() — 文字の検索 375
strcmp() — ストリングの比較 376
strcmpi() - 大/小文字を区別しないストリングの比較 378
strcoll() — ストリングの比較 379
strcpy() — ストリングのコピー 380
strcspn() — 最初に一致した文字のオフセットの検索 381
strdup - ストリングの複製 383
strerror() — ランタイム・エラー・メッセージを指すポインターの設定 384

strfmon() — 通貨値からストリングへの変換 . . . 384
strftime() — 日付/時刻からストリングへの変換 . . 387
stricmp() - 大/小文字を区別しないストリングの比較 390
strlen() — ストリング長の判別 392
strncasecmp() — 大/小文字を区別しないストリングの比較. 393
strncat() — ストリングの連結 394
strncmp() — ストリングの比較 395
strncpy() — ストリングのコピー 397
strnicmp - 大/小文字の区別をしないサブストリングの比較. 398
strnset - strset - ストリング内の文字の設定 . . . 400
strpbrk() — ストリング内の文字の検索. 401
strptime()— ストリングから日付/時刻への変換 . . 402
strrchr() — ストリング内で文字が最後に現れる位置の検出. 406
strspn() —最初の不一致文字のオフセットの検索 407
strstr() — サブストリングの位置検出 408
strtod() — strtof() — strtold — 文字ストリングから double、浮動、および long double への変換 . . . 409
strtod32() — strtod64() — strtod128() — 文字ストリングから 10 進浮動小数点への変換 412
strtok() — ストリングのトークン化 415
strtok_r() — ストリングのトークン化 (再開可能) 417
strtol() — strtoll() — 文字ストリングから long 型および long long 型整数への変換. 418
strtoul() — strtoull() — 文字ストリングから符号なし long 型整数および符号なし long long 型整数への変換. 420
strxfrm() — ストリングの変換. 422
swprintf() — ワイド文字のフォーマット設定とバッファーへの書き込み 423
swscanf() — ワイド文字データの読み取り 425
system() — コマンドの実行 426
tan() — 正接の計算 427
tanh() — 双曲線正接の計算. 428
time() — 現在時刻の判別 429
time64() — 現在時刻の判別 430
tmpfile() — 一時ファイルの作成 432
tmpnam() — 一時ファイル名の作成 433
toascii() — 文字から ASCII で表現可能な文字への変換 433
tolower() - toupper() — 英大/小文字の変換. . . . 434
towctrans() — ワイド文字の変換 435
towlower() -towupper() — ワイド文字の英大/小文字の変換. 436
_ultoa - 符号なし long 型整数からストリングへの変換 438
ungetc() — 入力ストリームへの文字のプッシュ . . 439
ungetwc() — 入力ストリームへのワイド文字のプッシュ 440
va_arg() - va_end() - va_start() — 関数引数のアクセス 442
vfprintf() — ストリームへの引数データの出力 . . 444
vfscanf() — フォーマット済みデータの読み取り 446
vfwprintf() — 引数データのワイド文字としてのフォーマット設定とストリームへの書き込み. . . . 447
vfwscanf() — フォーマット済みワイド文字データの読み取り 449
vprintf() — 引数データの出力 452
vscanf() — フォーマット済みデータの読み取り . . 453
vsnprintf() — 引数データのバッファーへの出力 . . 455
vsprintf() — 引数データのバッファーへの出力 . . 456
vsscanf() — フォーマット済みデータの読み取り 458
vswprintf() — ワイド文字のフォーマット設定とバッファーへの書き込み 459
vswscanf() — フォーマット済みワイド文字データの読み取り 461
vwprintf() — 引数データのワイド文字としてのフォーマット設定と出力 463
vwscanf() — フォーマット済みワイド文字データの読み取り 465
wcrtomb() — ワイド文字からマルチバイト文字への変換 (再開可能) 467
wcscat() — ワイド文字ストリングの連結 471
wcschr() — ワイド文字の検索 472
wcscmp() — ワイド文字ストリングの比較. . . . 474
wcscoll() —言語照合ストリングの比較 475
wcscpy() — ワイド文字ストリングのコピー . . . 476
wcscspn() — 最初に一致したワイド文字のオフセットの検索 477
wcsftime() — フォーマット済み日時への変換. . . 479
__wcsicmp() — 大/小文字の区別をしないワイド文字ストリングの比較 480
wcslen() — ワイド文字ストリング長の計算 . . . 482
wcslocaleconv() — ワイド・ロケール情報の検索 483
wcsncat() — ワイド文字ストリングの連結. . . . 484
wcsncmp() — ワイド文字ストリングの比較 . . . 485
wcsncpy() — ワイド文字ストリングのコピー. . . 487
__wcsnicmp() — 大/小文字の区別をしないワイド文字ストリングの比較 488
wcspbrk() — ストリング内のワイド文字の位置検出 490
wcsptime()— ワイド文字ストリングから日付/時刻への変換. 491
wcsrchr() — ストリング内でワイド文字が最後に現れる位置の検出. 493
wcsrtombs() — ワイド文字ストリングからマルチバイト・ストリングへの変換 (再開可能) 494
wcsspn() — 最初の不一致ワイド文字のオフセットの検索. 496
wcsstr() — ワイド文字サブストリングの位置検出 497
wcstod() — ワイド文字ストリングから double 型への変換. 498
wcstod32() — wcstod64() — wcstod128()—ワイド文字ストリングから 10 進浮動小数点への変換 . . . 499
wcstok() — ワイド文字ストリングのトークン化 502
wcstol() — wcstoll() — ワイド文字ストリングから long 型および long long 型整数への変換 503
wcstombs() — ワイド文字ストリングからマルチバイト・ストリングへの変換 505

wcstoul() — wcstoull() — ワイド文字ストリングから符号なし long 型整数および符号なし long long 型整数への変換. 508
wcswcs() — ワイド文字サブストリングの位置検出 510
wcswidth() — ワイド文字ストリングの表示幅の判別 511
wcsxfrm() — ワイド文字ストリングの変換 . . . 512
wctob() — ワイド文字からバイトへの変換 . . . 513
wctomb() — ワイド文字からマルチバイト文字への変換 515
wctrans() —文字マッピングのハンドルの取得. . . 516
wctype() — 文字特性種別のハンドルの取得 . . . 517
wcwidth() — ワイド文字の表示幅の判別 520
wfopen() —オープン・ファイル 521
wmemchr() —ワイド文字バッファーでのワイド文字の位置検出 521
wmemcmp() —ワイド文字バッファーの比較 . . . 522
wmemcpy() —ワイド文字バッファーのコピー. . . 524
wmemmove() — ワイド文字バッファーのコピー 525
wmemset() — 値に対するワイド文字バッファーの設定 526
wprintf() — データのワイド文字としてのフォーマット設定と出力. 527
wscanf() — ワイド文字書式ストリングを使用したデータの読み取り 528

第 3 章 ランタイムに関する考慮事項 531

errno マクロ. 531
統合ファイル・システム対応 C ストリーム入出力の errno 値 533
例外マッピングに対するレコード入出力エラー・マクロ 535
シグナル処理アクションの定義 536
  i5/OS 例外マッピングに対するシグナル . . . 538
ハンドラー理由コードの取り消し. 539
例外クラス 540
データ型の互換性 542
ランタイムの文字セット. 548
CCSID およびロケールの理解 549
  文字および文字ストリングの CCSID 549
  ワイド文字 553
非同期シグナル・モデル. 555
Unicode のサポート 557
  Unicode サポートを使用する理由 557
  疑似 CCSID ニュートラル 558
  他の ILE 言語からの Unicode 558
  標準ファイル 561
  考慮事項 562
  ファイルのデフォルト CCSID 563
  改行文字 563
  変換エラー 563
| ヒープ・メモリー 564
| ヒープ・メモリーの概要. 564
| ヒープ・メモリー・マネージャー. 565
| デフォルト・メモリー・マネージャー 566
| 高速プール・メモリー・マネージャー 569
| デバッグ・メモリー・マネージャー 573
| 環境変数 577
| C2M1211/C2M1212 メッセージの問題の診断 . . 579

付録 A. ライブラリー関数および拡張機能 583

標準 C ライブラリー関数表 (名前順) 583
ILE C ライブラリーの C ライブラリー関数への拡張機能に関する表 598

付録 B. 特記事項 603

プログラミング・インターフェース情報 604
商標 605

参考文献. 607

索引 609

表

1. グループ化の例 190
2. 通貨フォーマットの例 190
3. 通貨フィールド 190
4. 精度の値 245
5. Errno の値 300
6. strcasecmp() の戻り値 373
7. フラグ 385
8. 変換文字 386
9. strncasecmp() の戻り値 393
10. __wcsicmp() の戻り値 481
11. __wcsicmp() の戻り値 489
12. errno マクロ 531
13. 統合ファイル・システム対応 C ストリーム入出力の errno 値 533
14. 例外マッピングに対するレコード入出力エラー・マクロ 535
15. シグナル値の処理アクション定義 536
16. シグナル値のデフォルト・アクション 537
17. i5/OS 例外マッピングに対するシグナル 538
18. 取り消された呼び出し理由コードの判別 539
19. 呼び出しの取り消しに対する共通の理由コード 540
20. 例外クラス 541
21. ILE C データ型と ILE RPG との互換性 542
22. ILE C データ型と ILE COBOL との互換性 543
23. ILE C データ型と ILE CL との互換性 544
24. ILE C データ型と OPM RPG/400 との互換性 545
25. ILE C データ型と OPM COBOL/400 との互換性 545
26. ILE C データ型と CL との互換性 547
27. コマンド行の CL 呼び出しから ILE C プログラムへ渡される引数 547
28. コンパイル済みの制御言語プログラムから ILE C プログラムへ渡される CL 定数 . . . 547
29. コンパイル済みの制御言語プログラムから ILE C プログラムへ渡される CL 変数 . . . 548
30. インバリアント文字 548
31. さまざまな CCSID でのバリアント文字 549
32. 使用するメモリー・マネージャーを示す環境変数 577
33. デフォルト・メモリー・マネージャー・オプション 577
34. 高速プール・メモリー・マネージャー・オプション 578
35. デバッグ・メモリー・マネージャー・オプション 578
36. 標準 C ライブラリー関数 583
37. ILE C ライブラリー拡張機能 599

© Copyright IBM Corp. 1999, 2010

本書について

本書には、以下の参照情報が記載されています。

- インクルード・ファイル
- ランタイム関数
- ランタイムの考慮事項

Integrated Language Environment® (ILE) の C アプリケーションおよび C++ アプリケーションを作成する際に、本書を参照として使用してください。

C または C++ のプログラミング言語でプログラムする方法や ILE についての概念などは、本書では説明していません。以下の資料を、手引きとしてご利用ください。

- *C/C++ Legacy Class Libraries Reference*, SC09-7652-00
- *ILE Concepts*, SC41-5606-09
- *ILE C/C++ for AS/400 MI Library Reference*, SC09-2418-00
- *Standard C/C++ Library Reference*, SC09-4949-01
- *IBM Rational Development Studio for i: ILE C/C++ コンパイラー参照*, SC88-4025-02
- *IBM Rational Development Studio for i: ILE C/C++ 解説書*, SC88-4026-02
- *IBM Rational Development Studio for i: ILE C/C++ Programmer's Guide*, SC09-2712-07

その他の前提条件および関連情報については、 xii ページの『前提条件および関連情報』および 607 ページの『参考文献』を参照してください。

本書の対象読者

本書は、C/C++ プログラミング言語を熟知されたプログラマー、および ILE C/C++ アプリケーションの書き込みまたは保守を行うプログラマーを対象としています。適用できる IBM® i メニュー、表示、または制御言語 (CL) コマンドの使用経験が必要です。「*ILE Concepts*」のマニュアルで説明されている 統合言語環境 の知識も必要になります。

例についての注記

本書の例は、ライブラリー関数の使用方法について説明しており、平易な文体で書かれています。 C/C++ 言語構成の使用については、これらの例ですべて説明されているわけではありません。一部の例では、コードの一部分だけが示されていて、コードを追加しないとコンパイルできないものもあります。これらの例ではすべて、C ロケールの使用を想定しています。

ライブラリー関数の完全な実行可能例のすべてと、マシン・インターフェースの命令については、ソース・ファイル QACSRC のライブラリー QCPPLE 内に置かれています。各例の名前は、関数名または命令名と同じになります。例えば、本書で `_Rcommit()` 関数の使用方法を説明している例のソース・コードは、ライブラリー QCPPLE、ファイル QACSRC、メンバー RCOMMIT 内にあります。 QSYSINC ライブラリーをインストールしておく必要があります。

前提条件および関連情報

IBM i の技術情報を確認する場合には、まず IBM i Information Center を使用してください。

Information Center には次の Web サイトからアクセスできます。

`http://www.ibm.com/systems/i/infocenter/`

IBM i Information Center には、新規および更新済みのシステム情報 (ソフトウェアのインストール、Linux®、WebSphere®、Java™、高可用性、データベース、論理区画、CL コマンド、およびシステム・アプリケーション・プログラミング・インターフェース (API) など) が記載されています。さらに、システムのハードウェアとソフトウェアの計画、トラブルシューティング、および構成を支援するための、アドバイスや検索の機能も提供しています。

すべての新規のハードウェアのオーダーで、*System i Access for Windows DVD*, SK3T-4098が届けられます。この DVD によって、IBM i Access for Windows ライセンス・プログラムをインストールできます。IBM i Access Family は、PC を IBM i モデルに接続するための、クライアント機能およびサーバー機能を備えています。

その他の関連情報については、 607 ページの『参考文献』を参照してください。

変更内容の要約

本書の情報に関する変更内容は、以下のとおりです。

- ヒープ・メモリーに関する新しいセクションが追加されました。 564 ページの『ヒープ・メモリー』を参照してください。
- 以下の関数の説明が更新されました。
  - calloc() (58 ページの『calloc() ― ストレージの予約と初期化』を参照してください。)
  - _C_Quickpool_Debug() (69 ページの『_C_Quickpool_Debug() ― 高速プール・メモリー・マネージャー特性の変更』を参照してください。)
  - _C_Quickpool_Init() (71 ページの『_C_Quickpool_Init() ― 高速プール・メモリー・マネージャーの初期化』を参照してください。)
  - _C_Quickpool_Report() (73 ページの『_C_Quickpool_Report() ― 高速プール・メモリー・マネージャー・レポートの生成』を参照してください。)
  - _C_TS_malloc_debug() (82 ページの『_C_TS_malloc_debug() ― 使用されるテラスペース・メモリー量の判別 (オプションのダンプおよび検査を使用)』を参照してください。)
  - _C_TS_malloc_info() (84 ページの『_C_TS_malloc_info() ― 使用されるテラスペース・メモリー量の判別』を参照してください。)
  - free() (134 ページの『free() ― ストレージ・ブロックの解放』を参照してください。)
  - malloc() (203 ページの『malloc() ― ストレージ・ブロックの予約』を参照してください。)
  - Rclose() (270 ページの『_Rclose() ― ファイルのクローズ』を参照してください。)
  - realloc() (276 ページの『realloc() ― 予約ストレージ・ブロック・サイズの変更』を参照してください。)
  - regcomp() (279 ページの『regcomp() ― 正規表現のコンパイル』を参照してください。)
  - Rreadd() (308 ページの『_Rreadd() ― 相対レコード番号によるレコードの読み取り』を参照してください。)
  - Rreadf() (310 ページの『_Rreadf() ― 最初のレコードの読み取り』を参照してください。)
  - Rreadk() (314 ページの『_Rreadk() ― キーによるレコードの読み取り』を参照してください。)
  - Rreadl() (318 ページの『_Rreadl() ― 最終レコードの読み取り』を参照してください。)
  - Rreadn() (319 ページの『_Rreadn() ― 次のレコードの読み取り』を参照してください。)
  - Rreadp() (324 ページの『_Rreadp() ― 前のレコードの読み取り』を参照してください。)
  - Rreads() (326 ページの『_Rreads() ― 同じレコードの読み取り』を参照してください。)
  - strtok() (415 ページの『strtok() ― ストリングのトークン化』を参照してください。)

© Copyright IBM Corp. 1999, 2010

第 1 部 ランタイム・ライブラリー関数

© Copyright IBM Corp. 1999, 2010

第 1 章 インクルード・ファイル

インクルード・ファイルには、ランタイム・ライブラリーが提供され、マクロ定義と定数定義、型定義、および関数宣言が組み込まれています。一部の関数では、正常に作動させるために、インクルード・ファイルからの定義と宣言が必要になります。ファイルが提供する必要なステートメントが、ソースに直接コーディングされている場合は、ファイルのインクルードはオプションです。

この章では、各インクルード・ファイルに関する説明と、その内容の紹介を行い、ファイルで宣言されている関数のリストを記載します。

i5/OS® オペレーティング・システムに、QSYSINC (System Openness Includes) ライブラリーをインストールしておく必要があります。QSYSINC には、システム API、Dynamic Screen Manager (DSM)、ILE ヘッダー・ファイルなど、C/C++ ユーザーに役立つインクルード・ファイルが含まれています。 QSYSINC ライブラリーには、マシン・インターフェース (MI) 組み込み用と ILE C/C++ MI 関数用のプロトタイプおよびテンプレートが組み込まれた、ヘッダー・ファイルが含まれています。これらのヘッダー・ファイルの詳細は、「*ILE C/C++ for AS/400 MI Library Reference*」を参照してください。

<assert.h>

`<assert.h>` インクルード・ファイルは、`assert` マクロを定義します。 `assert` を使用するときには、`assert.h` を設定する必要があります。

`assert` の定義は、`#ifndef` プリプロセッサー・ブロック内にあります。 `#define` ディレクティブを通じて、またはコンパイル・コマンドを使用して、ID NDEBUG を定義していない場合は、`assert` マクロがアサーションの式をテストします。アサーションが false の場合、システムは `stderr` へメッセージを出力し、プログラムの異常終了シグナルを発します。アサーションが false の場合、システムではダンプ・ジョブ (DMPJOB) OUTPUT(*PRINT) も実行します。

NDEBUG が定義されている場合、`assert` は何も行わないように定義されます。 NDEBUG を定義することにより、プログラム・アサーションを抑制することができます。

<ctype.h>

`<ctype.h>` インクルード・ファイルは、文字分類に使用される関数を定義します。 `<ctype.h>` で定義される関数は、以下のとおりです。

isascii[1]	isblank[2]	isgraph	ispunct	toascii[1]
isalnum	iscntrl	islower	isspace	tolower
isalpha	isdigit	isprint	isupper	toupper
			isxdigit	

注: [1] コンパイル・コマンドで LOCALETYPE(*CLD) が指定されている場合、これらの関数は使用できません。
注: [2] この関数は C++ のみに適用できます。

<decimal.h>

<decimal.h> インクルード・ファイルには、パック 10 進数 型およびその属性の範囲を指定する、定数の定義が含まれています。 decimal、digitsof、precisionof などのキーワードを使用する場合、<decimal.h> ファイルをソース・コードの #include ディレクティブに付属させる必要があります。

<errno.h>

<errno.h> インクルード・ファイルは、errno 変数に設定されるマクロを定義します。 <errno.h> インクルード・ファイルは、C ライブラリー関数のエラー・レポートに使用される値のマクロを定義し、errno マクロを定義します。 errno には整数値を割り当てることができ、その値を実行時にテストすることができます。現在の errno 値の表示に関する情報については、「*IBM Rational Development Studio for i: ILE C/C++ Programmer's Guide*」の『Checking the Errno Value』を参照してください。

注: ライブラリー関数呼び出しの後で errno の値をテストする場合は、呼び出し時にこの値がリセットされないことがあるため、呼び出し前にこの値を 0 に設定してください。

<except.h>

<except.h> インクルード・ファイルは、ILE C 例外処理で使用される型およびマクロを宣言します。

_INTRPT_Hndlr_Parms_T の定義は、以下のとおりです。

```
typedef _Packed struct {
  unsigned int    Block_Size;
  _INVFLAGS_T     Tgt_Flags;
  char            reserved[8];
  _INVPTR         Target;
  _INVPTR         Source;
  _SPCPTR         Com_Area;
  char            Compare_Data[32];
  char            Msg_Id[7];
  char            reserved1;
  _INTRPT_Mask_T  Mask;
  unsigned int    Msg_Ref_Key;
  unsigned short  Exception_Id;
  unsigned short  Compare_Data_Len;
  char            Signal_Class;
  char            Priority;
  short           Severity;
  char            reserved3[4];
  int             Msg_Data_Len;
  char            Mch_Dep_Data[10];
  char            Tgt_Inv_Type;
  _SUSPENDPTR     Tgt_Suspend;
  char            Ex_Data[48];
} _INTRPT_Hndlr_Parms_T;
```

エレメント
  説明

Block_Size
  例外ハンドラーに渡されるパラメーター・ブロックのサイズ。

Tgt_Flags
  システムが使用するフラグが含まれています。

reserved
  8 バイトの予約フィールド。

Target 呼び出しスタック項目の呼び出しポインターのうち、例外ハンドラーを使用可能にしたもの。

Source 呼び出しスタック項目の呼び出しポインターのうち、例外を発生させたもの。その呼び出しスタック項目がすでに存在しない場合、例外の処理時に制御が再開されると、これが呼び出しスタック項目のポインターになります。

Com_Area
#プラグマの exception_handler で 2 番目のパラメーターとして指定された、通信域変数へのポインター。通信域が指定されなかった場合、この値は NULL になります。

Compare_Data
比較データ (CPF、MCH など)。4 バイトのメッセージ接頭語のあとに、関連メッセージのメッセージ・データから取得された 28 バイトのメッセージが続いたもので構成されています。メッセージ・データが 28 よりも大きい場合、これらが最初の 28 バイトに相当します。 MCH メッセージの場合、これらがシステムが戻す例外の関連データ (置換テキスト) の最初の 28 バイトに相当します。

Msg_Id
メッセージ ID (例: CPF123D)。*STATUS メッセージ型は、このフィールドでは更新されません。

reserved1
1 バイトの埋め込み。

Mask 発生した例外 (10 進数データ・エラーなど) の型を識別する、8 バイトの例外マスク。使用可能な型は、 541 ページの表 20 に記載されています。

Msg_Ref_Key
メッセージを一意的に識別するために使用されるキー。

Exception_Id
例外 ID の 2 進値 (例: 0x123D)。値を表示するには、変換指定子 %x を使用して、16 進値で情報が保管されるようにします。

Compare_Data_Len
比較データの長さ。

Signal_Class
内部シグナル・クラス。

Priority
ハンドラーの優先度。

Severity
メッセージ重大度。

reserved3
4 バイトの予約フィールド。

Msg_Data_Len
使用可能なメッセージ・データの長さ。

Mch_Dep_Data
マシン依存のデータ。

Tgt_Inv_Type
呼び出しの型。マクロの定義は <mimchobs.h> で行います。

Tgt_Suspend
ターゲットの中断ポインター。

Ex_Data
例外データの最初の 48 バイト。

_CNL_Hndlr_Parms_T の定義は以下のとおりです。

```
typedef _Packed struct {
  unsigned int    Block_Size;
  _INVFLAGS_T     Inv_Flags;
  char            reserved[8];
  _INVPTR         Invocation;
  _SPCPTR         Com_Area;
  _CNL_Mask_T     Mask;
} _CNL_Hndlr_Parms_T;
```

エレメント
説明

Block_Size
取り消しハンドラーに渡されるパラメーター・ブロックのサイズ。

Inv_Flags
システムが使用するフラグが含まれています。

reserved
8 バイトの予約フィールド。

Invocation
取り消し中の呼び出しの呼び出しポインター。

Com_Area
取り消しハンドラーが定義したハンドラー通信域のポインター。

Mask 取り消し理由を表す 4 バイトの値。

次の組み込みは <except.h> で定義されます。

組み込み
説明

__EXBDY
__EXBDY 組み込みまたは _EXBDY マクロの目的は、例外依存操作の境界として機能することにあります。例外依存操作を行うと、例外がシグナル通知される場合があります。 EXBDY を使用すると、コード動作を行う最適化をプログラマーが選択的に抑制できるようになります。例えば、除算はゼロ除算をシグナル通知する可能性があるため、例外依存操作となります。 EXBDY と除算の両方を含む実行パスは、2 つを同じ順序で実行します。その際、最適化が行われると場合と、行われない場合とがあります。以下に例を示します。

```
b = exp1;
c = exp2;
...
_EXBDY();
a = b/c;
```

__VBDY
__VBDY 組み込みまたは _VBDY マクロの目的は、例外パスで使用される可能性のある変数用に、現在のホームの保管場所が設定されていることを確認することにあります。これによって、例外ハンドラーの変数の現行値における可視性が確保されます。 VBDY によって、プログラマーが選択的に最適化を抑止するようにできます。一貫した順序で変数更新を実行するための、冗長ストア除去および転送ストア動作などが、これに当たります。次の例では、VBDY を使って、各コードのブロックが例外をシグナル通知する前に、状態がホーム保管場所になるようにしています。

VBDY は、もっぱら EXBDY と組み合わせて使用されます。これによって、先に状態変数への割り当てを行い、ホームの保管場所を確実に更新し、さらに後に行う例外依存操作がこれらの割り当て以前に移動されないようにします。

```
 state = 1;
 _VBDY();
 /* Do stuff that may signal an exception.         */
 state = 2;
 _VBDY();
 /* More stuff that may signal an exception.       */
state = 3;
_VBDY();
```

組み込みについて詳しくは、「*ILE C/C++ for AS/400 MI Library Reference*」を参照してください。

<float.h>

`<float.h>` インクルード・ファイルは、2 進浮動小数点のデータ型の範囲を指定する定数を定義します。例えば、`double` 型のオブジェクトの最大桁数、`float` 型のオブジェクトの最小指数、などがこれに該当します。さらに、マクロ変数 __STDC_WANT_DEC_FP__ が定義されている場合は、10 進浮動小数点データ型の範囲を指定する定数も、このインクルード・ファイルによって定義されます。例えば、`_Decimal64` 型のオブジェクトの最大桁数、`_Decimal32` 型のオブジェクトの最小指数、などがこれに該当します。

<inttypes.h>

`<inttypes.h>` インクルード・ファイルには `<stdint.h>` が含まれており、これが他の機能と共に拡張されます。

次のマクロは、書式指定子用に定義されています。これらのマクロは、C プログラム用に定義されています。 `<inttypes.h>` がインクルードされる前に __STDC_FORMAT_MACROS が定義されたときにのみ、これらは C++ 用に定義されます。

PRId8
PRId16
PRId32
PRId64
PRIdFAST8
PRIdFAST16
PRIdFAST32
PRIdFAST64
PRIdLEAST8
PRIdLEAST16
PRIdLEAST32
PRIdLEAST64
PRIdMAX
PRIi8
PRIi16
PRIi32
PRIi64
PRIiFAST8
PRIiFAST16
PRIiFAST32
PRIiFAST64
PRIiLEAST8
PRIiLEAST16
PRIiLEAST32
PRIiLEAST64
PRIiMAX
PRIo8
PRIo16
PRIo32
PRIo64
PRIoFAST8
PRIoFAST16
PRIoFAST32
PRIoFAST64
PRIoLEAST8
PRIoLEAST16
PRIoLEAST32
PRIoLEAST64
PRIoMAX
PRIu8
PRIu16
PRIu32
PRIu64
PRIuFAST8
PRIuFAST16
PRIuFAST32
PRIuFAST64
PRIuLEAST8
PRIuLEAST16
PRIuLEAST32
PRIuLEAST64
PRIuMAX
PRIx8
PRIx16
PRIx32
PRIx64
PRIxFAST8
PRIxFAST16
PRIxFAST32
PRIxFAST64
PRIxLEAST8
PRIxLEAST16
PRIxLEAST32
PRIxLEAST64
PRIxMAX
PRIX8
PRIX16
PRIX32
PRIX64
PRIXFAST8
PRIXFAST16
PRIXFAST32
PRIXFAST64
PRIXLEAST8
PRIXLEAST16
PRIXLEAST32
PRIXLEAST64
PRIXMAX
SCnd16
SCnd32
SCnd64
SCndFAST16
SCndFAST32
SCndFAST64
SCndLEAST16
SCndLEAST32
SCndLEAST64
SCndMAX
SCNo16
SCNo32
SCNo64
SCNoFAST16
SCNoFAST32
SCNoFAST64
SCNoLEAST16
SCNoLEAST32
SCNoLEAST64
SCNoMAX
SCNu16
SCNu32
SCNu64
SCNuFAST16
SCNuFAST32
SCNuFAST64
SCnuLEAST16
SCnuLEAST32
SCnuLEAST64
SCnuMAX
SCnx16
SCnx32
SCnx64
SCnxFAST16
SCnxFAST32
SCnxFAST64
SCnxLEAST16
SCnxLEAST32
SCnxLEAST64
SCnxMAX

<langinfo.h>

<langinfo.h> インクルード・ファイルには、nl_langinfo が使用する宣言および定義が含まれています。

<limits.h>

<limits.h> インクルード・ファイルは、整数データおよび文字データの型の範囲を指定する定数を定義します。例えば、char 型のオブジェクトの最大値が、これに該当します。

<locale.h>

<locale.h> インクルード・ファイルは setlocale()、localeconv()、および wcslocaleconv() の各ライブラリー関数を宣言します。国際市場向けのアプリケーションの作成時に C ロケールを変更する際に、これらの関数が役立ちます。

<locale.h> インクルード・ファイルは、struct lconv 型の宣言も行うとともに、次のマクロ定義も宣言します。

NULL
LC_ALL
LC_C
LC_C_FRANCE
LC_C_GERMANY
LC_C_ITALY
LC_C_SPAIN
LC_C_UK
LC_C_USA
LC_COLLATE
LC_CTYPE
LC_MESSAGES
LC_MONETARY
LC_NUMERIC
LC_TIME
LC_TOD
LC_UCS2_ALL
LC_UCS2_COLLATE
LC_UCS2_CTYPE
LC_UNI_ALL

LC_UNI_COLLATE	LC_UNI_CTYPE	LC_UNI_TIME	LC_UNI_NUMERIC
LC_UNI_MESSAGES	LC_UNI_MONITARY	LC_UNI_TOD	

<math.h>

<math.h> インクルード・ファイルは、浮動小数点の数学関数をすべて宣言します。

acos	cosh	frexp	nextafter	sqrt
asin	erf	gamma	nextafterl	tan
atan	erfc	hypot	nexttoward	tanh
atan2	exp	ldexp	nexttowardl	
Bessel	fabs	log	pow	
ceil	floor	log10	sin	
cos	fmod	modf	sinh	

注:

1. ベッセル関数は、j0、j1、jn、y0、y1、および yn の各関数のグループを指します。
2. 浮動小数点数は、15 桁の有効数字のみで保証されます。複数の浮動小数点数が計算に使用されている場合、期待される結果に大きな影響を与えます。

<math.h> は、HUGE_VAL マクロを定義します。このマクロは、正の double 式に展開されますが、無限大をサポートするシステムでは無限大になる場合があります。

すべての数学関数において、入力引数がその関数に許可された値の範囲外である場合に、**ドメイン・エラー**が発生します。ドメイン・エラーが起こると、errno が EDOM の値に設定されます。

関数の値が double 値で表せない場合、範囲エラーが発生します。結果の絶対値が大きすぎる (オーバーフローしている) 場合、HUGE_VAL マクロの正または負の値が関数によって戻され、errno が ERANGE に設定されます。結果が小さすぎる (アンダーフローしている) 場合、関数によってゼロが戻されます。

<mallocinfo.h>

_C_TS_malloc_info および _C_TS_malloc_debug のインクルード・ファイルです。

<monetary.h>

<monetary.h> ヘッダー・ファイルには、通貨量の出力に関連する宣言および定義が含まれています。次の通貨関数は、strfmon() および wcsfmon() に定義されます。コンパイル・コマンドで LOCALETYPE(*CLD) が指定されている場合、strfmon() 関数は使用できません。 wcsfmon() 関数が使用できるのは、コンパイル・コマンドで LOCALETYPE(*LOCALEUTF) が指定されている場合のみになります。

<nl_types.h>

<nl_types.h> ヘッダー・ファイルには、カタログ定義と以下のカタログ関数が含まれています: catclose()、catgets()、catopen()。 LOCALETYPE(*CLD) または SYSIFCOPT(*NOIFSIO) のいずれかがコンパイル・コマンドで指定されている場合、これらの定義は使用できません。

<pointer.h>

<pointer.h> インクルード・ファイルには、i5/OS ポインター型用の typedefs ディレクティブおよび pragma ディレクティブが含まれています。該当するポインターには、スペース・ポインター、オープン・ポインター、呼び出しポインター、ラベル・ポインター、システム・ポインター、および中断ポインターがあります。 typedefs _ANYPTR および _SPCPTRCN も、<pointer.h> で定義されます。

<recio.h>

<recio.h> インクルード・ファイルは、すべての ILE C レコード入出力 (I/O) 操作の型、マクロ、およびプロトタイプ関数を定義します。

次の関数は <recio.h> で定義されます。

_Racquire	_Rclose	_Rcommit	_Rdelete
_Rdevatr	_Rfeod	_Rfeov	_Rformat
_Rindara	_Riofbk	_Rlocate	_Ropen
_Ropnfbk	_Rpgmdev	_Rreadd	_Rreadf
_Rreadindv	_Rreadk	_Rreadl	_Rreadn
_Rreadnc	_Rreadp	_Rreads	_Rrelease
_Rrlslck	_Rrollbck	_Rupdate	_Rupfb
_Rwrite	_Rwrited	_Rwriterd	_Rwrread

次の位置決めマクロは、recio.h で定義されます。

__END	__END_FRC	__FIRST	__KEY_EQ
__KEY_GE	__KEY_GT	__KEY_LE	__KEY_LT
__KEY_NEXTEQ	__KEY_NEXTUNQ	__KEY_PREVEQ	__KEY_PREVUNQ
__KEY_LAST	__KEY_NEXT	__NO_POSITION	__PREVIOUS
__PRIOR	__RRN_EQ	__START	__START_FRC
__LAST	__NEXT		

次のマクロは、recio.h で定義されます。

__DATA_ONLY	__DFT	__NO_LOCK	__NULL_KEY_MAP

次の方向マクロは、recio.h で定義されます。

__READ_NEXT	__READ_PREV

次の関数およびマクロは、位置指定モードまたは移動モードをサポートしています。

_Rreadd	_Rreadf	_Rreadindv	_Rreadk
_Rreadl	_Rreadn	_Rreadnc	_Rreadp
_Rreads	_Rupdate	_Rwrite	_Rwrited
_Rwriterd	_Rwrread		

バッファー・パラメーターを含むレコード入出力関数は、移動モードまたは位置指定モードで機能します。移動モードでは、ユーザー提供のバッファーとシステム・バッファーの間で、データが移動します。位置指定モードでは、ユーザーはシステム・バッファーでデータにアクセスする必要があります。システム・バッ

ファーのポインターは、_RFILE 構造体で公開されています。位置指定モードを使用するように指定するには、レコード入出力関数のバッファー・パラメーターを NULL にコード化します。

関数の数値には、サイズ・パラメーターが含まれています。移動モードの場合、これはユーザー提供のバッファーとシステム・バッファーの間でコピーされるデータ・バイト数になります。すべてのレコード入出力関数では、指定されたサイズにかかわらず、一度に 1 レコードを処理します。このレコードのサイズは、ファイル記述により定義されます。これは、レコード入出力関数の呼び出しでユーザーが定義するサイズ・パラメーターと等しくならない場合があります。バッファー間を移動するデータ量は、現行のレコード形式または指定された最小サイズのうち、小さいほうのレコード長と等しくなります。このサイズ・パラメーターは、位置指定モードでは無視されます。

次の型は recio.h で定義されます。

オープンされたレコード入出力操作を制御するための情報

```
typedef _Packed struct {
  char                         reserved1[16];
  volatile void  *const *const in_buf;
  volatile void  *const *const out_buf;
  char                         reserved2[48];
  _RIOFB_T                     riofb;
  char                         reserved3[32];
  const unsigned int           buf_length;
  char                         reserved4[28];
  volatile char  *const        in_null_map;
  volatile char  *const        out_null_map;
  volatile char  *const        null_key_map;
  char                         reserved5[48];
  const int                    min_length;
  short                        null_map_len;
  short                        null_key_map_len;
  char                         reserved6[8];
}_RFILE;
```

エレメント	説明
in_null_map	データベース・ファイルからの読み取り時に、どちらのフィールドを NULL と見なすかを指定します。
out_null_map	データベース・ファイルへの書き込み時に、どちらのフィールドを NULL と見なすかを指定します。
null_key_map	データベースをキーで読み取っている場合に、どちらのフィールドに NULL を組み込むかを指定します。
null_map_len	in_null_map および out_null_map の長さを指定します。
null_key_map_len	null_key_map の長さを指定します。

レコード入出力のフィードバック情報

```
typedef struct {
  unsigned char    *key;
  _Sys_Struct_T    *sysparm;
  unsigned long    rrn;
  long             num_bytes;
  short            blk_count;
  char             blk_filled_by;
  int              dup_key    :1;
  int              icf_locate :1;
  int              reserved1 :6;
  char             reserved2[20];
}_RIOFB_T;
```

エレメント	説明
key	キー・シーケンス・アクセス・パスを使用してファイルを処理している場合、配置、読み取り、および書き込みが正常に行われたレコードのキー値のポインターが、このフィールドに入ります。
sysparm	このフィールドは、ICF ファイル、ディスプレイ・ファイル、およびプリンター・ファイルのメジャー戻りコードおよびマイナー戻りコードのポインターです。
rrn	このフィールドには、配置、読み取り、書き込みが正常に行われたレコードの相対レコード番号が入ります。
num_bytes	このフィールドには、読み取りまたは書き込みされたバイト数が入ります。
blk_count	このフィールドには、ブロック内に残っているレコード数が入ります。ファイルが入力用にオープンされている場合、blkrcd=y が指定され、読み取り関数が呼び出されて、ブロック内に残っているレコード数によって、このフィールドが更新されます。
blk_filled_by	このフィールドは、ブロックを埋める処理を表します。ファイルが入力用にオープンされている場合、blkrcd=y が指定され、読み取り関数が呼び出されます。 _Rreadn 関数がブロックを埋めた場合、このフィールドには __READ_NEXT マクロが設定されます。_Rreadp 関数がブロックを埋めた場合には、__READ_PREV マクロが設定されます。

システム固有の情報

```
typedef struct {
  void              *sysparm_ext;
  _Maj_Min_rc_T    _Maj_Min;
  char              reserved1[12];
} _Sys_Struct_T;
```

メジャー戻りコードおよびマイナー戻りコード

```
 typedef struct  {
       char    major_rc[2];
       char    minor_rc[2];
 } _Maj_Min_rc_T;
```

次のマクロは、`recio.h` で定義されます。

_FILENAME_MAX	最も長いファイル名を保持するのに十分大きな文字配列のサイズを意味する、整数定数式に展開されます。これは、ストリーム入出力マクロと同じです。
_ROPEN_MAX	同時にオープン可能な最大ファイル数を意味する、整数定数式に展開されます。

次の NULL フィールド・マクロは、`recio.h` で定義されます。

エレメント
説明

_CLEAR_NULL_MAP(file, type)
NULL 出力フィールド・マップをクリアして、*file* に書き込まれる NULL フィールドがレコード内に存在しないことを示す。 *type* は、現行レコード・フォーマットの NULL フィールド・マップに対応する typedef を指します。

_CLEAR_UPDATE_NULL_MAP(file, type)
NULL 入力フィールド・マップをクリアして、*file* に書き込まれるレコード内に NULL フィールドが存在しないことを示す。 *type* は、現行レコード・フォーマットの NULL フィールド・マップに対応する typedef を指します。

_QRY_NULL_MAP(file, type)
前に読み取られたレコードで NULL であったフィールドの数を戻します。*type* は、現行レコード・フォーマットの NULL フィールド・マップに対応する typedef を指します。

_CLEAR_NULL_KEY_MAP(file, type)
NULL キー・フィールド・マップをクリアして、*file* に書き込まれるレコード内に NULL キー・フィールドが存在しないことを示します。 *type* は、現行レコード・フォーマットの NULL キー・フィールド・マップに対応する typedef を指します。

_SET_NULL_MAP_FIELD(file, type, field)
出力 NULL フィールド・マップ内の指定した *field* を設定して、レコードが *file* に書き込まれる際に *field* が NULL とみなされるようにします。

_SET_UPDATE_NULL_MAP_FIELD(file, type, field)
入力 NULL フィールド・マップ内の指定した *field* を設定して、レコードが *file* に書き込まれる際に *field* が NULL とみなされるようにします。 *type* は、レコード・フォーマットの NULL キー・フィールド・マップに対応する typedef を指します。

_QRY_NULL_MAP_FIELD(file, type, field)
前の読み取りレコードで *field* が NULL と見なされるように、NULL 入力フィールド・マップ内の指定された *field* が設定されている場合は、 1 を戻します。フィールドが NULL でない場合は、ゼロを戻します。 *type* は、現行レコード・フォーマットの NULL キー・フィールド・マップに対応する typedef を指します。

_SET_NULL_KEY_MAP_FIELD(file, type, field)
指定されたフィールド・マップを指定して、レコードが *file* から読み取られる際に *field* が NULL と見なされるようにします。 *type* は、現行レコード・フォーマットの NULL キー・フィールド・マップに対応する typedef を指します。

_QRY_NULL_KEY_MAP(file, type)
前に読み取られたレコードのキーで NULL だったフィールドの数を戻します。 *type* は、現行レコード・フォーマットの NULL フィールド・マップに対応する typedef を指します。

_QRY_NULL_KEY_MAP_FIELD(file, type, field)
前の読み取りレコードにおいて *field* は NULL と見なされることを、NULL キー・フィールド・マップで指定されたフィールドが示す場合に、1 を戻します。 *field* が NULL でない場合、ゼロを戻します。 *type* は、現行レコード・フォーマットの NULL キー・フィールド・マップに対応する typedef を指します。

<regex.h>

<regex.h> インクルード・ファイルは、以下の正規表現関数を定義します。

`regcomp()`	`regerror()`	`regexec()`	`regfree()`

<regex.h> インクルード・ファイルは、コンパイル済みの正規表現を保管できる *regmatch_t* 型、*regex_t* 型を宣言し、さらに以下のマクロも宣言します。

`regcomp()` 関数の *cflags* パラメーターの値。

REG_BASIC
REG_EXTENDED
REG_ICASE
REG_NEWLINE
REG_NOSUB

`regexec()` 関数の *eflags* パラメーターの値。

REG_NOTBOL
REG_NOTEOL

`regerror()` 関数の *errcode* パラメーターの値。

REG_NOMATCH
REG_BADPAT
REG_ECOLLATE
REG_ECTYPE
REG_EESCAPE
REG_ESUBREG
REG_EBRACK
REG_EPAREN
REG_EBRACE
REG_BADBR
REG_ERANGE
REG_ESPACE
REG_BADRPT
REG_ECHAR
REG_EBOL
REG_EEOL
REG_ECOMP
REG_EEXEC
REG_LAST

コンパイル・コマンドで LOCALETYPE(*CLD) が指定されている場合、これらの宣言および定義は使用できません。

<setjmp.h>

`<setjmp.h>` インクルード・ファイルは、`setjmp()` 関数および `longjmp()` 関数を宣言します。また、プログラム状態の保存および復元を行う際に `setjmp()` 関数および `longjmp()` 関数が使用するバッファー・タイプ (`jmp_buf`) の定義も行います。

<signal.h>

<signal.h> インクルード・ファイルは、シグナルの値を定義し、signal() 関数および raise() 関数を宣言します。

<signal.h> インクルード・ファイルは、以下のマクロの定義も行います。

SIGABRT	SIG_ERR	SIGILL	SIGOTHER	SIGUSR1
SIGALL	SIGFPE	SIGINT	SIGSEGV	SIGUSR2
SIG_DFL	SIG_IGN	SIGIO	SIGTERM	

<signal.h> は、i5/OS の C 標準ライブラリーへの拡張機能である _GetExcData 関数を宣言します。

<stdarg.h>

<stdarg.h> インクルード・ファイルは、可変長引数リスト (va_arg()、va_start()、および va_end()) を使って関数内の引数へのアクセスを可能にするマクロを定義します。 <stdarg.h> インクルード・ファイルは、va_list 型の定義も行います。

<stddef.h>

<stddef.h> インクルード・ファイルは、一般的に使用されるポインター、変数、および型を宣言します (以下のリストを参照)。

ptrdiff_t
異なる 2 つのポインターの型の typedef

size_t sizeof により戻される値の型の typedef

wchar_t
ワイド文字定数用の typedef。

<stddef.h> インクルード・ファイルは、NULL マクロおよび offsetof マクロを定義します。 NULL は、データ・オブジェクトを指すことのないポインターです。 offsetof マクロは、構造体メンバーと構造体の先頭との間のバイト数まで拡張します。offsetof マクロには、所定の形式があります。

```
offsetof(structure_type, member)
```

<stddef.h> インクルード・ファイルは、i5/OS の C への拡張機能である extern 変数の _EXCP_MSGID を宣言します。

<stdint.h>

<stdint.h> インクルード・ファイルは、幅が指定された整数型のセットを宣言し、対応するマクロのセットを定義します。他の標準的なインクルード・ファイルで定義された型に対応する整数型の制限を指定するマクロについても、定義を行います。

次にあるような、厳密に幅を指定した整数型が定義されます。

int8_t	int32_t	uint8_t	uint32_t
int16_t	int64_t	uint16_t	uint64_t

次にあるような、最小幅の整数型が定義されます。

```
int_least8_t     int_least32_t    uint_least8_t    uint_least32_t
int_least16_t    int_least64_t    uint_least16_t   uint_least64_t
```

次にあるような、最速最小幅の整数型が定義されます。

```
int_fast8_t      int_fast32_t     uint_fast8_t     uint_fast32_t
int_fast16_t     int_fast64_t     uint_fast16_t    uint_fast64_t
```

次にあるような、最大幅の整数型が定義されます。

```
intmax_t
uintmax_t
```

次のマクロは、厳密に幅を指定した整数型を制限するために定義されます (注記 1 (17 ページ) を参照)。

```
INT8_MAX         INT16_MIN        INT64_MAX        UINT16_MAX
INT8_MIN         INT32_MAX        INT64_MIN        UINT32_MAX
INT16_MAX        INT32_MIN        UINT8_MAX        UINT64_MAX
```

次のマクロは、最小幅の整数型を制限するために定義されます (注記 1 (17 ページ) を参照)。

```
INT_LEAST8_MAX   INT_LEAST16_MIN  INT_LEAST64_MIN  UINT_LEAST16_MAX
INT_LEAST8_MIN   INT_LEAST32_MAX  INT_LEAST64_MIN  UINT_LEAST32_MAX
INT_LEAST16_MAX  INT_LEAST32_MIN  UINT_LEAST8_MAX  UINT_LEAST64_MAX
```

次のマクロは、最速最小幅の整数型を制限するために定義されます (注記 1 (17 ページ) を参照)。

```
INT_FAST8_MAX    INT_FAST16_MIN   INT_FAST64_MIN   UINT_FAST16_MAX
INT_FAST8_MIN    INT_FAST32_MAX   INT_FAST64_MIN   UINT_FAST32_MAX
INT_FAST16_MAX   INT_FAST32_MIN   UINT_FAST8_MAX   UINT_FAST64_MAX
```

次のマクロは、最大幅の整数型を制限するために定義されます (注記 1 (17 ページ) を参照)。

```
INTMAX_MIN
INTMAX_MAX
UINTMAX_MAX
```

次のマクロは、その他の整数型を制限するために定義されます (注記 1 (17 ページ) を参照)。

```
PTRDIFF_MAX      SIG_ATOMIC_MIN   WCHAR_MIN
PTRDIFF_MIN      SIZE_MAX         WINT_MAX
SIG_ATOMIC_MAX   WCHAR_MAX        WINT_MIN
```

次のマクロは、最小幅の整数型の定数式用に定義されます (注記 2 (17 ページ) を参照)。

```
INT8_C           INT32_C          UINT8_C          UINT32_C
INT16_C          INT64_C          UINT16_C         UINT64_C
```

次のマクロは、最大幅の整数型の定数式用に定義されます (注記 2 を参照)。

```
INTMAX_C
UINTMAX_C
```

注:

1. これらのマクロは、C プログラム用に定義されています。 <stdint.h> がインクルードされる前に __STDC_LIMIT_MACROS が定義されたときにのみ、これらは C++ 用に定義されます。
2. これらのマクロは、C プログラム用に定義されています。 <stdint.h> がインクルードされる前に __STDC_CONSTANT_MACROS が定義されたときにのみ、これらは C++ 用に定義されます。

<stdio.h>

<stdio.h> インクルード・ファイルは、ストリーム入出力関数の定数、マクロ、および型を定義し、ストリーム入出力関数を宣言します。ストリーム入出力は、次のとおりです。

_C_Get_Ssn_Handle	fprintf	fwrite	remove	vfscanf
clearerr	fputc	fwscanf [1]	rename	vfwprintf [1]
fclose	_fputchar	getc	rewind	vfwscanf[1]
fdopen[2]	fputs	getchar	scanf	vprintf
feof	fputwc[1]	gets	setbuf	vscanf
ferror	fputws[1]	getwc [1]	setvbuf	vsscanf
fflush	fread	getwchar [1]	snprintf	vsnprintf
fgetc	freopen	perror	sprintf	vsprintf
fgetpos	fscanf	printf	sscanf	vwprintf [1]
fgets	fseek	putc	tmpfile	vwscanf[1]
fgetwc [1]	fsetpos	putchar	tmpnam	wfopen[2]
fgetws [1]	ftell	puts	ungetc	wprintf [1]
fileno[2]	fwide [1]	putwc [1]	ungetwc [1]	wscanf [1]
fopen	fwprintf [1]	putwchar [1]	vfprintf	

注: [1] コンパイル・コマンドで LOCALETYPE(*CLD) または SYSIFCOPT(*NOIFSIO) のいずれかが指定されている場合、これらの関数は使用できません。
注: [2] コンパイル・コマンドで SYSIFCOPT(*IFSIO) が指定されている場合に、これらの関数を使用できます。

<stdio.h> インクルード・ファイルは、以下にリストされたマクロも定義します。これらの定数を、プログラムで使用することができます。ただし、値は変更しないでください。

BUFSIZ ストリーム入出力用にバッファーを割り振る際に、setbuf ライブラリー関数が使用するバッファー・サイズを指定します。この値は、システム割り振りバッファーのサイズを設定し、setbuf とともに使用されます。

EOF ファイル終わり (場合によってはエラー) が検出されたときに、入出力関数によって戻される値。

FOPEN_MAX
同時にオープン可能なファイルの数。

FILENAME_MAX
サポートされる最長のファイル名。妥当な制限がない場合、FILENAME_MAX が推奨サイズです。

L_tmpnam
tmpnam 関数で生成可能な一時名の最長サイズ。

TMP_MAX
tmpnam 関数で生成可能な固有のファイル名の最小数。

NULL データ・オブジェクトを指すことのないポインター。

FILE 構造化型は <stdio.h> で定義されます。ストリーム入出力関数は、FILE 型のポインターを使用して、指定のストリームにアクセスします。システムは、ストリームを保守するのに FILE 構造体の情報を使用します。

統合ファイル・システムでコンパイル・パラメーター SYSIFCOPT(*IFSIO) が使用可能な場合、ifs.h が <stdio.h> にインクルードされます。

C 標準ストリームの stdin、stdout、および stderr も、<stdio.h> で定義されます。

SEEK_CUR、SEEK_END、および SEEK_SET の各マクロは整数定数式に展開され、fseek() の 3 番目の引数として使用することができます。

_IOFBF、_IOLBF、および _IONBF の各マクロは、setvbuf 関数の 3 番目の引数として使用するのに適切な、別個の値を持つ整数定数式に展開されます。

<stdio.h> で定義される fpos_t 型は fgetpos() および fsetpos() で使用されます。

NULL についての詳細は、 15 ページの『<stddef.h>』を参照してください。

<stdlib.h>

<stdlib.h> インクルード・ファイルは、次の関数を宣言します。

abort	_C_Quickpool_Report	ldiv	realloc	strtoul
abs	div	lldiv	srand	strtoull
atexit	exit	malloc	strtod	system
atof	free	mblen	strtod32	_ultoa[1]
atoi	_gcvt[1]	mbstowcs	strtod64	wcstombs
atol	getenv	mbtowc	strtod128	wctomb
bsearch	_itoa[1]	putenv	strtof	
calloc	_ltoa[1]	qsort	strtol	
_C_Quickpool_Debug	labs	rand	strtold	
_C_Quickpool_Init	llabs	rand_r	strtoll	

注: [1] これらの関数は、C++ にのみ適用できます。

<stdlib.h> インクルード・ファイルには、次のマクロの定義が含まれています。

NULL NULL ポインター値。

EXIT_SUCCESS
0 に展開。atexit 関数が使用します。

EXIT_FAILURE
8 に展開。atexit 関数が使用します。

RAND_MAX
: rand 関数が戻すことのできる最大数を表す整数に展開されます。

MB_CUR_MAX
: 現行ロケールのマルチバイト文字の最大バイト数を表す整数に展開されます。

NULL、size_t 型および wchar_t 型について詳しくは、 15 ページの『<stddef.h>』を参照してください。

<string.h>

<string.h> インクルード・ファイルは、ストリング処理関数を宣言します。

memchr	strcat	strcspn	strncmp	strset[1]
memcmp	strchr	strdup[1]	strncpy	strspn
memcpy	strcmp	strerror	strnicmp[1]	strstr
memicmp[1]	strcmpi[1]	stricmp[1]	strnset[1]	strtok
memmove	strcoll	strlen	strpbrk	strtok_r
memset	strcpy	strncat	strrchr	strxfrm

注: [1] これらの関数は、C++ プログラムに使用できます。 __cplusplus__strings__ マクロがプログラムで定義されている場合にのみ、C でも使用できます。

<string.h> インクルード・ファイルは、NULL マクロ、および size_t 型も定義します。

NULL および size_t 型について詳しくは、 15 ページの『<stddef.h>』を参照してください。

<strings.h>

strcasecmp 関数および strncasecmp 関数が含まれています。

<time.h>

<time.h> インクルード・ファイルは、時間関数および日付関数を宣言します。

asctime	ctime_r	gmtime64	localtime_r	strptime[1]
asctime_r	ctime64_r	gmtime_r	localtime64_r	time
clock	difftime	gmtime64_r	mktime	time64
ctime	difftime64	localtime	mktime64	
ctime64	gmtime	localtime64	strftime	

注: [1] コンパイル・コマンドで LOCALETYPE(*CLD) が指定されている場合、これらの関数は使用できません。

<time.h> インクルード・ファイルは、以下のものも提供します。

- カレンダー時間のコンポーネントを含んだ tm 構造体。 tm 構造体メンバーのリストについては、 167 ページの『gmtime() — 時間の変換』を参照してください。
- clock 関数で戻される値の秒ごとの数と等しい、CLOCKS_PER_SEC マクロ。
- clock_t 型、time_t 型、time64_t 型、および size_t 型。
- NULL ポインター値。

NULL および size_t 型について詳しくは、 15 ページの『<stddef.h>』を参照してください。

<wchar.h>

<wchar.h> ヘッダー・ファイルには、ワイド文字ストリングの操作に関連した宣言および定義が含まれています。 SYSIFCOPT(*IFSIO) が指定されている場合、ファイルを処理する関数はすべてアクセス可能になります。

btowc[1]	mbsrtowcs[1]	wcscmp	wcsrchr	wcswcs
fgetwc[2]	putwc[2]	wcscoll[1]	wcsrtombs[1]	wcswidth[1]
fgetws[2]	putwchar[2]	wcscpy	wcsspn	wcsxfrm[1]
fputwc[2]	swprintf[1]	wcscspn	wcsstr	wctob[1]
fputws[2]	swscanf[2]	wcsftime[1]	wcstod[1]	wcwidth[1]
fwide[2]	ungetwc[2]	__wcsicmp[1]	wcstod32[1]	wmemchr
fwprintf[2]	vfwprintf[2]	wcslen	wcstod64[1]	wmemcmp
fwscanf[2]	vswscanf[1]	wcsncat	wcstod128[1]	wmemcpy
getwc[2]	vswprintf[1]	wcsncmp	wcstok	wmemmove
getwchar[2]	vwprintf[2]	wcsncpy	wcstol[1]	wmemset
mbrlen[1]	wcrtomb[1]	__wcsnicmp[1]	wcstoll[1]	wprintf[2]
mbrtowc[1]	wcscat	wcspbrk	wcstoul[1]	wscanf[2]
mbsinit[1]	wcschr	wcsptime[3]	wcstoull[1]	

注: [1] コンパイル・コマンドで LOCALETYPE(*CLD) が指定されている場合、これらの関数は使用できません。
注: [2] コンパイル・コマンドで SYSIFCOPT(*IFSIO) および LOCALETYPE(*LOCALE) が指定されている場合に、これらの関数を使用できます。
注: [3] コンパイル・コマンドで LOCALETYPE(*LOCALEUTF) が指定されている場合に、これらの関数を使用できます。

<wchar.h> は、NULL マクロ、size_t 型および wchar_t 型も定義します。

NULL、size_t 型および wchar_t 型について詳しくは、 15 ページの『<stddef.h>』を参照してください。

<wctype.h>

<wctype.h> ヘッダー・ファイルは、次のワイド文字関数を宣言します。

iswalnum	iswgraph	iswspace	towlower	wctrans
iswalpha	iswlower	iswupper	towupper	
iswcntrl	iswprint	iswxdigit	towctrans	
iswdigit	iswpunct	iswctype	wctype	

<wctype.h> ヘッダー・ファイルには、ワイド文字分類の宣言および定義も含まれています。コンパイル・コマンドで LOCALETYPE(*CLD) が指定されている場合、これらの宣言および定義は使用できません。

<xxcvt.h>

<xxcvt.h> インクルード・ファイルには、QXXDTOP、QXXDTOZ、QXXITOP 、QXXITOZ、QXXPTOI、QXXPTOD、QXXZTOD、および QXXZTOI の各変換関数が使用する宣言が含まれています。

<xxdtaa.h>

<xxdtaa.h> インクルード・ファイルには、データ域インターフェース関数の QXXCHGDA と QXXRTVDA、および _DTAA_NAME_T 型の宣言が含まれています。

_DTAA_NAME_T の定義は、以下のとおりです。

```
typedef struct _DTAA_NAME_T {
        char  dtaa_name[10];
        char  dtaa_lib[10];
}_DTAA_NAME_T;
```

<xxenv.h>

<xxenv.h> インクルード・ファイルには、EPM 環境ハンドリング・プログラムの QPXXCALL および QPXXDLTE の宣言が含まれています。このインターフェースからは、ILE プロシージャーは呼び出せません。

_ENVPGM_T の定義は、以下のとおりです。

```
typedef struct _ENVPGM_T {
    char pgmname[10];
    char pgmlib[10];
} _ENVPGM_T;
```

<xxfdbk.h>

<xxfdbk.h> インクルード・ファイルには、i5/OS フィードバック域が使用する宣言が含まれています。フィードバック域から情報を取得する方法については、 296 ページの『_Riofbk() ― 入出力フィードバック情報の取得』および 305 ページの『_Ropnfbk() ― オープン・フィードバック情報の取得』を参照してください。

<xxfdbk.h> インクルード・ファイルで定義される型の例は、次のとおりです。

```
typedef _Packed struct _XXIOFB_T {
    short       file_dep_fb_offset;
    int         write_count;
    int         read_count;
    int         write_read_count;
    int         other_io_count;
    char        reserved1;
    char        cur_operation;
    char        rec_format[10];
    char        dev_class[2];
    char        dev_name[10];
    int         last_io_rec_len;
    char        reserved2[80];
    short       num_recs_retrieved;
    short       last_io_rec_len2;
    char        reserved3[2];
    int         cur_blk_count;
    char        reserved4[8];
} _XXIOFB_T;
```

オープン・フィードバック域について詳しくは、Information Center のカテゴリー『ファイルおよびファイル・システム』を参照してください。

マシン・インターフェース (MI) インクルード・ファイル

MI ヘッダー・ファイルの説明については、「*ILE C/C++ for AS/400 MI Library Reference*」を参照してください。

第 2 章 ライブラリー関数

この章では、標準的な C/C++ ライブラリー関数、およびライブラリー関数への ILE C/C++ 拡張機能 (ILE C/C++ MI 関数を除く) について説明します。 MI 関数については、「*ILE C/C++ for AS/400 MI Library Reference*」を参照してください。

このセクションにリストされている各ライブラリー関数の説明には、以下が含まれています。

- 関数を宣言するインクルード・ファイルを表示するフォーマットの記述。
- 関数によって戻されるデータ型。
- 関数の引数の必須データ型。

以下に log() 関数のフォーマットの例を示します。

```
#include <math.h>
double log(double x);
```

この例では以下のことが示されています。

- プログラムにファイル math.h を含める必要があります。
- log() 関数は double 型を戻します。
- log() 関数には、double 型の引数 *x* が必要です。

このセクションの例はライブラリー関数の使用法を概説するものであり、必ずしも完全なものではありません。

この章ではライブラリー関数をアルファベット順でリストしています。使用する関数が不明な場合は、『C/C++ ライブラリー』のライブラリー関数の要約を参照してください。

注: すべての関数は、特に注意書きがない限りスレッド・セーフとみなされます。

C/C++ ライブラリー

この章には、使用可能な C/C++ ライブラリー関数の要約、および各関数の本書内の位置がまとめられています。また、その関数の内容も簡単に説明されています。各ライブラリー関数は、実行する関数のタイプに従ってリストされています。

エラー処理

関数	ヘッダー・ファイル	ページ	説明
assert()	assert.h	46	診断メッセージを出力します。
atexit()	stdlib.h	48	プログラムの最後に実行する関数を登録します。
clearerr()	stdio.h	65	エラー標識をリセットします。
feof()	stdio.h	99	ストリーム入力のファイル終了標識をテストします。
ferror()	stdio.h	100	指定されたストリームのエラー標識をテストします。

© Copyright IBM Corp. 1999, 2010

関数	ヘッダー・ファイル	ページ	説明
_GetExcData()	signal.h	161	C シグナル・ハンドラー内からの例外についての情報を検索します。この関数は、コンパイル・コマンドに SYSIFCOPT(*SYNCSIGNAL) が指定されている場合には定義されません。
perror()	stdio.h	236	エラー・メッセージを stderr に出力します。
raise()	signal.h	267	シグナルを開始します。
signal()	signal.h	362	オペレーティング・システムからの割り込みシグナルの処理を許可します。
strerror()	string.h	384	システム・エラー・メッセージへのポインターを取得します。

検索およびソート

関数	ヘッダー・ファイル	ページ	説明
bsearch()	stdlib.h	54	ソートした列のバイナリー・サーチを実行します。
qsort()	stdlib.h	256	エレメントの配列に対してクイック・ソートを実行します。

数学関数

関数	ヘッダー・ファイル	ページ	説明
abs()	stdlib.h	39	整数の絶対値を計算します。
ceil()	math.h	64	ある数以上の整数のうちの、最小の整数を表す double 値を計算します。
div()	stdlib.h	90	整数の商および剰余を計算します。
erf()	math.h	91	誤差関数を計算します。
erfc()	math.h	91	大きな数の誤差関数を計算します。
exp()	math.h	93	指数関数を計算します。
fabs()	math.h	94	浮動小数点数の絶対値を計算します。
floor()	math.h	112	ある数以下の整数のうちの、最大の整数を表す double 値を計算します。
fmod()	math.h	113	ある引数を別の引数で除算した浮動小数点剰余を計算します。
frexp()	math.h	137	浮動小数点数を小数部と指数とに分離します。
gamma()	math.h	156	ガンマ関数を計算します。
hypot()	math.h	175	斜辺を計算します。
labs()	stdlib.h	184	long 型整数の絶対値を計算します。
llabs()	stdlib.h	184	long long 型整数の絶対値を計算します。
ldexp()	math.h	185	浮動小数点数に 2 の整数乗の乗算を行います。
ldiv()	stdlib.h	186	long 型整数の商および剰余を計算します。
lldiv()	stdlib.h	186	long long 型整数の商および剰余を計算します。
log()	math.h	198	自然対数を計算します。

関数	ヘッダー・ファイル	ページ	説明
log10()	math.h	199	基数を 10 として対数を計算します。
modf()	math.h	232	引数の符号付き小数部分を計算します。
nextafter()	math.h	232	表現可能な次の浮動小数点値を計算します。
nextafterl()	math.h	232	表現可能な次の浮動小数点値を計算します。
nexttoward()	math.h	232	表現可能な次の浮動小数点値を計算します。
nexttowardl()	math.h	232	表現可能な次の浮動小数点値を計算します。
pow()	math.h	237	乗数になる引数の値を計算します。
sqrt()	math.h	369	ある数値の平方根を計算します。

三角関数

関数	ヘッダー・ファイル	ページ	説明
acos()	math.h	40	逆余弦を計算します。
asin()	math.h	45	逆正弦を計算します。
atan()	math.h	47	逆正接を計算します。
atan2()	math.h	47	逆正接を計算します。
cos()	math.h	68	余弦を計算します。
cosh()	math.h	69	双曲線余弦を計算します。
sin()	math.h	365	正弦を計算します。
sinh()	math.h	366	双曲線正弦を計算します。
tan()	math.h	427	正接を計算します。
tanh()	math.h	428	双曲線正接を計算します。

ベッセル関数

関数	ヘッダー・ファイル	ページ	説明
j0()	math.h	53	第 1 種 0 次ベッセル関数。
j1()	math.h	53	第 1 種 1 次ベッセル関数。
jn()	math.h	53	第 1 種 *n* 次ベッセル関数。
y0()	math.h	53	第 2 種 0 次ベッセル関数。
y1()	math.h	53	第 2 種 1 次ベッセル関数。
yn()	math.h	53	第 2 種 *n* 次ベッセル関数。

時間処理

関数	ヘッダー・ファイル	ページ	説明
asctime()	time.h	41	構造体として格納された時間を、ストレージで文字ストリングに変換します。
asctime_r()	time.h	43	構造体として格納された時間を、ストレージで文字ストリングに変換します。 (asctime() の再始動可能な関数)
clock()	time.h	67	プロセッサー時間を判別します。

関数	ヘッダー・ファイル	ページ	説明
ctime()	time.h	75	long 値として格納された時間を文字ストリングに変換します。
ctime64()	time.h	77	long long 値として格納された時間を文字ストリングに変換します。
ctime_r()	time.h	78	long 値として格納された時間を文字ストリングに変換します。 (ctime() の再始動可能な関数)
ctime64_r()	time.h	80	long long 値として格納された時間を文字ストリングに変換します。 (ctime64() の再始動可能な関数)
difftime()	time.h	86	2 つの時刻の差を計算します。
difftime64()	time.h	88	2 つの時刻の差を計算します。
gmtime()	time.h	167	時刻を協定世界時構造に変換します。
gmtime_r()	time.h	171	時刻を協定世界時構造に変換します。 (gmtime() の再始動可能な関数)
gmtime64()	time.h	169	時刻を協定世界時構造に変換します。
gmtime64_r()	time.h	173	時刻を協定世界時構造に変換します。 (gmtime64() の再始動可能な関数)
localtime()	time.h	192	時刻を現地時間に変換します。
localtime64()	time.h	194	時刻を現地時間に変換します。
localtime_r()	time.h	195	時刻を現地時間に変換します。 (localtime() の再始動可能な関数)
localtime64_r()	time.h	197	時刻を現地時間に変換します。 (localtime64() の再始動可能な関数)
mktime()	time.h	228	現地時間をカレンダー時刻に変換します。
mktime64()	time.h	230	現地時間をカレンダー時刻に変換します。
time()	time.h	429	時間を秒数で戻します。
time64()	time.h	430	時間を秒数で戻します。

型変換

関数	ヘッダー・ファイル	ページ	説明
atof()	stdlib.h	49	文字ストリングを浮動小数点値に変換します。
atoi()	stdlib.h	50	文字ストリングを整数に変換します。
atol()	stdlib.h	52	文字ストリングを long 型整数に変換します。
atoll()	stdlib.h	52	文字ストリングを long 型整数に変換します。
_gcvt()	stdlib.h	157	浮動小数点値をストリングに変換します。
_itoa()	stdlib.h	183	整数をストリングに変換します。
_ltoa()	stdlib.h	200	long 型整数をストリングに変換します。
strtod()	stdlib.h	409	文字ストリングを倍精度の 2 進浮動小数点値に変換します。
strtod32()	stblib.h	412	文字ストリングを単精度の 10 進浮動小数点値に変換します。
strtod64()	stblib.h	412	文字ストリングを倍精度の 10 進浮動小数点値に変換します。

関数	ヘッダー・ファイル	ページ	説明
strtod128()	stblib.h	412	文字ストリングを 4 倍精度の 10 進浮動小数点値に変換します。
strtof()	stblib.h	409	文字ストリングを 2 進浮動小数点値に変換します。
strtol()	stdlib.h	418	文字ストリングを long 型整数に変換します。
strtold()	stdlib.h	409	文字ストリングを倍精度の 2 進浮動小数点値に変換します。
strtoll()	stdlib.h	418	文字ストリングを long long 型整数に変換します。
strtoul()	stdlib.h	420	文字ストリングを符号なし long 型整数に変換します。
strtoull()	stdlib.h	420	ストリングを符号なし long long 型整数に変換します。
toascii()	ctype.h	433	文字を対応する ASCII 値に変換します。
_ultoa()	stdlib.h	438	符号なし long 型整数をストリングに変換します。
wcstod()	wchar.h	498	ワイド文字ストリングを倍精度の 2 進浮動小数点値に変換します。
wcstod32()	wchar.h	499	ワイド文字ストリングを単精度の 10 進浮動小数点値に変換します。
wcstod64()	wchar.h	499	ワイド文字ストリングを倍精度の 10 進浮動小数点値に変換します。
wcstod128()	wchar.h	499	ワイド文字ストリングを 4 倍精度の 10 進浮動小数点値に変換します。
wcstol()	wchar.h	503	ワイド文字ストリングを long 型整数に変換します。
wcstoll()	wchar.h	503	ワイド文字ストリングを long long 型整数に変換します。
wcstoul()	wchar.h	508	ワイド文字ストリングを符号なし long 型整数に変換します。
wcstoull()	wchar.h	508	ワイド文字ストリングを符号なし long long 型整数に変換します。

変換

関数	ヘッダー・ファイル	ページ	説明
QXXDTOP()	xxcvt.h	259	浮動小数点値をパック 10 進値に変換します。
QXXDTOZ()	xxcvt.h	260	浮動小数点値をゾーン 10 進値に変換します。
QXXITOP()	xxcvt.h	261	整数値をパック 10 進値に変換します。
QXXITOZ()	xxcvt.h	262	整数値をゾーン 10 進値に変換します。
QXXPTOD()	xxcvt.h	262	パック 10 進値を浮動小数点値に変換します。
QXXPTOI()	xxcvt.h	263	パック 10 進値を整数値に変換します。
QXXZTOD()	xxcvt.h	265	ゾーン 10 進値を浮動小数点値に変換します。
QXXZTOI()	xxcvt.h	266	ゾーン 10 進値を整数値に変換します。

レコード入出力

関数	ヘッダー・ファイル	ページ	説明
_Racquire()	recio.h	269	レコード入出力操作用のデバイスを準備します。
_Rclose()	recio.h	270	レコード入出力操作用にオープンしたファイルをクローズします。
_Rcommit()	recio.h	271	現行トランザクションを完了し、新規コミットメント境界を確立します。
_Rdelete()	recio.h	273	現在ロックされているレコードを削除します。
_Rdevatr()	recio.h xxfdbk.h	275	fp およびデバイス pgmdev に参照されるファイルの、デバイス属性フィードバック域のコピーへのポインターを戻します。
_Rfeod()	recio.h	290	fp によるファイル参照に対して、強制的にファイル終了状態にします。
_Rfeov()	recio.h	291	テープに対して強制的にボリューム終了状態にします。
_Rformat()	recio.h	292	fp によるファイル参照に対してレコード・フォーマットを fmt に設定します。
_Rindara()	recio.h	294	後続のレコード入出力操作に使用する、別の標識領域を設定します。
_Riofbk()	recio.h xxfdbk.h	296	fp によって参照されるファイルの、入出力フィードバック域のコピーへのポインターを戻します。
_Rlocate()	recio.h	298	fp と関連付けられ、key, klen_rrn および opt パラメーターによって指定されたファイル内のレコードに移動します。
_Ropen()	recio.h	301	レコード入出力操作用のファイルをオープンします。
_Ropnfbk()	recio.h xxfdbk.h	305	fp により参照されるファイルのオープン・フィードバック域のコピーへのポインターを戻します。
_Rpgmdev()	recio.h	306	デフォルトのプログラム装置を設定します。
_Rreadd()	recio.h	308	相対レコード番号でレコードを読み取ります。
_Rreadf()	recio.h	310	最初のレコードを読み取ります。
_Rreadindv()	recio.h	312	送信勧誘された装置からデータを読み取ります。
_Rreadk()	recio.h	314	key によりレコードを読み取ります。
_Rreadl()	recio.h	318	最終レコードを読み取ります。
_Rreadn()	recio.h	319	次のレコードを読み取ります。
_Rreadnc()	recio.h	322	サブファイル内にある、次の変更済みレコードを読み取ります。
_Rreadp()	recio.h	324	前のレコードを読み取ります。
_Rreads()	recio.h	326	同じレコードを読み取ります。
_Rrelease()	recio.h	328	指定したデバイスにおいて、レコード入出力操作を行えなくします。
_Rrlslck()	recio.h	329	現在ロックされているレコードを解放します。
_Rrollbck()	recio.h	331	最新のコミットメント境界を、現行のコミットメント境界として再確立します。

関数	ヘッダー・ファイル	ページ	説明
_Rupdate()	recio.h	332	現在ロックされているレコードに、更新用の書き込みを行います。
_Rupfb()	recio.h	334	最新のレコード入出力操作に関する情報を使って、フィードバック構造体を更新します。
_Rwrite()	recio.h	336	ファイルの終わりにレコードを書き込みます。
_Rwrited()	recio.h	338	相対レコード番号でレコードを書き込みます。上書きの対象は、削除されたレコードのみです。
_Rwriterd()	recio.h	341	レコードの書き込みおよび読み取りを行います。
_Rwrread()	recio.h	342	入出力データに対して別のバッファーを指定できることを除き、_Rwriterd() として機能します。

ストリーム入出力

フォーマット済み入出力

関数	ヘッダー・ファイル	ページ	説明
fprintf()	stdio.h	122	文字のフォーマット設定を行い、出力ストリームに出力します。
fscanf()	stdio.h	138	データをストリームから引数により指定されたロケーションへ読み取ります。
fwprintf()	stdio.h	149	ワイド文字としてデータのフォーマット設定を行い、ストリームに書き込みます。
fwscanf()	stdio.h	153	ワイド・データを、ストリームから引数により指定されたロケーションへ読み取ります。
printf()	stdio.h	238	文字のフォーマット設定を行い、stdout に出力します。
scanf()	stdio.h	344	データを stdin から引数により指定されたロケーションに読み取ります。
snprintf()	stdio.h	367	snprintf() 関数はバッファーに **n** 文字書き込まれた後に停止することを除き、sprintf と同じです。
sprintf()	stdio.h	368	文字のフォーマット設定を行い、バッファーに書き込みます。
sscanf()	stdio.h	371	データをバッファーから引数により指定されたロケーションに読み取ります。
swprintf()	wchar.h	423	ワイド文字のフォーマット設定を行い、バッファーに書き込みます。
swscanf()	wchar.h	425	ワイド・データをバッファーから引数により指定されたロケーションに読み取ります。
vfprintf()	stdio.h stdarg.h	444	文字のフォーマット設定を行い、引数の可変値を使用して出力 stream に出力します。
vfscanf()	stdarg.h stdio.h	446	指定されたストリームから、引数の可変値により指定されたロケーションに、データを読み取ります。

関数	ヘッダー・ファイル	ページ	説明
vfwprintf()	stdio.h stdarg.h	447	引数データをワイド文字としてフォーマット設定し、引数の可変値を使用してストリームに書き込みます。
vfwscanf()	stdarg.h stdio.h	449	指定されたストリームから、引数の可変値により指定されたロケーションに、ワイド・データを読み取ります。
vprintf()	stdarg.h stdio.h	452	文字のフォーマット設定を行い、引数の可変値を使用して stdout に書き込みます。
vscanf()	stdarg.h stdio.h	453	データを stdin から引数の可変値により指定されたロケーションに読み取ります。
vsnprintf()	stdio.h stdarg.h	455	vsnprintf 関数はバッファーに **n** 文字書き込まれた後に停止することを除き、vsprintf と同じです。
vsprintf()	stdarg.h stdio.h	456	文字のフォーマット設定を行い、引数の可変値を使用してバッファーに書き込みます。
vsscanf()	stdarg.h stdio.h	458	バッファーから、引数の可変値により指定されたロケーションに、データを書き込みます。
vswprintf()	wchar.h stdarg.h	459	ワイド文字のフォーマット設定を行い、引数の可変値を使用してバッファーに書き込みます。
vswscanf()	stdarg.h wchar.h	461	バッファーから、引数の可変値により指定されたロケーションに、ワイド・データを▸読み取ります。
vwprintf()	wchar.h stdarg.h	463	ワイド文字のフォーマット設定を行い、引数の可変値を使用して stdout に書き込みます。
vwscanf()	stdarg.h stdio.h	465	ワイド・データを stdin から引数の可変値により指定されたロケーションに読み取ります。
wprintf()	stdio.h	527	ワイド文字のフォーマット設定を行い、stdout に書き込みます。
wscanf()	stdio.h	528	ワイド・データを stdin から引数により指定されたロケーションに読み取ります。

文字およびストリングの入出力

関数	ヘッダー・ファイル	ページ	説明
fgetc()	stdio.h	102	指定された入力ストリームから文字を読み取ります。
fgets()	stdio.h	105	指定された入力ストリームからストリングを読み取ります。
fgetwc()	stdio.h	107	指定されたストリームからワイド文字を読み取ります。
fgetws()	stdio.h	109	指定されたストリームからワイド文字ストリングを読み取ります。
fputc()	stdio.h	124	指定された出力ストリームに文字を出力します。

関数	ヘッダー・ファイル	ページ	説明
_fputchar()	stdio.h	126	STDOUT に文字を書き込みます。
fputs()	stdio.h	127	指定された出力ストリームにストリングを出力します。
fputwc()	stdio.h	128	指定されたストリームにワイド文字を書き込みます。
fputws()	stdio.h	130	指定されたストリームにワイド文字ストリングを書き込みます。
getc()	stdio.h	158	指定された入力ストリームから文字を読み取ります。
getchar()	stdio.h	158	stdin から文字を読み取ります。
gets()	stdio.h	162	stdin から行を読み取ります。
getwc()	stdio.h	163	指定されたストリームからワイド文字を読み取ります。
getwchar()	stdio.h	165	stdin からワイド文字を取得します。
putc()	stdio.h	249	指定された出力ストリームに文字を出力します。
putchar()	stdio.h	249	文字を stdout に出力します。
puts()	stdio.h	251	ストリングを stdout に出力します。
putwc()	stdio.h	252	指定されたストリームにワイド文字を書き込みます。
putwchar()	stdio.h	254	ワイド文字を stdout に書き込みます。
ungetc()	stdio.h	439	文字を指定された入力ストリームにプッシュして戻します。
ungetwc()	stdio.h	440	ワイド文字を指定された入力ストリームにプッシュして戻します。

直接入出力

関数	ヘッダー・ファイル	ページ	説明
fread()	stdio.h	132	指定された入力ストリームから複数の項目を読み取ります。
fwrite()	stdio.h	152	指定された出力ストリームに複数の項目を書き込みます。

ファイルの位置決め

関数	ヘッダー・ファイル	ページ	説明
fgetpos()	stdio.h	104	ファイル・ポインターの現在位置を取得します。
fseek()	stdio.h	140	ファイル・ポインターを新規位置に移動します。
fseeko()	stdio.h	140	fseek() と同じ。
fsetpos()	stdio.h	142	ファイル・ポインターを新規位置に移動します。
ftell()	stdio.h	144	ファイル・ポインターの現在位置を取得します。
ftello()	stdio.h	144	ftell() と同じ。

関数	ヘッダー・ファイル	ページ	説明
rewind()	stdio.h	288	ファイル・ポインターをファイルの先頭位置に位置変更します。

ファイル・アクセス

関数	ヘッダー・ファイル	ページ	説明
fclose()	stdio.h	95	指定されたストリームをクローズします。
fdopen()	stdio.h	96	入力ストリームまたは出力ストリームをファイルと関連付けます。
fflush()	stdio.h	101	システムに要求して、バッファーの内容をファイルに書き込ませます。
fopen()	stdio.h	114	指定されたストリームをオープンします。
freopen()	stdio.h	136	ファイルをクローズし、ストリームを再割り当てします。
fwide()	stdio.h	146	ストリームの指向を設定します。
setbuf()	stdio.h	351	バッファリングの制御を許可します。
setvbuf()	stdio.h	360	指定されたストリームのバッファリングとバッファー・サイズを制御します。
wfopen()	stdio.h	521	指定されたストリームをオープンし、ファイル名およびモードをワイド文字として受け入れます。

ファイル操作

関数	ヘッダー・ファイル	ページ	説明
fileno()	stdio.h	111	ファイル・ハンドルを判別します。
remove()	stdio.h	286	指定されたファイルを削除します。
rename()	stdio.h	287	指定されたファイルの名前を変更します。
tmpfile()	stdio.h	432	一時ファイルを作成し、そのファイルへのポインターを戻します。
tmpnam()	stdio.h	433	一時ファイル名を作成します。

引数リストの処理

関数	ヘッダー・ファイル	ページ	説明
va_arg()	stdarg.h	442	複数の関数引数へのアクセスを許可します。
va_end()	stdarg.h	442	複数の関数引数へのアクセスを許可します。
va_start()	stdarg.h	442	複数の関数引数へのアクセスを許可します。

疑似乱数

関数	ヘッダー・ファイル	ページ	説明
rand(), rand_r()	stdlib.h	268	疑似乱数整数を戻します。(rand_r() は、rand() の再始動可能な関数です。)

関数	ヘッダー・ファイル	ページ	説明
srand()	stdlib.h	370	疑似乱数の開始点を設定します。

動的メモリー管理

関数	ヘッダー・ファイル	ページ	説明
calloc()	stdlib.h	58	列のストレージ・スペースを予約し、すべてのエレメントの値を 0 に初期設定します。
_C_Quickpool_Debug()	stdlib.h	69	高速プール・メモリー・マネージャー特性を変更します。
_C_Quickpool_Init()	stdlib.h	71	高速プール・メモリー・マネージャー・アルゴリズムの使用法を初期設定します。
_C_Quickpool_Report()	stdlib.h	73	現行の活動化グループ内で高速プール・メモリー・マネージャー・アルゴリズムにより使用されるメモリーのスナップショットを含む、スプール・ファイルを生成します。
_C_TS_malloc_debug()	mallocinfo.h	82	_C_TS_malloc_info と同じ情報を戻します。らに、テラスペースでコンパイルされるときに malloc 関数により使用されるメモリー構造体についての詳細情報のスプール・ファイルを作成します。
_C_TS_malloc_info()	mallocinfo.h	84	現在のメモリー使用量情報を戻します。
free()	stdlib.h	134	ストレージ・ブロックを解放します。
malloc()	stdlib.h	203	ストレージ・ブロックを予約します。
realloc()	stdlib.h	276	オブジェクトに対して割り振られるストレージ・サイズを変更します。

メモリー・オブジェクト

関数	ヘッダー・ファイル	ページ	説明
memchr()	string.h	221	指定した文字の最初のオカレンスのバッファーを検索します。
memcmp()	string.h	222	2 つのバッファーを比較します。
memcpy()	string.h	223	バッファーをコピーします。
memicmp()	string.h	224	2 つのバッファーを大/小文字に関係なく比較します。
memmove()	string.h	226	バッファーを移動します。
memset()	string.h	227	バッファーを指定した値に設定します。
wmemchr()	wchar.h	521	ワイド文字バッファー内でワイド文字を位置指定します。
wmemcmp()	wchar.h	522	2 つのワイド文字バッファーを比較します。
wmemcpy()	wchar.h	524	ワイド文字バッファーをコピーします。
wmemmove()	wchar.h	525	ワイド文字バッファーを移動します。
wmemset()	wchar.h	526	ワイド文字バッファーを指定した値に設定します。

環境の対話

関数	ヘッダー・ファイル	ページ	説明
abort()	stdlib.h	38	プログラムを異常終了します。
_C_Get_Ssn_Handle()	stdio.h	57	DSM API で使用するために、C セッションにハンドルを戻します。
exit()	stdlib.h	92	初期スレッドで呼び出された場合は、プログラムを正常に終了します。
getenv()	stdlib.h	160	指定された変数の環境変数を検索します。
localeconv()	locale.h	187	現行のロケールに従って、struct lconv の数量をフォーマット設定します。
longjmp()	setjmp.h	201	スタック環境を復元します。
nl_langinfo()	langinfo.h	234	現行ロケールから情報を取り出します。
putenv()	stdlib.h	250	既存の変数を変更するか、新しい変数を作成することにより、環境変数の値を設定します。
setjmp()	setjmp.h	352	スタック環境を保存します。
setlocale()	locale.h	354	ロケールを変更するか、照会します。
system()	stdlib.h	426	オペレーティング・システムのコマンド・インタープリターに、ストリングを渡します。
wcslocaleconv()	locale.h	483	現行ロケールに従って struct wcslconv 内の数量のフォーマット設定を行います。

ストリング操作

関数	ヘッダー・ファイル	ページ	説明
strcasecmp()	strings.h	373	大/小文字を区別しないで、ストリングを比較します。
strcat()	string.h	374	2 つのストリングを連結します。
strchr()	string.h	375	ストリング中の指定された文字が最初に現れる位置を位置指定します。
strcmp()	string.h	376	2 つのストリングの値を比較します。
strcmpi()	string.h	378	2 つのストリングの値を、大/小文字に関係なく比較します。
strcoll()	string.h	379	2 つのストリングのロケール定義値を比較します。
strcpy()	string.h	380	あるストリングを別のストリングにコピーします。
strcspn()	string.h	381	2 番目のストリングに含まれない、最初のサブストリングの長さを文字列の長さとして検索します。
strdup()	string.h	383	ストリングを複写します。
strfmon()	string.h	384	通貨の値をストリングに変換します。
strftime()	time.h	387	日時を定様式ストリングに変換します。
stricmp()	string.h	390	2 つのストリングの値を、大/小文字に関係なく比較します。
strlen()	string.h	392	ストリングの長さを計算します。

関数	ヘッダー・ファイル	ページ	説明
strncasecmp()	strings.h	393	大/小文字を区別しないで、ストリングを比較します。
strncat()	string.h	394	ストリング内の指定された長さ分を、別のストリングに追加します。
strncmp()	string.h	395	2 つのストリングを、指定した長さまで比較します。
strncpy()	string.h	397	ストリング内の指定された長さ分を、別のストリングにコピーします。
strnicmp()	string.h	398	2 つのサブストリングの値を、大/小文字に関係なく比較します。
strnset()	string.h	400	ストリングの文字を設定します。
strpbrk()	string.h	401	ストリング内の指定された文字を位置指定します。
strptime()	time.h	402	ストリングをフォーマット済みの時刻に変換します。
strrchr()	string.h	406	ストリング内の文字が最後に現れる位置を検索します。
strspn()	string.h	407	指定された文字セットの一部ではない、ストリング内の最初の文字を位置指定します。
strstr()	string.h	408	別のストリング内の、ストリングが最初に現れる位置を位置指定します。
strtok()	string.h	415	ストリング内の指定されたトークンを位置指定します。
strtok_r()	string.h	417	ストリング内の指定されたトークンを位置指定します。(strtok() の再始動可能な関数)。
strxfrm()	string.h	422	ロケールに応じてストリングを変換します。
wcsftime()	wchar.h	479	フォーマット済みの日時に変換します。
wcsptime()	wchar.h	491	ストリングをフォーマット済みの時刻に変換します。
wcsstr()	wchar.h	497	ワイド文字のサブストリングを位置指定します。
wcstok()	wchar.h	502	ワイド文字ストリングをトークン化します。

文字テスト

関数	ヘッダー・ファイル	ページ	説明
isalnum()	ctype.h	176	英数字をテストします。
isalpha()	ctype.h	176	英字をテストします。
isascii()	ctype.h	177	ASCII 値をテストします。
isblank()	ctype.h	179	ブランク文字やタブ文字をテストします。
iscntrl()	ctype.h	176	制御文字をテストします。
isdigit()	ctype.h	176	10 進数字をテストします。
isgraph()	ctype.h	176	スペースを除く出力可能文字をテストします。
islower()	ctype.h	176	小文字をテストします。
isprint()	ctype.h	176	スペースを含む出力可能文字をテストします。

関数	ヘッダー・ファイル	ページ	説明
ispunct()	ctype.h	176	ロケールで定義されている句読文字をテストします。
isspace()	ctype.h	176	空白文字をテストします。
isupper()	ctype.h	176	大文字をテストします。
isxdigit()	ctype.h	176	ワイド 16 進数字の 0 から 9、a から f、または A から F をテストします。

マルチバイト文字のテスト

関数	ヘッダー・ファイル	ページ	説明
iswalnum()	wctype.h	180	ワイド英数字をテストします。
iswalpha()	wctype.h	180	ワイド英字をテストします。
iswcntrl()	wctype.h	180	ワイド制御文字をテストします。
iswctype()	wctype.h	182	文字特性をテストします。
iswdigit()	wctype.h	180	ワイド 10 進数字をテストします。
iswgraph()	wctype.h	180	スペースを除く印字文字をテストします。
iswlower()	wctype.h	180	ワイド小文字をテストします。
iswprint()	wctype.h	180	ワイド印字文字をテストします。
iswpunct()	wctype.h	180	ロケールで定義されているワイド句読文字をテストします。
iswspace()	wctype.h	180	ワイド空白文字をテストします。
iswupper()	wctype.h	180	ワイド大文字をテストします。
iswxdigit()	wctype.h	180	ワイド 16 進数字の 0 から 9、a から f、または A から F をテストします。

文字ケースのマッピング

関数	ヘッダー・ファイル	ページ	説明
tolower()	ctype.h	434	文字を小文字に変換します。
toupper()	ctype.h	434	文字を大文字に変換します。
towlower()	ctype.h	436	ワイド文字を小文字に変換します。
towupper()	ctype.h	436	ワイド文字を大文字に変換します。

マルチバイト文字の操作

関数	ヘッダー・ファイル	ページ	説明
btowc()	stdio.h wchar.h	56	1 バイト文字をワイド文字に変換します。
mblen()	stdlib.h	205	マルチバイト文字の長さを判別します。
mbrlen()	stdlib.h	207	マルチバイト文字の長さを判別します。 (mblen() の再始動可能な関数)

関数	ヘッダー・ファイル	ページ	説明
mbrtowc()	stdlib.h	210	マルチバイト文字をワイド文字に変換します。(mbtowc() の再始動可能な関数)
mbsinit()	stdlib.h	213	状態オブジェクトが初期状態であるかどうかをテストします。
mbsrtowcs()	stdlib.h	214	マルチバイト・ストリングをワイド文字ストリングに変換します。 (mbstowcs() の再始動可能な関数)
mbstowcs()	stdlib.h	216	マルチバイト・ストリングをワイド文字ストリングに変換します。
mbtowc()	stdlib.h	220	マルチバイト文字をワイド文字に変換します。
towctrans()	wctype.h	435	ワイド文字を変換します。
wcrtomb()	stdlib.h	467	ワイド文字をマルチバイト文字に変換します。(wctomb() の再始動可能な関数)。
wcscat()	wchar.h	471	ワイド文字ストリングを連結します。
wcschr()	wchar.h	472	ワイド文字ストリングの中でワイド文字を検索します。
wcscmp()	wchar.h	474	2 つのワイド文字ストリングを比較します。
wcscoll()	wchar.h	475	2 つのワイド文字ストリングのロケール定義値を比較します。
wcscpy()	wchar.h	476	ワイド文字ストリングをコピーします。
wcscspn()	wchar.h	477	ワイド文字ストリングの中で複数の文字を検索します。
__wcsicmp()	wchar.h	480	2 つのワイド文字ストリングを、大/小文字に関係なく比較します。
wcslen()	wchar.h	482	ワイド文字ストリングの長さを検索します。
wcsncat()	wchar.h	484	ワイド文字ストリング・セグメントを連結します。
wcsncmp()	wchar.h	485	ワイド文字ストリング・セグメントを比較します。
wcsncpy()	wchar.h	487	ワイド文字ストリング・セグメントをコピーします。
__wcsnicmp()	wchar.h	488	2 つのワイド文字サブストリングを、大/小文字に関係なく比較します。
wcspbrk()	wchar.h	490	ストリング内のワイド文字を位置指定します。
wcsrchr()	wchar.h	493	ストリング内のワイド文字を位置指定します。
wcsrtombs()	stdlib.h	494	ワイド文字ストリングをマルチバイト文字ストリングに変換します。 (wcstombs() の再始動可能な関数)。
wcsspn()	wchar.h	496	一致していない最初のワイド文字のオフセットを検索します。
wcstombs()	stdlib.h	505	ワイド文字ストリングをマルチバイト文字ストリングに変換します。
wcswcs()	wchar.h	510	別のワイド文字ストリング内のワイド文字ストリングを位置指定します。
wcswidth()	wchar.h	511	ワイド文字ストリングの表示幅を判別します。
wcsxfrm()	wchar.h	512	ロケールに応じてワイド文字ストリングを変換します。

関数	ヘッダー・ファイル	ページ	説明
wctob()	stdlib.h	513	ワイド文字を 1 バイト文字に変換します。
wctomb()	stdlib.h	515	ワイド文字を複数のマルチバイト文字に変換します。
wctrans()	wctype.h	516	文字マッピングのハンドルを取得します。
wctype()	wchar.h	517	文字特性分類のハンドルを取得します。
wcwidth()	wchar.h	520	ワイド文字の表示幅を判別します。

データ域

関数	ヘッダー・ファイル	ページ	説明
QXXCHGDA()	xxdtaa.h	258	データ域を変更します。
QXXRTVDA()	xxdtaa.h	264	dtaname により指定されたデータ域のコピーを取り出します。

メッセージ・カタログ

関数	ヘッダー・ファイル	ページ	説明
catclose()	nl_types.h	60	メッセージ・カタログをクローズします。
catgets()	nl_types.h	61	オープンしたメッセージ・カタログからメッセージを読み取ります。
catopen()	nl_types.h	63	メッセージ・カタログをオープンします。

正規表現

関数	ヘッダー・ファイル	ページ	説明
regcomp()	regex.h	279	正規表現をコンパイルします。
regerror()	regex.h	281	正規表現のエラー・メッセージを戻します。
regexec()	regex.h	283	コンパイル済み正規表現を実行します。
regfree()	regex.h	285	正規表現用にメモリーを解放します。

abort() — プログラムの停止

フォーマット

```
#include <stdlib.h>
void abort(void);
```

言語レベル: ANSI

スレッド・セーフ: はい。

説明

abort() 関数はプログラムを異常終了させ、ホスト環境に制御を戻します。 exit() 関数と同様、abort() 関数はプログラムを終了する前にバッファーを削除し、オープン・ファイルをクローズします。

abort() 関数を呼び出すと SIGABRT 信号が発信されます。SIGABRT がシグナル・ハンドラーによってキャッチされた場合は abort() 関数によるプログラムの終了は起こらず、シグナル・ハンドラーは戻りません。

注: SYSIFCOPT(*ASYNCSIGNAL) でコンパイルされた場合、abort() 関数はシグナル・ハンドラー内で呼び出すことはできません。

戻り値

戻り値はありません。

abort() の使用例

次の例は、ファイル myfile が正常にオープンするかどうかをテストします。エラーが起こると、エラー・メッセージが出力され、プログラムは abort() 関数の呼び出しにより終了します。

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
   FILE *stream;

   if ((stream = fopen("mylib/myfile", "r")) == NULL)
   {
      perror("Could not open data file");
      abort();
   }
}
```

関連情報

- 92 ページの『exit() — プログラムの終了』
- 362 ページの『signal() — 割り込みシグナルの処理』
- 18 ページの『<stdlib.h>』
- i5/OS Information Center の『API』トピック内にある signal() API を参照してください。

abs() — 整数の絶対値の計算

フォーマット

```
#include <stdlib.h>
int abs(int n);
```

言語レベル: ANSI

スレッド・セーフ: はい。

説明

abs() 関数は整数の引数 *n* の絶対値を戻します。

戻り値

エラーの戻り値はありません。引数の絶対値が整数で表すことができない場合、結果は予期できません。最小許容整数の値は、<limits.h> インクルード・ファイル内の INT_MIN により定義されます。

abs() の使用例

次の例は整数 x の絶対値を計算し、それを y に割り当てます。

```
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
    int x = -4, y;

    y = abs(x);

    printf("The absolute value of x is %d.¥n", y);

    /********************* Output **************************
            The absolute value of x is 4.
    *****************************************************/
}
```

関連情報

- 94 ページの『fabs() ― 浮動小数点絶対値の計算』
- 184 ページの『labs() ― llabs() ― long 型および long long 型整数の絶対値の計算』
- 8 ページの『<limits.h>』
- 18 ページの『<stdlib.h>』

acos() ― 逆余弦の計算

フォーマット

```
#include <math.h>
double acos(double x);
```

言語レベル: ANSI

スレッド・セーフ: はい。

説明

acos() 関数は *x* の逆余弦 (ラジアン表記) を、0 から Π の範囲で計算します。

戻り値

acos() 関数は *x* の逆余弦を戻します。*x* の値は -1 と 1 の間 (両端を含む) でなければなりません。*x* が -1 よりも小さい、または 1 よりも大きい場合、acos() は errno に EDOM を設定し、0 を戻します。

acos() の使用例

次の例は *x* の値を要求するプロンプトを出します。*x* が 1 よりも大きいか -1 よりも小さい場合はエラー・メッセージを出力します。それ以外の場合は *x* の逆余弦を *y* に割り当てます。

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define MAX  1.0
#define MIN -1.0

int main(void)
```

```
{
  double x, y;

  printf( "Enter x¥n" );
  scanf( "%lf", &x );

  /* Output error if not in range */
  if ( x > MAX )
    printf( "Error: %lf too large for acos¥n", x );
  else if ( x < MIN )
    printf( "Error: %lf too small for acos¥n", x );
  else {
    y = acos( x );
    printf( "acos( %lf ) = %lf¥n", x, y );
  }
}

/*******  Expected output if 0.4 is entered:  *********

Enter x
acos( 0.400000 ) = 1.159279
*/
```

関連情報

- 45 ページの『asin() ― 逆正弦の計算』
- 47 ページの『atan() - atan2() ― 逆正接の計算』
- 68 ページの『cos() ― 余弦の計算』
- 69 ページの『cosh() ― 双曲線余弦の計算』
- 365 ページの『sin() ― 正弦の計算』
- 366 ページの『sinh() ― 双曲線正弦の計算』
- 427 ページの『tan() ― 正接の計算』
- 428 ページの『tanh() ― 双曲線正接の計算』
- 9 ページの『<math.h>』

asctime() ― 時間から文字ストリングへの変換

フォーマット

```
#include <time.h>
char *asctime(const struct tm *time);
```

言語レベル: ANSI

スレッド・セーフ: いいえ。代わりに asctime_r() を使用します。

説明

asctime() 関数は、*time* により指される構造体として格納された時間を、文字ストリングに変換します。*time* 値は、gmtime() 関数、gmtime64() 関数、localtime() 関数、または localtime64() 関数の呼び出しから取得できます。

asctime() が作成するストリングの結果にはちょうど 26 文字が含まれ、以下の形式となっています。

```
"%.3s %.3s%3d %.2d:%.2d:%.2d %d\n"
```

戻されるストリングの例を以下に示します。

```
   Sat Jul 16 02:03:55 1994\n\0
```
または
```
   Sat Jul 16  2:03:55 1994\n\0
```

asctime() 関数は 24 時間クロック形式を使用します。曜日は、Sun、Mon、Tue、Wed、Thu、Fri、および Sat に省略されます。月は、Jan、Feb、Mar、Apr、May、Jun、Jul、Aug、Sep、Oct、Nov、および Dec に省略されます。すべてのフィールドには固定幅があります。一桁しかない日付は、ゼロまたはブランク・スペースがその前に置かれます。改行文字 (¥n) および NULL 文字 (¥0) がストリングの最後の位置を占めます。

日時の関数は、1970 年 1 月 1 日 00:00:00 世界時から始まります。

戻り値

asctime() 関数は結果として生じた文字ストリングを指すポインターを戻します。関数が正常に実行されなかった場合、NULL を戻します。

注: asctime() 関数と ctime() 関数、および他の時間関数は、戻りストリングの保持用に静的に割り振られた共通のバッファーを使用できます。これらの関数の 1 つを呼び出すごとに、以前の呼び出しの結果が破棄される可能性があります。asctime_r()、ctime_r()、gmtime_r()、および localtime_r() 関数は、戻りストリングの保持用に静的に割り振られた共通のバッファーを使用しません。再入可能性を希望する場合は、これらの関数を asctime() 関数、ctime() 関数、gmtime() 関数、および localtime() 関数の代わりに使用することができます。

asctime() の使用例

次の例はシステム・クロックをポーリングし、現在時刻を示すメッセージを出力します。

```
#include <time.h>
#include <stdio.h>

int main(void)
{
    struct tm *newtime;
    time_t ltime;

/* Get the time in seconds */
    time(&ltime);
/* Convert it to the structure tm */
    newtime = localtime(&ltime);

        /* Print the local time as a string */
    printf("The current date and time are %s",
             asctime(newtime));
}

/****************  Output should be similar to:  ******************
The current date and time are Fri Sep 16 13:29:51 1994
*/
```

関連情報

- 43 ページの『asctime_r() — 時間から文字ストリングへの変換 (再始動可能)』
- 75 ページの『ctime() — 時間から文字ストリングへの変換』
- 77 ページの『ctime64() — 時間から文字ストリングへの変換』
- 80 ページの『ctime64_r() — 時間から文字ストリングへの変換 (再始動可能)』
- 78 ページの『ctime_r() — 時間から文字ストリングへの変換 (再始動可能)』

- 167 ページの『gmtime() — 時間の変換』
- 169 ページの『gmtime64() — 時間の変換』
- 173 ページの『gmtime64_r() — 時間の変換 (再始動可能)』
- 171 ページの『gmtime_r() — 時間の変換 (再始動可能)』
- 192 ページの『localtime() — 時間の変換』
- 194 ページの『localtime64() — 時間の変換』
- 197 ページの『localtime64_r() — 時間の変換 (再始動可能)』
- 195 ページの『localtime_r() — 時間の変換 (再始動可能)』
- 228 ページの『mktime() — 地方時の変換』
- 230 ページの『mktime64() — 地方時の変換』
- 387 ページの『strftime() — 日付/時刻からストリングへの変換』
- 429 ページの『time() — 現在時刻の判別』
- 238 ページの『printf() — 定様式の文字の出力』
- 354 ページの『setlocale() — ロケールの設定』
- 430 ページの『time64() — 現在時刻の判別』
- 19 ページの『<time.h>』

asctime_r() — 時間から文字ストリングへの変換 (再始動可能)

フォーマット

```
#include <time.h>
char *asctime_r(const struct tm *tm, char *buf);
```

言語レベル: XPG4

スレッド・セーフ: はい。

説明

この関数は asctime() 関数の再始動可能バージョンです。

asctime_r() 関数は、*tm* により指される構造体として格納された時間を、文字ストリングに変換します。*tm* 値は、gmtime_r()、gmtime64_r()、localtime_r()、または localtime64_r() の呼び出しから取得できます。

asctime_r() が作成するストリングの結果にはちょうど 26 文字が含まれ、以下の形式となっています。

```
"%.3s %.3s%3d %.2d:%.2d:%.2d %d¥n"
```

戻されるストリングの例を以下に示します。

```
   Sat Jul 16 02:03:55 1994\n\0
または
   Sat Jul 16  2:03:55 1994\n\0
```

asctime_r() 関数は 24 時間クロック形式を使用します。曜日は、Sun、Mon、Tue、Wed、Thu、Fri、および Sat に省略されます。月は、Jan、Feb、Mar、Apr、May、Jun、Jul、Aug、Sep、Oct、Nov、および Dec に省略されます。すべてのフィールドには固定幅があります。一桁しかない日付は、ゼロまたはブランク・スペースがその前に置かれます。改行文字 (¥n) および NULL 文字 (¥0) がストリングの最後の位置を占めます。

日時の関数は、1970 年 1 月 1 日 00:00:00 世界時から始まります。

戻り値

asctime_r() 関数は結果として生じた文字ストリングを指すポインターを戻します。関数が正常に実行されなかった場合、NULL を戻します。

asctime_r() の使用例

次の例はシステム・クロックをポーリングし、現在時刻を示すメッセージを出力します。

```
#include <time.h>
#include <stdio.h>

int main(void)
{
    struct tm *newtime;
    time_t ltime;
    char  mybuf[50];

/* Get the time in seconds */
    time(&ltime);
/* Convert it to the structure tm */
    newtime = localtime_r(&ltime());
/* Print the local time as a string */
    printf("The current date and time are %s",
             asctime_r(newtime, mybuf));
}

/****************  Output should be similar to  ******************
The current date and time are Fri Sep 16 132951 1994
*/
```

関連情報

- 41 ページの『asctime() ― 時間から文字ストリングへの変換』
- 75 ページの『ctime() ― 時間から文字ストリングへの変換』
- 77 ページの『ctime64() ― 時間から文字ストリングへの変換』
- 80 ページの『ctime64_r() ― 時間から文字ストリングへの変換 (再始動可能)』
- 78 ページの『ctime_r() ― 時間から文字ストリングへの変換 (再始動可能)』
- 167 ページの『gmtime() ― 時間の変換』
- 169 ページの『gmtime64() ― 時間の変換』
- 173 ページの『gmtime64_r() ― 時間の変換 (再始動可能)』
- 171 ページの『gmtime_r() ― 時間の変換 (再始動可能)』
- 192 ページの『localtime() ― 時間の変換』
- 194 ページの『localtime64() ― 時間の変換』
- 197 ページの『localtime64_r() ― 時間の変換 (再始動可能)』
- 195 ページの『localtime_r() ― 時間の変換 (再始動可能)』
- 228 ページの『mktime() ― 地方時の変換』
- 230 ページの『mktime64() ― 地方時の変換』
- 387 ページの『strftime() ― 日付/時刻からストリングへの変換』
- 429 ページの『time() ― 現在時刻の判別』
- 238 ページの『printf() ― 定様式の文字の出力』

- 19 ページの『<time.h>』

asin() ― 逆正弦の計算

フォーマット

```
#include <math.h>
double asin(double x);
```

言語レベル: ANSI

スレッド・セーフ: はい。

説明

`asin()` 関数は *x* の逆正弦を、$-\pi/2$ から $\pi/2$ ラジアンの範囲で計算します。

戻り値

`asin()` 関数は *x* の逆正弦を戻します。*x* の値は -1 と 1 の間でなければなりません。*x* が -1 よりも小さい、または 1 よりも大きい場合、`asin()` 関数は errno に EDOM を設定し、0 の値を戻します。

asin() の使用例

次の例は *x* の値に対してプロンプトを出します。*x* が 1 よりも大きいか -1 よりも小さい場合はエラー・メッセージを出力します。それ以外の場合は *x* の逆正弦を *y* に割り当てます。

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define MAX  1.0
#define MIN -1.0

int main(void)
{
  double x, y;

  printf( "Enter x¥n" );
  scanf( "%lf", &x );

  /* Output error if not in range */
  if ( x > MAX )
    printf( "Error: %lf too large for asin¥n", x );
  else if ( x < MIN )
    printf( "Error: %lf too small for asin¥n", x );
  else
  {
    y = asin( x );
    printf( "asin( %lf ) = %lf¥n", x, y );
  }
}

/****************  Output should be similar to  ******************
Enter x
asin( 0.200000 ) = 0.201358
*/
```

関連情報

- 40 ページの『acos() ― 逆余弦の計算』
- 47 ページの『atan() - atan2() ― 逆正接の計算』

- 68 ページの『cos() ― 余弦の計算』
- 69 ページの『cosh() ― 双曲線余弦の計算』
- 365 ページの『sin() ― 正弦の計算』
- 366 ページの『sinh() ― 双曲線正弦の計算』
- 427 ページの『tan() ― 正接の計算』
- 428 ページの『tanh() ― 双曲線正接の計算』
- 9 ページの『<math.h>』

assert() ― 条件の検証

フォーマット

```
#include <assert.h>
void assert(int expression);
```

言語レベル: ANSI

スレッド・セーフ: いいえ。

説明

`assert()` 関数は診断メッセージを `stderr` に出力し、*expression* が false (ゼロ) の場合にプログラムを異常終了します。診断メッセージの形式は以下のとおりです。

```
Assertion failed: expression, file filename, line line-number.
```

expression が true (ゼロ以外) の場合、`assert()` 関数は、アクションを実行しません。

プログラムの論理エラーを識別するには、`assert()` 関数を使用します。プログラムが意図したとおりに実行されている場合にのみ true を持つ *expression* を選択します。プログラムをデバッグした後で、特別の非デバッグ ID NDEBUG を使用してプログラムから `assert()` 呼び出しを除去することができます。#define ディレクティブを使用して NDEBUG を任意の値に定義した場合、C プリプロセッサーはすべての assert 呼び出しを void 式に展開します。NDEBUG を使用する場合は、<assert.h> をプログラムに組み込む前に、NDEBUG を定義する必要があります。

戻り値

戻り値はありません。

注: `assert()` 関数はマクロとして定義されます。 `assert()` で #undef ディレクティブを使用しないでください。

`assert()` の使用例

この例では、`assert()` 関数がヌル・ストリングおよび空ストリングについて *string* をテストし、これらの引数を処理する前に *length* が正であることを確認します。

```
#include <stdio.h>
#include <assert.h>

void analyze (char *, int);

int main(void)
{
   char *string = "ABC";
```

```
   int length = 3;

   analyze(string, length);
   printf("The string %s is not null or empty, "
          "and has length %d ¥n", string, length);
}

void analyze(char *string, int length)
{
   assert(string != NULL);     /* cannot be NULL */
   assert(*string != '¥0');    /* cannot be empty */
   assert(length > 0);         /* must be positive */
}

/****************  Output should be similar to  ******************
The string ABC is not null or empty, and has length 3
```

関連情報

- 38 ページの『abort() — プログラムの停止』
- 3 ページの『<assert.h>』

atan() - atan2() — 逆正接の計算

フォーマット

```
#include <math.h>
double atan(double x);
double atan2(double y, double x);
```

言語レベル: ANSI

スレッド・セーフ: はい。

説明

atan() 関数と atan2() 関数は、それぞれ *x* と *y/x* の逆正接を計算します。

戻り値

atan() 関数は -π/2 から π/2 ラジアンの範囲の値を戻します。 atan2() 関数は -π から π ラジアンの範囲の値を戻します。atan2() 関数の両方の引数がゼロの場合、この関数は errno に EDOM を設定し、0 を戻します。

atan() の使用例

この例では atan() 関数と atan2() 関数を使用して逆正接を計算します。

```
#include <math.h>
#include <stdio.h>

int main(void)
{
    double a,b,c,d;

    c = 0.45;
    d = 0.23;

    a = atan(c);
    b = atan2(c,d);

    printf("atan( %lf ) = %lf/n", c, a);
```

```
    printf("atan2( %lf, %lf ) = %lf/n", c, d, b);

}

/****************  Output should be similar to  *****************
atan( 0.450000 ) = 0.422854
atan2( 0.450000, 0.230000 ) = 1.098299
*****************************************************************/
```

関連情報

- 40 ページの『acos() ― 逆余弦の計算』
- 45 ページの『asin() ― 逆正弦の計算』
- 68 ページの『cos() ― 余弦の計算』
- 69 ページの『cosh() ― 双曲線余弦の計算』
- 365 ページの『sin() ― 正弦の計算』
- 366 ページの『sinh() ― 双曲線正弦の計算』
- 427 ページの『tan() ― 正接の計算』
- 428 ページの『tanh() ― 双曲線正接の計算』
- 9 ページの『<math.h>』

atexit() ― プログラム終了の記録関数

フォーマット

```
#include <stdlib.h>
int atexit(void (*func)(void));
```

言語レベル: ANSI

スレッド・セーフ: はい。

説明

atexit() 関数は、正常なプログラム終了時にシステムによって呼び出される *func* により示される関数を記録します。 atexit() 関数を使用して最大 32 の関数を登録し、移植性を改善できます。関数は後入れ先出し法で処理されます。 atexit() 関数は OPM デフォルトの活動化グループから呼び出すことができません。ほとんどの関数は atexit 関数で使用できますが、exit 関数が使用されると atexit 関数は失敗します。

戻り値

atexit() 関数は、正常に終了すると 0 を戻し、失敗するとゼロ以外の値を戻します。

atexit() の使用例

次の例では atexit() 関数を使用してプログラム終了時に goodbye() を呼び出します。

```
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
   void goodbye(void);
   int rc;
```

```
   rc = atexit(goodbye);
   if (rc != 0)
      perror("Error in atexit");
   exit(0);
}

void goodbye(void)
   /* This function is called at normal program end */
{
   printf("The function goodbye was called at program end¥n");
}

/****************  Output should be similar to:  ******************

The function goodbye was called at program end
*/
```

関連情報

- 92 ページの『exit() ― プログラムの終了』
- 362 ページの『signal() ― 割り込みシグナルの処理』
- 18 ページの『<stdlib.h>』

atof() ― 文字ストリングから浮動小数点への変換

フォーマット

```
#include <stdlib.h>
double atof(const char *string);
```

言語レベル: ANSI

スレッド・セーフ: はい。

ロケール依存: この関数の振る舞いは、現行ロケールの LC_CTYPE カテゴリーおよび LC_NUMERIC カテゴリーの影響を受ける可能性があります。詳細については、 549 ページの『CCSID およびロケールの理解』を参照してください。

説明

atof() 関数は文字ストリングを倍精度浮動小数点値に変換します。

入力データ *string* は、指定した戻りの型の数値として解釈できる文字のシーケンスです。関数によって、数値の一部として認識できない入力ストリングは、先頭文字のところで読み取りが停止されます。この文字は、ストリングを終了するヌル文字が可能です。

atof() 関数では、次の形式の *string* をとります。

空白文字は、複数のスペースやタブなどの、isspace() 関数に対して true である同じ文字で構成されます。 atof() 関数は、最初の空白文字を無視します。

atof() 関数の場合、*digits* は 1 桁以上の小数桁数です。小数点の前に桁数が表示されていない場合は、少なくとも 1 桁が小数点の後ろに表示される必要があります。小数桁数は指数よりも前に置くことができ、小数桁数の後ろに文字 e または E を入れて指数を表します。指数は 10 進数の整数で、符号付きの場合があります。

atof() 関数は、数字以外の文字が E の後ろに続く場合、または指数として e が読み込まれた場合には失敗します。例えば、100elf は浮動小数点値 100.0 に変換されます。精度は最大 17 桁の有効文字桁です。

戻り値

atof() 関数は、入力文字を数字として解釈することにより作成される double 値を戻します。関数が入力データをその型の値に変換できない場合は、戻り値は 0 です。オーバーフローの場合は、関数は errno に ERANGE を設定し、値 -HUGE_VAL または +HUGE_VAL を戻します。

atof() の使用例

この例は、ストリングとして格納された数を、数値に変換する方法を示しています。

```
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
    double x;
    char *s;

    s = " -2309.12E-15";
    x = atof(s);     /* x = -2309.12E-15 */

    printf("x = %.4e¥n",x);
}

/*******************  Output should be similar to:  ***************

x = -2.3091e-12
*/
```

関連情報

- 『atoi() ― 文字ストリングから整数への変換』
- 52 ページの『atol() ― atoll() ― 文字ストリングの long 型整数または long long 型整数への変換』
- 418 ページの『strtol() ― strtoll() ― 文字ストリングから long 型および long long 型整数への変換』
- 409 ページの『strtod() ― strtof() ― strtold ― 文字ストリングから double、浮動、および long double への変換』
- 412 ページの『strtod32() ― strtod64() ― strtod128() ― 文字ストリングから 10 進浮動小数点への変換』
- 18 ページの『<stdlib.h>』

atoi() ― 文字ストリングから整数への変換

フォーマット

```
#include <stdlib.h>
int atoi(const char *string);
```

言語レベル: ANSI

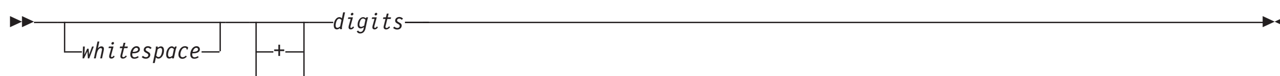
スレッド・セーフ: はい。

ロケール依存: この関数の振る舞いは、現行ロケールの LC_CTYPE カテゴリーの影響を受ける可能性があります。詳細については、 549 ページの『CCSID およびロケールの理解』を参照してください。

説明

`atoi()` 関数は、文字ストリングを整数値に変換します。入力データ *string* は、指定した戻りの型の数値として解釈できる文字のシーケンスです。関数によって、数値の一部として認識できない入力ストリングは、先頭文字のところで読み取りが停止されます。この文字は、ストリングを終了するヌル文字が可能です。

`atoi()` 関数は、小数点または指数を認識しません。この関数の string 引数の形式は次のとおりです。



ここで、*whitespace* は、複数のスペースやタブなどの、`isspace()` 関数に対して true である同じ文字で構成されます。`atoi()` 関数は、最初の空白文字を無視します。値 *digits* は、1 桁以上の小数桁数です。

戻り値

`atoi()` 関数は、入力文字を数字として解釈することにより作成される int 値を戻します。関数が入力データをその型の値に変換できない場合は、戻り値は 0 です。オーバーフローの場合、戻り値は予期できません。

`atoi()` の使用例

この例は、ストリングとして格納された数を、数値に変換する方法を示しています。

```
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
    int i;
    char *s;

    s = " -9885";
    i = atoi(s);     /* i = -9885 */

    printf("i = %d¥n",i);
}

/*******************  Output should be similar to:  ***************

i = -9885
*/
```

関連情報

- 49 ページの『atof() ― 文字ストリングから浮動小数点への変換』
- 52 ページの『atol() ― atoll() ― 文字ストリングの long 型整数または long long 型整数への変換』
- 409 ページの『strtod() ― strtof() ― strtold ― 文字ストリングから double、浮動、および long double への変換』
- 412 ページの『strtod32() ― strtod64() ― strtod128() ― 文字ストリングから 10 進浮動小数点への変換』

- 418 ページの『strtol() ― strtoll() ― 文字ストリングから long 型および long long 型整数への変換』
- 18 ページの『<stdlib.h>』

atol() ― atoll() ― 文字ストリングの long 型整数または long long 型整数への変換

フォーマット (atol())

```
#include <stdlib.h>
long int atol(const char *string);
```

フォーマット (atoll())

```
#include <stdlib.h>
long long int atoll(const char *string);
```

言語レベル: ANSI

スレッド・セーフ: はい。

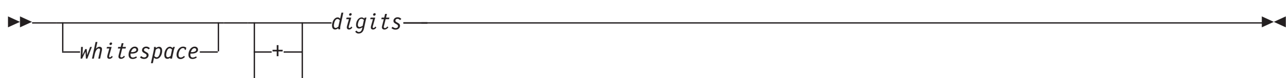
ロケール依存: これらの関数の振る舞いは、現行ロケールの LC_CTYPE カテゴリーの影響を受ける可能性があります。詳細については、 549 ページの『CCSID およびロケールの理解』を参照してください。

説明

atol() 関数は文字ストリングを long 型整数値に変換します。atoll() 関数は文字ストリングを long long 型整数値に変換します。

入力データ *string* は、指定した戻りの型の数値として解釈できる文字のシーケンスです。関数によって、数値の一部として認識できない入力ストリングは、先頭文字のところで読み取りが停止されます。この文字は、ストリングを終了するヌル文字が可能です。

atol() および atoll() 関数は、小数点または指数を認識しません。この関数の *string* 引数の形式は次のとおりです。



ここで、*whitespace* は、複数のスペースやタブなどの、isspace() 関数に対して true である同じ文字で構成されます。atol() および atoll() 関数は、最初の空白文字を無視します。値 *digits* は、1 桁以上の小数桁数です。

戻り値

atol() および atoll() 関数は、入力文字を数字として解釈することにより作成される long または long long 型整数値を戻します。関数が入力データをその型の値に変換できない場合は、戻り値は 0L です。オーバーフローの場合、戻り値は予期できません。

atol() の使用例

この例は、ストリングとして格納された数を、数値に変換する方法を示しています。

```
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
    long l;
    char *s;

    s = "98854 dollars";
    l = atol(s);     /* l = 98854 */

    printf("l = %.ld¥n",l);
}

/*******************  Output should be similar to:  ***************

l = 98854
*/
```

関連情報

- 49 ページの『atof() ― 文字ストリングから浮動小数点への変換』
- 50 ページの『atoi() ― 文字ストリングから整数への変換』
- 409 ページの『strtod() ― strtof() ― strtold ― 文字ストリングから double、浮動、および long double への変換』
- 412 ページの『strtod32() ― strtod64() ― strtod128() ― 文字ストリングから 10 進浮動小数点への変換』
- 418 ページの『strtol() ― strtoll() ― 文字ストリングから long 型および long long 型整数への変換』
- 18 ページの『<stdlib.h>』

ベッセル関数

フォーマット

```
#include <math.h>
double j0(double x);
double j1(double x);
double jn(int n, double x);
double y0(double x);
double y1(double x);
double yn(int n, double x);
```

言語レベル: ILE C Extension

スレッド・セーフ: はい。

説明

ベッセル関数はある種の微分式を解決します。関数 `j0()`、`j1()`、および `jn()` は、次数がそれぞれ 0、1、および *n* である第 1 種のベッセル関数です。関数 `y0()`、`y1()`、および `yn()` は、次数がそれぞれ 0、1、および *n* である第 2 種のベッセル関数です。

引数 *x* は正数でなければなりません。引数 *n* はゼロより大きいか等しくなっている必要があります。*n* がゼロ未満の場合は、負の指数になります。

戻り値

j0()、j1()、y0()、または y1() の場合、*x* の絶対値が大きすぎると、関数は errno に ERANGE を設定し、0 を戻します。y0()、y1()、または yn() では、*x* が負の場合、関数は errno を EDOM に設定し、値 -HUGE_VAL を戻します。y0、y1()、または yn() では、*x* がオーバーフローを起こした場合、関数は errno を ERANGE に設定し、値 -HUGE_VAL を戻します。

ベッセル関数 **の使用例**

この例は *x* に対して次数 0 の第 1 種ベッセル関数となる *y* を計算します。また、*x* について、次数 3 の第 2 種ベッセル関数となる *z* を計算します。

```
#include <math.h>
#include <stdio.h>

int main(void)
{
    double x, y, z;
    x = 4.27;

    y = j0(x);       /* y = -0.3660 is the order 0 bessel */
                     /* function of the first kind for x  */
    z = yn(3,x);     /* z = -0.0875 is the order 3 bessel */
                     /* function of the second kind for x */

    printf("y = %lf¥n", y);
    printf("z = %lf¥n", z);
}
/***************** Output should be similar to: **********************

 y = -0.366022
 z = -0.087482
**********************************************************************/
```

関連情報

- 91 ページの『erf() - erfc() ― 誤差関数の計算』
- 156 ページの『gamma() ― ガンマ関数』
- 9 ページの『<math.h>』

bsearch() ― 配列の検索

フォーマット

```
#include <stdlib.h>
void *bsearch(const void *key, const void *base,
              size_t num, size_t size,
              int (*compare)(const void *key, const void *element));
```

言語レベル: ANSI

スレッド・セーフ: はい。

説明

bsearch() 関数は、それぞれ *size* バイトの *num* エレメントの配列のバイナリー・サーチを行います。配列は、*compare* で指される関数によって、昇順で格納する必要があります。*base* は検索する配列の基数のポインターであり、*key* は検索される値です。

compare 引数は、2 つの項目を比較し、その関係を指定する値を戻す、ユーザーが指定しなければならない関数を指すポインターです。 compare() 関数の引数リストの最初の項目は、検索される項目の値を指すポ

インターです。 compare() 関数の引数リストの 2 番目の項目は、*key* と比較される配列 *element* を指すポインターです。compare() 関数はキー値を配列エレメントと比較してから、次の値のいずれかを戻す必要があります。

値	意味
0 より小さい値	*key* は *element* より小さい
0	*key* は *element* と等しい
0 より大きい値	*key* は *element* より大きい

戻り値

bsearch() 関数は、*base* が指す配列内の *key* を指すポインターを戻します。 2 つのキーが等しい場合、*key* が指すエレメントは未指定です。bsearch() 関数で *key* が見つからなかった場合は、NULL が戻されます。

bsearch() の使用例

この例はプログラム・パラメーターを指すポインターの argv 配列に対してバイナリー・サーチを行い、引数 PATH を検索します。まず、argv からプログラム名を除去し、それから配列をアルファベット順にソートしてから bsearch() を呼び出します。compare1() 関数および compare2() 関数は、*arg1* および *arg2* により指定される値を比較し、結果を bsearch() 関数に戻します。

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int compare1(const void *, const void *);
int compare2(const void *, const void *);

main(int argc, char *argv[])
{                          /* This program performs a binary        */
  char **result;           /* search on the argv array of pointers  */
  char *key = "PATH";      /* to the program parameters.  It first  */
  int i;                   /* removes the program name from argv    */
                           /* then sorts the array alphabetically   */
  argv++;                  /* before calling bsearch.               */
  argc--;

  qsort((char *)argv, argc, sizeof(char *), compare1);

  result = (char**)bsearch(&key, (char *)argv, argc, sizeof(char *), compare2);
  if (result != NULL)  {
      printf("result =<%s>¥n",*result);
   }
  else printf("result is null¥n");
}

         /*This function compares the values pointed to by arg1  */
         /*and arg2 and returns the result to qsort.  arg1 and   */
         /*arg2 are both pointers to elements of the argv array. */

int compare1(const void *arg1, const void *arg2)
{
    return (strcmp(*(char **)arg1, *(char **)arg2));
}

         /*This function compares the values pointed to by arg1  */
         /*and arg2 and returns the result to bsearch            */
         /*arg1 is a pointer to the key value, arg2 points to    */
         /*the element of argv that is being compared to the key */
```

```
        /*value.                                        */

int compare2(const void *arg1, const void *arg2)
{
    return (strcmp(*(char **)arg1, *(char **)arg2));
}
/********************  Output should be similar to:  *************

result = <PATH>

****************** When the input on the i5/OS command line is ********

CALL BSEARCH PARM(WHERE IS PATH IN THIS PHRASE'?')

*/
```

関連情報

- 256 ページの『qsort() ― 配列のソート』
- 18 ページの『<stdlib.h>』

btowc() ― 1 バイト文字のワイド文字への変換

フォーマット

```
#include <stdio.h>
#include <wchar.h>
wint_t btowc(int c);
```

言語レベル: ANSI

スレッド・セーフ: はい。

ロケール依存: この関数の振る舞いは、現行ロケールの LC_CTYPE カテゴリーの影響を受ける可能性があります。また、この振る舞いは、LOCALETYPE(*LOCALEUCS2) または LOCALETYPE(*LOCALEUTF) がコンパイル・コマンドに対して指定されている場合は、現行ロケールの LC_UNI_CTYPE カテゴリーの影響を受ける可能性もあります。この関数は、コンパイル・コマンドに対して LOCALETYPE(*CLD) が指定されている場合には使用できません。詳細については、 549 ページの『CCSID およびロケールの理解』を参照してください。

ワイド文字関数: 詳細については、 553 ページの『ワイド文字』を参照してください。

説明

`btowc()` 関数は 1 バイト値 *c* を *c* のワイド文字表記に変換します。初期シフト状態で *c* が有効な (1 バイトの) マルチバイト文字を構成していない場合は、`btowc()` 関数は WEOF を戻します。

戻り値

c が EOF の値を持っている場合、または (符号なし char) *c* が初期シフト状態で有効な (1 バイトの) マルチバイト文字を構成しない場合は、`btowc()` 関数は WEOF を戻します。その他の場合は、その文字のワイド文字表記を戻します。

変換エラーが発生した場合、`errno` は **ECONVERT** に設定される可能性があります。

`btowc()` の使用例

この例では、さまざまなタイプのデータを走査します。

```
#include <stdio.h>
#include <stdlib.h>
#include <wchar.h>
#include <local.h>

#define UPPER_LIMIT   0xFF

int main(void)
{
   int wc;
   int ch;
   if (NULL == setlocale(LC_ALL, "/QSYS.LIB/EN_US.LOCALE")) {
      printf("Locale could not be loaded¥n");
      exit(1);
   }
   for (ch = 0; ch <= UPPER_LIMIT; ++ch) {
      wc = btowc(ch);
      if (wc==WEOF) {
         printf("%#04x is not a one-byte multibyte character¥n", ch);
      } else {
         printf("%#04x has wide character representation: %#06x¥n", ch, wc);
      }
   }
  wc = btowc(EOF);
   if (wc==WEOF) {
      printf("The character is EOF.¥n", ch);
   } else {
      printf("EOF has wide character representation: %#06x¥n", wc);
   }
   return 0;
 }
  /***********************************************************************
      If the locale is bound to SBCS, the output should be similar to:
      0000 has wide character representation: 000000
      0x01 has wide character representation: 0x0001
      ...
      0xfe has wide character representation: 0x00fe
      0xff has wide character representation: 0x00ff
      The character is EOF.

  ************************************************************************/
```

関連情報

- 205 ページの『mblen() ― マルチバイト文字の長さの計算』
- 220 ページの『mbtowc() ― マルチバイト文字からワイド文字への変換』
- 210 ページの『mbrtowc() ― マルチバイト文字からワイド文字への変換 (再始動可能)』
- 214 ページの『mbsrtowcs() ― マルチバイト・ストリングからワイド文字ストリングへの変換 (再始動可能)』
- 354 ページの『setlocale() ― ロケールの設定』
- 467 ページの『wcrtomb() ― ワイド文字からマルチバイト文字への変換 (再開可能)』
- 494 ページの『wcsrtombs() ― ワイド文字ストリングからマルチバイト・ストリングへの変換 (再開可能)』
- 17 ページの『<stdio.h>』
- 20 ページの『<wchar.h>』

_C_Get_Ssn_Handle() ― C セッションへのハンドル

フォーマット

```
#include <stdio.h>

_SSN_HANDLE_T _C_Get_Ssn_Handle (void)
```

言語レベル: ILE C Extension

スレッド・セーフ: はい。

説明

Dynamic Screen Manager (DSM) API を操作するため C セッションにハンドルを戻します。

戻り値

`_C_Get_Ssn_Handle()` 関数はハンドルを C セッションに戻します。エラーが発生した場合、_SSN_HANDLE_T はゼロに設定されます。 DSM API での `_C_Get_Ssn_Handle()` 関数の使用について詳しくは、Information Center の『API』のトピックを参照してください。

calloc() ― ストレージの予約と初期化

フォーマット

```
#include <stdlib.h>
void *calloc(size_t num, size_t size);
```

言語レベル: ANSI

スレッド・セーフ: はい。

説明

`calloc()` 関数は、それぞれの長さが *size* バイトの *num* エレメントの配列用にストレージ・スペースを予約します。 `calloc()` 関数は次に、各エレメントのすべてのビットを初期値の 0 にします。

戻り値

`calloc()` 関数は、予約したスペースを指すポインターを戻します。戻り値が示すストレージ・スペースは、任意のタイプのオブジェクトのストレージ用に適切に位置合わせされます。型を指すポインターを取得するには、戻り値に対して型キャストを使用します。十分なストレージがない場合、あるいは *num* または *size* が 0 の場合は、戻り値は NULL です。

注:

1. すべてのヒープ・ストレージは、呼び出しルーチンの活動化グループと関連付けられます。したがって、ストレージの割り振りと割り振り解除は同じ活動化グループ内で行ってください。 1 つの活動化グループ内でヒープ・ストレージを割り振ったり、異なる活動化グループからそのストレージを割り振り解除したりすることはできません。活動化グループについて詳しくは、「*ILE Concepts*」のマニュアルを参照してください。
2. C ソース・コードを変更せずに、単一レベル・ストア・ストレージの代わりに**テラスペース**・ストレージを使用する場合、コンパイラー・コマンドで TERASPACE(*YES *TSIFC) パラメーターを指定します。これにより、`calloc()` ライブラリー関数が `_C_TS_calloc()` (テラスペース・ストレージでの `calloc()` ライブラリー関数のカウンター・パート) にマップされます。`_C_TS_calloc()` への各呼び出しにより割り振り可能なテラスペース・ストレージの最大量は、2GB - 224 バイト、つまり 2,147,483,424 バイトです。

| テラスペース・ストレージについて詳しくは、「*ILE Concepts*」のマニュアルを参照してください。

| 3. 高速プール・メモリー・マネージャーが現行の活動化グループ内で使用可能にされている場合、ストレージは高速プール・メモリー・マネージャーを使用して検索されます。詳細については、 71 ページの『_C_Quickpool_Init() — 高速プール・メモリー・マネージャーの初期化』を参照してください。

calloc() の使用例

次の例は必要な数の配列エントリーを求めるプロンプトを出し、それからエントリー用にストレージ内に十分なスペースを予約します。 calloc() が成功すると、例は各エントリーを出力します。それ以外の場合はエラーを出力します。

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
  long * array;                            /* start of the array
   */
  long * index;                            /* index variable
   */
  int    i;                                /* index variable
   */
  int  num;                                /* number of entries of the array
*/

  printf( "Enter the size of the array¥n" );
  scanf( "%i", &num);

                                            /* allocate num entries */
  if ( (index = array = (long *) calloc( num, sizeof( long ))) != NULL )
  {

    for ( i = 0; i < num; ++i )             /* put values in arr    */
       *index++ = i;                        /* using pointer no     */

    for ( i = 0; i < num; ++i )             /* print the array out  */
      printf( "array[%i ] = %i¥n", i, array[i] );
  }
  else
  { /* out of storage */
    perror( "Out of storage" );
    abort();
  }
}
/******************  Output should be similar to:  **********************

Enter the size of the array
array[ 0 ] = 0
array[ 1 ] = 1
array[ 2 ] = 2
*/
```

関連情報

- 69 ページの『_C_Quickpool_Debug() — 高速プール・メモリー・マネージャー特性の変更』
- 71 ページの『_C_Quickpool_Init() — 高速プール・メモリー・マネージャーの初期化』
- 73 ページの『_C_Quickpool_Report() — 高速プール・メモリー・マネージャー・レポートの生成』
- | 564 ページの『ヒープ・メモリー』
- 134 ページの『free() — ストレージ・ブロックの解放』
- 203 ページの『malloc() — ストレージ・ブロックの予約』

- 276 ページの『realloc() ― 予約ストレージ・ブロック・サイズの変更』
- 18 ページの『<stdlib.h>』

catclose() ―メッセージ・カタログのクローズ

フォーマット

```
#include <nl_types.h>
int catclose (nl_catd catd);
```

言語レベル: XPG4

スレッド・セーフ: はい。

ロケール依存: この関数は、コンパイル・コマンドに対して LOCALETYPE(*CLD) が指定されている場合には使用できません。

統合ファイル・システム・インターフェース: この関数は、コンパイル・コマンドに対して SYSIFCOPT(*NOIFSIO) が指定されている場合には使用できません。

説明

`catclose()` 関数は、*catd* により識別される、以前にオープンされたメッセージ・カタログをクローズします。

戻り値

クローズが正常に実行されると、0 が戻されます。それ以外の場合は、-1 が戻されます。これは失敗を示しており、*catd* が有効なメッセージ・カタログ記述子でない場合に起こることがあります。

errno の値は、次のいずれかに設定されます。

EBADF
: カタログ記述子が無効です。

EINTR
: シグナルが関数に割り込みました。

`catclose()` の使用例

```
#include <stdio.h>
#include <nl_types.h>
#include <locale.h>

/* Name of the message catalog is "/qsys.lib/mylib.lib/msgs.usrspc" */

int main(void) {

   nl_catd msg_file;
   char * my_msg;
   char * my_locale;

   setlocale(LC_ALL, NULL);
   msg_file = catopen("/qsys.lib/mylib.lib/msgs.usrspc", 0);

   if (msg_file != CATD_ERR)  {

     my_msg = catgets(msg_file, 1, 2, "oops");

     printf("%s¥n", my_msg);

     catclose(msg_file);
   }
}
```

関連情報

- 63 ページの『catopen() ―メッセージ・カタログのオープン』
- 『catgets() ― メッセージ・カタログからのメッセージの検索』

---

catgets() ― メッセージ・カタログからのメッセージの検索

フォーマット

```
#include <nl_types.h>
char *catgets(nl_catd catd, int set_id, int msg_id, char *s);
```

言語レベル: XPG4

スレッド・セーフ: はい。

ロケール依存: この関数の振る舞いは、現行ロケールの LC_CTYPE カテゴリーの影響を受ける可能性があります。この関数は、コンパイル・コマンドに対して LOCALETYPE(*CLD) が指定されている場合には使用できません。詳細については、 549 ページの『CCSID およびロケールの理解』を参照してください。

統合ファイル・システム・インターフェース: この関数は、コンパイル・コマンドに対して SYSIFCOPT(*NOIFSIO) が指定されている場合には使用できません。

説明

`catgets()` 関数は、セット *set_id* 内で、*catd* により識別されるメッセージ・カタログ内でメッセージ *msg_id* を検索します。*catd* は、以前の `catopen()` への呼び出しにより戻されるメッセージ・カタログ記述子です。*s* 引数は、識別メッセージを検索できない場合に `catgets()` が戻すデフォルト・メッセージを指します。

戻り値

メッセージが正常に検索できる場合は、catgets() はメッセージ・カタログに含まれるメッセージ・ストリングを指すポインターを戻します。検索されたメッセージの CCSID は、メッセージ・カタログ・ファイルがオープンしたときに、catopen() 関数への以前の呼び出しに対して *oflag* パラメーターで指定されたフラグにより判別されます。

- NL_CAT_JOB_MODE フラグが指定された場合は、検索されたメッセージはジョブの CCSID にあります。
- NL_CAT_CTYPE_MODE フラグが指定された場合は、検索されたメッセージは現行ロケールの LC_CTYPE カテゴリーの CCSID にあります。
- いずれのフラグも指定されていない場合は、検索されたメッセージの CCSID はメッセージ・カタログ・ファイルの CCSID に一致します。

メッセージが正常に検索されない場合は、デフォルト・ストリング *s* を指すポインターが戻されます。

errno の値は、次のいずれかに設定されます。

EBADF
カタログ記述子が無効です。

ECONVERT
変換エラーが発生しました。

EINTR
シグナルが関数に割り込みました。

catgets() の使用例

```
#include <stdio.h>
#include <nl_types.h>
#include <locale.h>

/* Name of the message catalog is "/qsys.lib/mylib.lib/msgs.usrspc" */

int main(void) {

   nl_catd msg_file;
   char * my_msg;
   char * my_locale;

   setlocale(LC_ALL, NULL);
   msg_file = catopen("/qsys.lib/mylib.lib/msgs.usrspc", 0);

   if (msg_file != CATD_ERR)  {

     my_msg = catgets(msg_file, 1, 2, "oops");

     printf("%s¥n", my_msg);

     catclose(msg_file);
   }
}
```

関連情報

- 60 ページの『catclose() —メッセージ・カタログのクローズ』
- 63 ページの『catopen() —メッセージ・カタログのオープン』

catopen() —メッセージ・カタログのオープン

フォーマット

```
#include <nl_types.h>
nl_catd catopen(const char *name, int oflag);
```

言語レベル: XPG4

スレッド・セーフ: はい。

ロケール依存: この関数の振る舞いは、現行ロケールの LC_MESSAGES カテゴリーの影響を受ける可能性があります。この関数は、コンパイル・コマンドに対して LOCALETYPE(*CLD) が指定されている場合には使用できません。詳細については、 549 ページの『CCSID およびロケールの理解』を参照してください。

統合ファイル・システム・インターフェース: この関数は、コンパイル・コマンドに対して SYSIFCOPT(*NOIFSIO) が指定されている場合には使用できません。

説明

`catopen()` 関数はメッセージ・カタログをオープンします。これは、メッセージを検索する前に実行する必要があります。名前にスラッシュ (/) が見つからない場合は、NLSPATH 環境変数および LC_MESSAGES カテゴリーを使用して、指定されたメッセージ・カタログを検索します。名前に 1 つ以上のスラッシュ (/) 文字が含まれている場合は、その名前はオープンするカタログのパス名と解釈されます。

NLSPATH 環境変数がない場合、あるいは NLSPATH で指定されたパスにメッセージ・カタログが見つからない場合は、デフォルトのパスが使用されます。デフォルトのパスは、LANG 環境変数の設定により影響を受ける可能性があります。例えば、 NL_CAT_LOCALE フラグが *oflag* パラメーターに設定されている場合や LANG 環境変数が設定されていない場合は、デフォルトのパスが LC_MESSAGES ロケール・カテゴリーにより影響を受ける可能性があります。

oflag パラメーターには、NL_CAT_LOCALE、NL_CAT_JOB_MODE、および NL_CAT_CTYPE_MODE の、3 つの値を指定できます。NL_CAT_JOB_MODE および NL_CAT_CTYPE_MODE は互いに排他的です。NL_CAT_JOB_MODE および NL_CAT_CTYPE_MODE フラグがどちらも *oflag* パラメーターに指定されている場合は、 catopen() 関数は戻り値 CATD_ERR を戻して失敗します。

カタログ・メッセージが `catgets()` 関数により戻される前にジョブ CCSID に変換したい場合は、パラメーターを NL_CAT_JOB_MODE に設定します。カタログ・メッセージが `catgets()` により戻される前に LC_CTYPE CCSID に変換したい場合は、パラメーターを NL_CAT_CTYPE_MODE に設定します。パラメーターを NL_CAT_JOB_MODE にも NL_CAT_CTYPE_MODE にも設定しない場合は、メッセージは変換されずに戻され、メッセージ・ファイルの CCSID にあります。

メッセージ・カタログ記述子は、`catclose()` への呼び出しによりクローズされるまで有効です。LC_MESSAGES ロケール・カテゴリーが変更されると、既存の開いたメッセージ・カタログが無効になることがあります。

注: メッセージ・カタログの名前は、有効な統合ファイル・システム・ファイル名である必要があります。

戻り値

メッセージ・カタログが正常にオープンされると、有効なカタログ記述子が戻されます。 `catopen()` が正常に行われないと、CATD_ERR ((nl_catd)-1) が戻されます。

catopen() 関数は、以下の状態では失敗することがあり、errno の値は以下のいずれかに設定できます。

EACCES
指定されたメッセージ・カタログを読み取るための十分な権限がないか、指定されたメッセージ・カタログのパス接頭部のコンポーネントを検索する十分な権限がありません。

ECONVERT
変換エラーが発生しました。

EMFILE
NL_MAXOPEN メッセージ・カタログは現在オープンしています。

ENAMETOOLONG
メッセージ・カタログのパス名の長さが PATH_MAX を超えているか、パス名コンポーネントが NAME_MAX より長くなっています。

ENFILE
システムで大量のファイルが現在オープンされています。

ENOENT
メッセージ・カタログが存在していないか、name 引数が空ストリングを指しています。

catopen() の使用例

```
#include <stdio.h>
#include <nl_types.h>
#include <locale.h>

/* Name of the message catalog is "/qsys.lib/mylib.lib/msgs.usrspc" */

int main(void) {

   nl_catd msg_file;
   char * my_msg;
   char * my_locale;

   setlocale(LC_ALL, NULL);
   msg_file = catopen("/qsys.lib/mylib.lib/msgs.usrspc", 0);

   if (msg_file != CATD_ERR)  {

     my_msg = catgets(msg_file, 1, 2, "oops");

     printf("%s¥n", my_msg);

     catclose(msg_file);
   }
}
```

関連情報

- 60 ページの『catclose() ―メッセージ・カタログのクローズ』
- 61 ページの『catgets() ― メッセージ・カタログからのメッセージの検索』

ceil() ― 整数の検索 >= 引数

フォーマット

```
#include <math.h>
double ceil(double x);
```

言語レベル: ANSI

スレッド・セーフ: はい。

説明

ceil() 関数は、*x* 以上の最小整数を計算します。

戻り値

ceil() 関数は整数を double 値として戻します。

ceil() の使用例

次の例は *y* を 1.05 よりも大きい最小整数に設定し、次に -1.05 よりも大きい最小整数に設定します。結果はそれぞれ、2.0 と -1.0 です。

```
#include <math.h>
#include <stdio.h>

int main(void)
{
   double y, z;

   y = ceil(1.05);       /* y = 2.0 */
   z = ceil(-1.05);      /* z = -1.0 */

   printf("y = %.2f ; z = %.2f¥n", y, z);
}
/***************** Output should be similar to: ***********************

 y = 2.00 ; z = -1.00
**********************************************************************/
```

関連情報

- 112 ページの『floor() ― 整数の検索 <=引数』
- 113 ページの『fmod() ― 浮動小数点の剰余の計算』
- 9 ページの『<math.h>』

---

clearerr() ― エラー標識のリセット

フォーマット

```
#include <stdio.h>
void clearerr (FILE *stream);
```

言語レベル: ANSI

スレッド・セーフ: はい。

説明

clearerr() 関数は、指定した *stream* のエラー標識とファイル終了標識をリセットします。いったん設定されると、指定したストリームの標識は、プログラムが clearerr() 関数または rewind() 関数を呼び出す

まで設定されたままになります。また、fseek() 関数はファイル終了標識もクリアします。ILE C/C++ ランタイム環境は、エラー標識やファイル終了標識を自動的にクリアしません。

戻り値

戻り値はありません。

errno の値は、次のいずれかに設定されます。

値　　意味

EBADF
ファイル・ポインター、または記述子が有効ではありません。

ENOTOPEN
ファイルはオープンされていません。

ESTDIN
stdin をオープンできません。

EIOERROR
リカバリー不能な入出力エラーが発生しました。

EIORECERR
リカバリー可能な入出力エラーが発生しました。

clearerr() の使用例

次の例はデータ・ストリームを読み取り、次に読み取りエラーが発生していないことを確認します。

```
#include <stdio.h>
#include <stdlib.h>

FILE *stream;
int c;

int main(void)
{
   if ((stream = fopen("mylib/myfile", "r")) != NULL)
   {
      if ((c=getc(stream)) == EOF)
      {
         if (ferror(stream))
         {
            perror("Read error");
            clearerr(stream);
         }
      }
   }
   else
      exit(0);
}
```

関連情報

- 99 ページの『feof() — ファイル終了標識のテスト』
- 100 ページの『ferror() — 読み取り/書き込みエラーのテスト』
- 140 ページの『fseek() — fseeko() — ファイル位置の位置変更』
- 236 ページの『perror() — エラー・メッセージの出力』
- 288 ページの『rewind() — 現在のファイル位置の調整』

- 384 ページの『strerror() ― ランタイム・エラー・メッセージを指すポインターの設定』
- 17 ページの『<stdio.h>』

clock() ― プロセッサー時間の判別

フォーマット

```
#include <time.h>
clock_t clock(void);
```

言語レベル: ANSI

スレッド・セーフ: はい。

説明

clock() 関数は、プログラムが使用したプロセッサー時間の概算を戻します。プロセス呼び出しに関連した、実装定義の時間枠の開始以降の経過時間です。秒単位の時間を取得するには、clock() で戻された値をマクロ CLOCKS_PER_SEC の値で割ります。

戻り値

プロセッサー時間の値が使用できないか、表示できない場合は、clock() 関数は値 (clock_t)-1 を戻します。

プログラムで使用された時間を測るには、プログラムの先頭で clock() を呼び出し、その戻り値を、その後の clock() の呼び出しで戻った値から減算してください。他のプラットフォームでは、clock() 関数をいつも信頼することはできません。system() 関数の呼び出しがクロックをリセットする可能性があるからです。

clock() の使用例

次の例は、プログラムが呼び出されてから経過した時間を出力します。

```
#include <time.h>
#include <stdio.h>

double time1, timedif;       /* use doubles to show small values */

int main(void)
{
    int  i;

    time1 = (double) clock();            /* get initial time */
    time1 = time1 / CLOCKS_PER_SEC;      /*    in seconds    */

    /* running the FOR loop 10000 times */
    for (i=0; i<10000; i++);

    /* call clock a second time */
    timedif = ( ((double) clock()) / CLOCKS_PER_SEC) - time1;
    printf("The elapsed time is %lf seconds¥n", timedif);
}
```

関連情報

- 86 ページの『difftime() ― 時差の計算』
- 88 ページの『difftime64() ― 時差の計算』

- 429 ページの『time() ― 現在時刻の判別』
- 430 ページの『time64() ― 現在時刻の判別』
- 19 ページの『<time.h>』

cos() ― 余弦の計算

フォーマット

```
#include <math.h>
double cos(double x);
```

言語レベル: ANSI

スレッド・セーフ: はい。

説明

`cos()` 関数は *x* の余弦を計算します。値 *x* は、ラジアンで表示されます。*x* が大きすぎると、結果の有効数字が部分的に消失することがあります。

戻り値

`cos()` 関数は *x* の余弦を計算します。errno の値は EDOM または ERANGE に設定される可能性があります。

`cos()` の使用例

次の例は、*x* の余弦となる *y* を計算します。

```
#include <math.h>
#include <stdio.h>

int main(void)
{
   double x, y;

   x = 7.2;
   y = cos(x);

   printf("cos( %lf ) = %lf¥n", x, y);
}

/*********************  Output should be similar to:  *******************

cos( 7.200000 ) = 0.608351
*/
```

関連情報

- 40 ページの『acos() ― 逆余弦の計算』
- 69 ページの『cosh() ― 双曲線余弦の計算』
- 365 ページの『sin() ― 正弦の計算』
- 366 ページの『sinh() ― 双曲線正弦の計算』
- 427 ページの『tan() ― 正接の計算』
- 428 ページの『tanh() ― 双曲線正接の計算』
- 9 ページの『<math.h>』

cosh() ― 双曲線余弦の計算

フォーマット

```
#include <math.h>
double cosh(double x);
```

言語レベル: ANSI

スレッド・セーフ: はい。

説明

cosh() 関数は *x* の双曲線余弦を計算します。値 *x* は、ラジアンで表示されます。

戻り値

cosh() 関数は *x* の双曲線余弦を戻します。結果が大きすぎると cosh() は値 HUGE_VAL を戻し、errno に ERANGE を設定します。

cosh() の使用例

次の例は、*x* の双曲線余弦となる *y* を計算します。

```
#include <math.h>
#include <stdio.h>

int main(void)
{
   double x,y;

   x = 7.2;
   y = cosh(x);

   printf("cosh( %lf ) = %lf¥n", x, y);
}

/*********************  Output should be similar to:  *******************

cosh( 7.200000 ) = 669.715755
*/
```

関連情報

- 40 ページの『acos() ― 逆余弦の計算』
- 68 ページの『cos() ― 余弦の計算』
- 365 ページの『sin() ― 正弦の計算』
- 366 ページの『sinh() ― 双曲線正弦の計算』
- 427 ページの『tan() ― 正接の計算』
- 428 ページの『tanh() ― 双曲線正接の計算』
- 9 ページの『<math.h>』

_C_Quickpool_Debug() ― 高速プール・メモリー・マネージャー特性の変更

フォーマット

```
#include <stdlib.h>
_C_Quickpool_Debug_T _C_Quickpool_Debug(_C_Quickpool_Debug_T *newval);
```

言語レベル: Extended

スレッド・セーフ: はい。

説明

| `_C_Quickpool_Debug()` 関数は、高速プール・メモリー・マネージャー特性を変更します。環境変数を使用
| してこのサポートを構成することもできます (セクション 577 ページの『環境変数』を参照してくださ
| い)。

`_C_Quickpool_Debug()` のパラメーターは以下のとおりです。

newval `_C_Quickpool_Debug_T` 構造体を指すポインター。この構造体には以下のフィールドがあります。

フラグ 変更される特性を指す符号なし整数値。フラグ・フィールドは次の値を含むことができます (任意の組み合わせで使用できます)。

`_C_INIT_MALLOC`
| 割り振られたすべてのストレージを指定した値に初期設定します。

`_C_INIT_FREE`
| 解放されたすべてのストレージを指定した値に初期設定します。

`_C_COLLECT_STATS`
| `_C_Quickpool_Report()` 関数で使用するために、高速プール・メモリー・マネージ
| ャーに関する統計を収集します。

malloc_val
| 割り振り済みメモリーの各バイトの初期設定に使用される、1 バイトの符号なし文字の
| 値。このフィールドは、`_C_INIT_MALLOC` フラグが指定されている場合にのみ、使用されま
| す。

free_val
解放済みメモリーの各バイトの初期設定に使用される、1 バイトの符号なし文字の値。このフィールドは、`_C_INIT_FREE` フラグが指定されている場合にのみ使用されます。

| *newval* の値が NULL である場合、現行の高速プール・メモリー・マネージャー特性を含む構造体が戻さ
| れ、高速プール・メモリー・マネージャー特性は何も変更されません。

戻り値

| 戻り値は、現行の関数呼び出しで要求された変更が行われる前に `_C_Quickpool_Debug()` 値を含む構造体で
| す。この値を後続の呼び出しで使用することにより、`_C_Quickpool_Debug()` 値を前の状態に復元すること
| ができます。

`_C_Quickpool_Debug()` の使用例

次の例では `_C_Quickpool_Debug()` を `_C_INIT_MALLOC` フラグおよび `_C_INIT_FREE` フラグと共に使用して、`malloc` 関数および `free` 関数のメモリーを初期化します。

```
#include <stdlib.h>
#include <stdio.h>
int main(void) {
  char *p;
  char *mtest = "AAAAAAAAAA";
  char *ftest = "BBBBBBBBBB";
  unsigned int cell_sizes[2]       = { 16, 64 };
  unsigned int cells_per_extent[2] = { 16, 16 };
  _C_Quickpool_Debug_T  dbgVals = { _C_INIT_MALLOC | _C_INIT_FREE, 'A', 'B' };

  if (_C_Quickpool_Init(2, cell_sizes, cells_per_extent)) {
    printf("Error initializing Quick Pool memory manager.\n");
    return -1;
  }

  _C_Quickpool_Debug(&dbgVals);

  if ((p = malloc(10)) == NULL) {
    printf("Error during malloc.¥n");
    return -2;
  }
  if (memcmp(p, mtest, 10)) {
    printf("malloc test failed¥n");
  }
  free(p);
  if (memcmp(p, ftest, 10)) {
    printf("free test failed¥n");
  }
  printf("Test successful!¥n");
  return 0;
}
/*****************Output should be similar to:*****************
Test successful!
*******************************************************************/
```

関連情報

- 『_C_Quickpool_Init() ― 高速プール・メモリー・マネージャーの初期化』
- 73 ページの『_C_Quickpool_Report() ― 高速プール・メモリー・マネージャー・レポートの生成』
- 18 ページの『<stdlib.h>』
- 564 ページの『ヒープ・メモリー』

_C_Quickpool_Init() ― 高速プール・メモリー・マネージャーの初期化

フォーマット

```
#include <stdlib.h>
int _C_Quickpool_Init(unsigned int numpools, unsigned int *cell_sizes, unsigned int *num_cells);
```

言語レベル: Extended

スレッド・セーフ: はい。

説明

`_C_Quickpool_Init()` 関数が呼び出されると、同じ活動化グループ内のメモリー・マネージャー関数 (`malloc`、`calloc`、`realloc`、および `free`) へのすべての後続の呼び出しは、高速プール・メモリー・マネージャーを使用します。このメモリー・マネージャーは、一部のアプリケーションのパフォーマンスを改善します。

高速プール・メモリー・マネージャーは、メモリーを一連のプールに分割します。各プールは同じサイズの多くのセルに分割されます。プールの数、各プールのセルのサイズ、および各プール・エクステントのセル数は、`_C_Quickpool_Init()` 関数を使用して設定されます。環境変数を使用してこのサポートを構成することもできます (セクション 577 ページの『環境変数』を参照してください)。

例えば、ユーザーが、それぞれのプールが 64 のセルを含む、4 つのプールを定義すると仮定します。最初のプールにはサイズが 16 バイトのセルがあり、2 番目のプールにはサイズが 256 バイトのセルがあり、3 番目のプールにはサイズが 1024 バイトのセルがあり、そして 4 番目のプールにはサイズが 2048 バイトのセルがあります。ストレージの要求が行われると、メモリー・マネージャーは、その要求をまず 1 つのプールに割り当てます。メモリー・マネージャーは、要求のストレージのサイズと所定のプールのセルのサイズとを比較します。

この例では、最初のプールは 1 バイトから 16 バイト間のサイズの要求を満たし、2 番目のプールは 17 バイトから 256 バイトのサイズの要求を、3 番目のプールは 257 バイトから 1024 バイトのサイズを、そして 4 番目のプールは 1025 バイトから 2048 バイトのサイズを満たします。最大セル・サイズよりも大きな要求はすべて、デフォルト・メモリー・マネージャーにより割り振られます。

プールに割り当てられてから、プールのフリー・キューが検査されます。各プールには、解放されまだ再割り振りされていないセルを含むフリー・キューがあります。フリー・キューにセルがある場合は、そのセルはフリー・キューから除去され、戻されます。それ以外の場合は、セルは現行のプールのエクステントから取り出されます。エクステントとは、1 ブロックとして割り当てられるセルの集合です。最初は、プールにはエクステントがありません。

プールに対して最初の要求が発生すると、1 つのエクステントがそのプールについて割り振られ、要求はこのエクステントから満たされます。そのエクステントがすべて使用されるまで、このプールに対する後続の要求も、そのエクステントによって満たされます。エクステントがいっぱいになると、プールに新しいエクステントが割り振られます。新しいエクステントの割り振りができない場合、メモリー障害が存在していると見なされます。デフォルト・メモリー・マネージャーを使用してストレージを割り振るための試みが行われます。この試行が正常に行われない場合は、NULL 値が戻されます。

numpools
: 高速プール・メモリー・マネージャーで使用するプールの数。このパラメーターには、1 から 64 の間の値を指定できます。

cell_sizes
: 符号なし整数値の配列。配列のエントリー数は、numpools パラメーターに指定した数と同数です。各エントリーは、所定のプールの 1 つのセル内のバイト数を指定します。これらの値は 16 バイトの倍数である必要があります。この値が 16 バイトの倍数でない数に指定された場合、そのセル・サイズは 16 バイトの倍数に一番近い数に切り上げられます。最小有効値は 16 バイトで最大有効値は 4096 バイトです。

num_cells
: 符号なし整数値の配列。配列のエントリー数は、numpools パラメーターに指定した数と同数です。各エントリーは、該当するプールの単一エクステント内のセル数を指定します。各値には任意の非負数を指定できますが、アーキテクチャーによる制約のために各エクステントの合計サイズは制限される場合があります。値ゼロは、実装で大きな値を選択する必要があることを示します。

以下は前述の例に対しての、`_C_Quickpool_Init()` の呼び出しです。

```
unsigned int cell_sizes[4] = { 16, 256, 1024, 2048 };
unsigned int cells_per_extent[4] = { 64, 64, 64, 64 };
rc = _C_Quickpool_Init(4,                        /* number of pools            */
                       cell_sizes,               /* cell sizes for each pool   */
                       cells_per_extent);        /* extent sizes for each pool */
```

戻り値

以下のリストは、_C_Quickpool_Init() 関数の戻り値を示しています。

0 Success

-1 代替メモリー・マネージャーが、この活動化グループ用にすでに使用可能になっています。

-2 制御構造のストレージの割り振り中にエラーが発生しました。

-3 無効なプール数が指定されました。

-4 _C_Quickpool_Init() が、無効な活動化グループから呼び出されました。

-5 _C_Quickpool_Init() の実行中に予期しない例外が発生しました。

_C_Quickpool_Init() の使用例

次の例では _C_Quickpool_Init() を使用して、高速プール・メモリー割り振りを使用可能にします。

```
#include <stdlib.h>
#include <stdio.h>
int main(void) {
  char *p;
  unsigned int cell_sizes[2]       = { 16, 64 };
  unsigned int cells_per_extent[2] = { 16, 16 };

  if (_C_Quickpool_Init(2, cell_sizes, cells_per_extent) {
    printf("Error initializing Quick Pool memory manager.\n");
    return -1;
  }
  if ((p = malloc(10)) == NULL) {
    printf("Error during malloc.¥n");
    return -2;
  }
  free(p);
  printf("Test successful!¥n");
  return 0;
}
/*****************Output should be similar to:*****************
Test successful!
*******************************************************************/
```

関連情報

- 69 ページの『_C_Quickpool_Debug() — 高速プール・メモリー・マネージャー特性の変更』
- 『_C_Quickpool_Report() — 高速プール・メモリー・マネージャー・レポートの生成』
- 18 ページの『<stdlib.h>』
- 564 ページの『ヒープ・メモリー』

_C_Quickpool_Report() — 高速プール・メモリー・マネージャー・レポートの生成

フォーマット

```
#include <stdlib.h>
void _C_Quickpool_Report(void);
```

言語レベル: Extended

スレッド・セーフ: はい。

説明

| _C_Quickpool_Report() 関数は、現行の活動化グループ内で高速プール・メモリー・マネージャーにより使用されるメモリーのスナップショットを含む、スプール・ファイルを生成します。高速プール・メモリー・マネージャーが現行の活動化グループで使用可能にされていない場合、または統計収集が使用可能にされていない場合、レポートは、収集されるデータは何もないことを示すメッセージになります。

| 高速プール・メモリー・マネージャーが使用可能にされており、さらに統計収集が使用可能にされている場合、生成されるレポートは、統計収集が使用可能にされている間の各 16 バイトのメモリーに対する割り振り試行の回数を示します。また、レポートは各プールに達した残りの割り振りの最大数 (ピーク割り振り) を示します。指定したメモリー範囲に対してストレージ要求がなされない場合、そのメモリー範囲はレポートに含まれません。 最大セル・サイズ (4096 バイト) よりも大きい割り振りは、出力されません。

戻り値

この関数に対する戻り値はありません。

_C_Quickpool_Report() の使用例

| 次の例では _C_Quickpool_Init() を使用して、高速プール・メモリー・マネージャーを使用可能にします。_C_COLLECT_STATS フラグを使用して情報を収集します。収集された情報は、_C_Quickpool_Report() を使用して出力されます。
|

```
#include <stdlib.h>
#include <stdio.h>
int main(void) {
  char *p;
  int   i;
  unsigned int cell_sizes[2]       = { 16, 64 };
  unsigned int cells_per_extent[2] = { 16, 16 };
  _C_Quickpool_Debug_T  dbgVals = { _C_COLLECT_STATS, 'A', 'B' };

  if (_C_Quickpool_Init(2, cell_sizes, cells_per_extent) {
    printf("Error initializing Quick Pool memory manager.\n");
    return -1;
  }

  _C_Quickpool_Debug(&dbgVals);

  for (i = 1; i <= 64; i++) {
    p = malloc(i);
    free(p);
  }
  p = malloc(128);
  free(p);
  _C_Quickpool_Report();
  return 0;
}
/*****************Spooled File Output should be similar to:**********
Pool 1 (16 bytes, 1 peak allocations):
1-16 bytes: 16 allocations
Pool 2 (64 bytes, 1 peak allocations):
17-32 bytes: 16 allocations
33-48 bytes: 16 allocations
49-64 bytes: 16 allocations
Remaining allocations smaller than the largest cell size (4096 bytes):
113-128 bytes: 1 allocations
********************************************************************/
```

関連情報

- 69 ページの『_C_Quickpool_Debug() ― 高速プール・メモリー・マネージャー特性の変更』
- 71 ページの『_C_Quickpool_Init() ― 高速プール・メモリー・マネージャーの初期化』
- 18 ページの『<stdlib.h>』
- 564 ページの『ヒープ・メモリー』

ctime() ― 時間から文字ストリングへの変換

フォーマット

```
#include <time.h>
char *ctime(const time_t *time);
```

言語レベル: ANSI

スレッド・セーフ: いいえ。代わりに `ctime_r()` を使用します。

ロケール依存: この関数の振る舞いは、現行ロケールの LC_TOD カテゴリーの影響を受ける可能性があります。詳細については、 549 ページの『CCSID およびロケールの理解』を参照してください。

説明

`ctime()` 関数は、*time* が指す時間の値を文字ストリング形式の現地時間に変換します。時間の値は通常、`time()` 関数を呼び出して取得します。

ctime() が作成するストリングの結果にはちょうど 26 文字が含まれ、以下の形式となっています。

```
"%.3s %.3s%3d %.2d:%.2d:%.2d %d¥n"
```

以下に例を示します。

```
Mon Jul 16 02:03:55 1987\n\0
```

ctime() 関数は 24 時間クロック形式を使用します。曜日は、Sun、Mon、Tue、Wed、Thu、Fri、および Sat に省略されます。月の省略形は、Jan、Feb、Mar、Apr、May、Jun、Jul、Aug、Sep、Oct、Nov、および Dec です。すべてのフィールドには固定幅があります。1 桁しかない日付は、ゼロがその前に置かれます。改行文字 (¥n) および NULL 文字 (¥0) がストリングの最後の位置を占めます。

戻り値

ctime() 関数は、文字ストリングの結果を指すポインターを戻します。関数が正常に実行されなかった場合、NULL を戻します。ctime() 関数の呼び出しは次の式と同じです。

```
asctime(localtime(&anytime))
```

注: asctime() 関数と ctime() 関数、および他の時間関数は、戻りストリングの保持用に静的に割り振られた共通のバッファーを使用できます。これらの関数の 1 つを呼び出すごとに、以前の呼び出しの結果が破棄される可能性があります。asctime_r()、ctime_r()、gmtime_r()、および localtime_r() 関数は、戻りストリングの保持用に静的に割り振られた共通のバッファーを使用しません。再入可能性を希望する場合は、これらの関数を asctime()、ctime()、gmtime()、および localtime() の代わりに使用することができます。

ctime() の使用例

次の例では、time() を使用してシステム・クロックをポーリングします。ポーリングの次に、現在の日時を示すメッセージを出力します。

```
#include <time.h>
#include <stdio.h>

int main(void)
{
   time_t ltime;

   time(&ltime);

   printf("the time is %s", ctime(&ltime));
}
```

関連情報

- 41 ページの『asctime() ― 時間から文字ストリングへの変換』
- 43 ページの『asctime_r() ― 時間から文字ストリングへの変換 (再始動可能)』
- 78 ページの『ctime_r() ― 時間から文字ストリングへの変換 (再始動可能)』
- 77 ページの『ctime64() ― 時間から文字ストリングへの変換』
- 80 ページの『ctime64_r() ― 時間から文字ストリングへの変換 (再始動可能)』
- 167 ページの『gmtime() ― 時間の変換』
- 169 ページの『gmtime64() ― 時間の変換』
- 173 ページの『gmtime64_r() ― 時間の変換 (再始動可能)』
- 171 ページの『gmtime_r() ― 時間の変換 (再始動可能)』

- 192 ページの『localtime() ー 時間の変換』
- 194 ページの『localtime64() ー 時間の変換』
- 197 ページの『localtime64_r() ー 時間の変換 (再始動可能)』
- 195 ページの『localtime_r() ー 時間の変換 (再始動可能)』
- 228 ページの『mktime() ー 地方時の変換』
- 230 ページの『mktime64() ー 地方時の変換』
- 354 ページの『setlocale() ー ロケールの設定』
- 387 ページの『strftime() ー 日付/時刻からストリングへの変換』
- 429 ページの『time() ー 現在時刻の判別』
- 430 ページの『time64() ー 現在時刻の判別』
- 238 ページの『printf() ー 定様式の文字の出力』
- 19 ページの『<time.h>』

ctime64() ー 時間から文字ストリングへの変換

フォーマット

```
#include <time.h>
char *ctime64(const time64_t *time);
```

言語レベル: ILE C Extension

スレッド・セーフ: いいえ。代わりに `ctime64_r()` を使用します。

ロケール依存: この関数の振る舞いは、現行ロケールの LC_TOD カテゴリーの影響を受ける可能性があります。詳細については、 549 ページの『CCSID およびロケールの理解』を参照してください。

説明

`ctime64()` 関数は、*time* が指す時間の値を文字ストリング形式の現地時間に変換します。時間の値は通常、`time64()` 関数を呼び出して取得します。

`ctime64()` 関数が作成するストリングの結果にはちょうど 26 文字が含まれ、以下の形式となっています。

```
"%.3s %.3s%3d %.2d:%.2d:%.2d %d¥n"
```

以下に例を示します。

```
Mon Jul 16 02:03:55 1987\n\0
```

`ctime64()` 関数は 24 時間クロック形式を使用します。使用される月と日の省略形は、ロケールから取り出します。すべてのフィールドには定数幅があります。1 桁しかない日付は、ゼロがその前に置かれます。改行文字 (¥n) および NULL 文字 (¥0) がストリングの最後の位置を占めます。

戻り値

`ctime64()` 関数は文字ストリングの結果を指すポインターを戻します。関数が正常に実行されなかった場合、NULL を戻します。`ctime64()` 関数の呼び出しは次の式と同じです。

```
asctime(localtime64(&anytime))
```

注: `asctime()` 関数と `ctime64()` 関数、および他の時間関数は、戻りストリングの保持用に静的に割り振られた共通のバッファーを使用できます。これらの関数の 1 つを呼び出すごとに、以前の呼び出しの

結果が破棄される可能性があります。asctime_r()、ctime64_r()、gmtime64_r()、および localtime64_r() 関数は、戻りストリングの保持用に静的に割り振られた共通のバッファーを使用しません。再入可能性を希望する場合は、これらの関数を asctime() 関数、ctime64() 関数、gmtime64() 関数、および localtime64() 関数の代わりに使用することができます。

ctime64() の使用例

次の例では、time64() を使用してシステム・クロックをポーリングします。ポーリングの次に、現在の日時を示すメッセージを出力します。

```
#include <time.h>
#include <stdio.h>

int main(void)
{
   time64_t ltime;

   time64(&ltime);

   printf("the time is %s", ctime64(&ltime));
}
```

関連情報

- 41 ページの『asctime() ― 時間から文字ストリングへの変換』
- 43 ページの『asctime_r() ― 時間から文字ストリングへの変換 (再始動可能)』
- 75 ページの『ctime() ― 時間から文字ストリングへの変換』
- 77 ページの『ctime64() ― 時間から文字ストリングへの変換』
- 80 ページの『ctime64_r() ― 時間から文字ストリングへの変換 (再始動可能)』
- 167 ページの『gmtime() ― 時間の変換』
- 169 ページの『gmtime64() ― 時間の変換』
- 173 ページの『gmtime64_r() ― 時間の変換 (再始動可能)』
- 171 ページの『gmtime_r() ― 時間の変換 (再始動可能)』
- 192 ページの『localtime() ― 時間の変換』
- 194 ページの『localtime64() ― 時間の変換』
- 197 ページの『localtime64_r() ― 時間の変換 (再始動可能)』
- 195 ページの『localtime_r() ― 時間の変換 (再始動可能)』
- 228 ページの『mktime() ― 地方時の変換』
- 230 ページの『mktime64() ― 地方時の変換』
- 354 ページの『setlocale() ― ロケールの設定』
- 387 ページの『strftime() ― 日付/時刻からストリングへの変換』
- 429 ページの『time() ― 現在時刻の判別』
- 430 ページの『time64() ― 現在時刻の判別』
- 238 ページの『printf() ― 定様式の文字の出力』
- 19 ページの『<time.h>』

ctime_r() ― 時間から文字ストリングへの変換 (再始動可能)

フォーマット

```
#include <time.h>
char *ctime_r(const time_t *time, char *buf);
```

言語レベル: XPG4

スレッド・セーフ: はい。

ロケール依存: この関数の振る舞いは、現行ロケールの LC_TOD カテゴリーの影響を受ける可能性があります。詳細については、 549 ページの『CCSID およびロケールの理解』を参照してください。

説明

この関数は ctime() 関数の再始動可能バージョンです。

ctime_r() 関数は、*time* が指す時間の値を文字ストリング形式の現地時間に変換します。時間の値は通常、time() 関数を呼び出して取得します。

ctime_r() 関数が作成するストリングの結果にはちょうど 26 文字が含まれ、以下の形式になっています。

```
"%.3s %.3s%3d %.2d:%.2d:%.2d %d\n"
```

以下に例を示します。

```
Mon Jul 16 02:03:55 1987\n\0
```

ctime_r() 関数は 24 時間クロック形式を使用します。曜日は、Sun、Mon、Tue、Wed、Thu、Fri、および Sat に省略されます。月の省略形は、Jan、Feb、Mar、Apr、May、Jun、Jul、Aug、Sep、Oct、Nov、および Dec です。すべてのフィールドには固定幅があります。1 桁しかない日付は、ゼロがその前に置かれます。改行文字 (¥n) および NULL 文字 (¥0) がストリングの最後の位置を占めます。

戻り値

ctime_r() 関数は、文字ストリングの結果を指すポインターを戻します。関数が正常に実行されなかった場合、NULL を戻します。ctime_r() 関数の呼び出しは次の式と同じです。

```
asctime_r(localtime_r(&anytime, buf2), buf)
```

ここで、buf は文字を指すポインターです。

ctime_r() の使用例

次の例では、ctime_r() を使用してシステム・クロックをポーリングします。ポーリングの次に、現在の日時を示すメッセージを出力します。

```
#include <time.h>
#include <stdio.h>

int main(void)
{
   time_t ltime;
   char buf[50];

   time(&ltime);
 printf("the time is %s", ctime_r(&ltime, buf));
}
```

関連情報

- 41 ページの『asctime() ― 時間から文字ストリングへの変換』

- 43 ページの『asctime_r() — 時間から文字ストリングへの変換 (再始動可能)』
- 75 ページの『ctime() — 時間から文字ストリングへの変換』
- 77 ページの『ctime64() — 時間から文字ストリングへの変換』
- 『ctime64_r() — 時間から文字ストリングへの変換 (再始動可能)』
- 167 ページの『gmtime() — 時間の変換』
- 169 ページの『gmtime64() — 時間の変換』
- 173 ページの『gmtime64_r() — 時間の変換 (再始動可能)』
- 171 ページの『gmtime_r() — 時間の変換 (再始動可能)』
- 192 ページの『localtime() — 時間の変換』
- 194 ページの『localtime64() — 時間の変換』
- 197 ページの『localtime64_r() — 時間の変換 (再始動可能)』
- 195 ページの『localtime_r() — 時間の変換 (再始動可能)』
- 228 ページの『mktime() — 地方時の変換』
- 230 ページの『mktime64() — 地方時の変換』
- 387 ページの『strftime() — 日付/時刻からストリングへの変換』
- 429 ページの『time() — 現在時刻の判別』
- 430 ページの『time64() — 現在時刻の判別』
- 19 ページの『<time.h>』

ctime64_r() — 時間から文字ストリングへの変換 (再始動可能)

フォーマット

```
#include <time.h>
char *ctime64_r(const time64_t *time, char *buf);
```

言語レベル: ILE C Extension

スレッド・セーフ: はい。

ロケール依存: この関数の振る舞いは、現行ロケールの LC_TOD カテゴリーの影響を受ける可能性があります。詳細については、 549 ページの『CCSID およびロケールの理解』を参照してください。

説明

この関数は ctime64() 関数の再始動可能バージョンです。

ctime64() 関数は、*time* が指す時間の値を文字ストリング形式の現地時間に変換します。*time* の値は通常、time64() 関数を呼び出して取得します。

ctime64_r() 関数が作成するストリングの結果にはちょうど 26 文字が含まれ、以下の形式となっています。

```
"%.3s %.3s%3d %.2d:%.2d:%.2d %d\n"
```

以下に例を示します。

```
Mon Jul 16 02:03:55 1987\n\0
```

ctime64_r() 関数は 24 時間クロック形式を使用します。使用される月と日の省略形は、ロケールから取り出します。すべてのフィールドには定数幅があります。1 桁しかない日付は、ゼロがその前に置かれます。改行文字 (¥n) および NULL 文字 (¥0) がストリングの最後の位置を占めます。

戻り値

ctime64_r() 関数は文字ストリングの結果を指すポインターを戻します。関数が正常に実行されなかった場合、NULL を戻します。ctime64_r() 関数の呼び出しは次の式と同じです。

```
asctime_r(localtime64_r(&anytime, buf2), buf)
```

ctime64_r() の使用例

次の例では、time64() を使用してシステム・クロックをポーリングします。ポーリングの次にメッセージを出力し、現在の日時を示します。

```
#include <time.h>
#include <stdio.h>

int main(void)
{
   time64_t ltime;
   char buf[50];

   time64(&ltime);
 printf("the time is %s", ctime64_r(&ltime, buf));
}
```

関連情報

- 41 ページの『asctime() ― 時間から文字ストリングへの変換』
- 43 ページの『asctime_r() ― 時間から文字ストリングへの変換 (再始動可能)』
- 75 ページの『ctime() ― 時間から文字ストリングへの変換』
- 77 ページの『ctime64() ― 時間から文字ストリングへの変換』
- 78 ページの『ctime_r() ― 時間から文字ストリングへの変換 (再始動可能)』
- 167 ページの『gmtime() ― 時間の変換』
- 169 ページの『gmtime64() ― 時間の変換』
- 173 ページの『gmtime64_r() ― 時間の変換 (再始動可能)』
- 171 ページの『gmtime_r() ― 時間の変換 (再始動可能)』
- 192 ページの『localtime() ― 時間の変換』
- 194 ページの『localtime64() ― 時間の変換』
- 197 ページの『localtime64_r() ― 時間の変換 (再始動可能)』
- 195 ページの『localtime_r() ― 時間の変換 (再始動可能)』
- 228 ページの『mktime() ― 地方時の変換』
- 230 ページの『mktime64() ― 地方時の変換』
- 387 ページの『strftime() ― 日付/時刻からストリングへの変換』
- 429 ページの『time() ― 現在時刻の判別』
- 430 ページの『time64() ― 現在時刻の判別』
- 19 ページの『<time.h>』

_C_TS_malloc_debug() ― 使用されるテラスペース・メモリー量の判別 (オプションのダンプおよび検査を使用)

フォーマット

```
#include <mallocinfo.h>
int _C_TS_malloc_debug(unsigned int dump_level, unsigned int verify_level,
                       struct _C_mallinfo_t *output_record, size_t sizeofoutput);
```

言語レベル: Extended

スレッド・セーフ: はい。

説明

`_C_TS_malloc_debug()` 関数は、指定した output_record 構造体内で使用されるテラスペース・メモリーの量を判別し、その情報を戻します。指定した dump_level パラメーターが 0 よりも大きい場合は、使用される内部メモリー構造体も `stdout` にダンプします。指定した verify_level パラメーターが 0 よりも大きい場合は、内部メモリー構造体に対する検査確認も行います。検査が失敗すると、失敗を示すメッセージが `stdout` に生成されます。dump_level と verify_level の両方のパラメーターが 0 の場合、この関数は `_C_TS_malloc_info` 関数と同じ振る舞いをします。

次のマクロは、dump_level パラメーターに指定される <mallocinfo.h> インクルード・ファイル内で定義されます。

_C_NO_DUMPS	ダンプされる情報はありません
_C_DUMP_TOTALS	全体の合計および各チャンクの合計が出力されます
_C_DUMP_CHUNKS	各チャンクについての追加情報が出力されます
_C_DUMP_NODES	各チャンク内のすべてのノードの追加情報が出力されます
_C_DUMP_TREE	空きノードを追跡するために使用されるデカルト・ツリーの追加情報が出力されます
_C_DUMP_ALL	使用可能なすべての情報が出力されます

次のマクロは、verify_level パラメーターに指定される <mallocinfo.h> インクルード・ファイル内で定義されます。

_C_NO_CHECKS	検査確認は実行されません
_C_CHECK_TOTALS	合計が正確であるかどうかが検査されます
_C_CHECK_CHUNKS	各チャンクについて追加検査が実行されます
_C_CHECK_NODES	各チャンク内のすべてのノードについて追加検査が実行されます。
_C_CHECK_TREE	空きノードを追跡するために使用されるデカルト・ツリーの追加検査が実行されます。
_C_CHECK_ALL	すべての検査が実行されます
_C_CHECK_ALL_AND_ABORT	すべての検査が実行され、いずれかの検査が失敗すると、`abort()` 関数が呼び出されます。

注: この関数は、アプリケーション内のテラスペース・メモリー使用量の詳細デバッグ用です。

戻り値

正常に実行された場合、この関数は 0 を戻します。エラーが発生した場合、関数は負の値を戻します。

_C_TS_malloc_debug() の使用例

この例は、_C_TS_malloc_debug() から戻された情報を stdout に出力します。このプログラムは TERASPACE(*YES *TSIFC) でコンパイルされます。

```
#include <stdio.h>
#include <stdlib.h>
#include <mallocinfo.h>

int main (void)
{
   _C_mallinfo_t info;
   int           rc;
   void         *m;

   /* Allocate a small chunk of memory */
   m = malloc(500);

   rc = _C_TS_malloc_debug(_C_DUMP_TOTALS,
                           _C_NO_CHECKS,
                           &info, sizeof(info));

   if (rc == 0) {
      Printf("_C_TS_malloc_debug successful¥n");
   }
   else {
      printf("_C_TS_malloc_debug failed (rc = %d)¥n", rc);
   }

   free(m);
}

/*****************************************************
  The output should be similar to:

     total_bytes        = 524288
     allocated_bytes    = 688
     unallocated_bytes  = 523600
     allocated_blocks   = 1
     unallocated_blocks = 1
     requested_bytes    = 500
     pad_bytes          = 12
     overhead_bytes     = 176
   Number of memory chunks  = 1
   Total bytes              = 524288
   Total allocated bytes    = 688
   Total unallocated bytes  = 523600
   Total allocated blocks   = 1
   Total unallocated blocks = 1
   Total requested bytes    = 500
   Total pad bytes          = 12
   Total overhead bytes     = 176
   _C_TS_malloc_debug successful
 *****************************************************
```

関連情報

- 84 ページの『_C_TS_malloc_info() ― 使用されるテラスペース・メモリー量の判別』
- 58 ページの『calloc() ― ストレージの予約と初期化』
- 134 ページの『free() ― ストレージ・ブロックの解放』
- 203 ページの『malloc() ― ストレージ・ブロックの予約』

- 276 ページの『realloc() ― 予約ストレージ・ブロック・サイズの変更』
- 9 ページの『<mallocinfo.h>』
- 564 ページの『ヒープ・メモリー』

_C_TS_malloc_info() ― 使用されるテラスペース・メモリー量の判別

フォーマット

```
#include <mallocinfo.h>
int _C_TS_malloc_info(struct _C_mallinfo_t *output_record, size_t sizeofoutput);
```

言語レベル: Extended

スレッド・セーフ: はい。

説明

_C_TS_malloc_info() 関数は、指定した output_record 構造体内で使用されるテラスペース・メモリーの量を判別し、その情報を戻します。

注: この関数は、アプリケーション内のテラスペース・メモリー使用量の詳細デバッグ用です。

戻り値

正常に実行された場合、この関数は 0 を戻します。エラーが発生した場合、関数は負の値を戻します。

_C_TS_malloc_info() の使用例

この例は、_C_TS_malloc_info() から戻された情報を stdout に出力します。このプログラムは TERASPACE(*YES *TSIFC) でコンパイルされます。

```
#include <stdio.h>
#include <stdlib.h>
#include <mallocinfo.h>

int main (void)
{
   _C_mallinfo_t info;
   int           rc;
   void         *m;

   /* Allocate a small chunk of memory */
   m = malloc(500);

   rc = _C_TS_malloc_info(&info, sizeof(info));

   if (rc == 0) {
      printf("Total bytes              = %llu¥n",
             info.total_bytes);
      printf("Total allocated bytes    = %llu¥n",
             info.allocated_bytes);
      printf("Total unallocated bytes  = %llu¥n",
             info.unallocated_bytes);
      printf("Total allocated blocks   = %llu¥n",
             info.allocated_blocks);
      printf("Total unallocated blocks = %llu¥n",
             info.unallocated_blocks);
      printf("Total requested bytes    = %llu¥n",
             info.requested_bytes);
      printf("Total pad bytes          = %llu¥n",
             info.pad_bytes);
      printf("Total overhead bytes     = %llu¥n",
             info.overhead_bytes);
   }
   else {
      printf("_C_TS_malloc_info failed (rc = %d)¥n", rc);
   }

   free(m);
}

/****************************************************
  The output should be similar to:

   Total bytes              = 524288
   Total allocated bytes    = 688
   Total unallocated bytes  = 523600
   Total allocated blocks   = 1
   Total unallocated blocks = 1
   Total requested bytes    = 500
   Total pad bytes          = 12
   Total overhead bytes     = 176
 ****************************************************
```

関連情報

- 82 ページの『_C_TS_malloc_debug() — 使用されるテラスペース・メモリー量の判別 (オプションのダンプおよび検査を使用)』
- 58 ページの『calloc() — ストレージの予約と初期化』
- 134 ページの『free() — ストレージ・ブロックの解放』
- 203 ページの『malloc() — ストレージ・ブロックの予約』
- 276 ページの『realloc() — 予約ストレージ・ブロック・サイズの変更』
- 9 ページの『<mallocinfo.h>』

| • 564 ページの『ヒープ・メモリー』

difftime() ― 時差の計算

フォーマット

```
#include <time.h>
double difftime(time_t time2, time_t time1);
```

言語レベル: ANSI

スレッド・セーフ: はい。

説明

`difftime()` 関数は *time2* と *time1* の差 (秒数) を計算します。

戻り値

`difftime()` 関数は、*time1* から *time2* への経過時間 (秒数) を倍精度の数値として戻します。型 time_t は <time.h> で定義されます。

`difftime()` の使用例

次の例は `difftime()` を使用するタイミング・アプリケーションを示しています。2 から 10 000 までの素数を検索するのにかかる平均時間を計算します。

```
#include <time.h>
#include <stdio.h>

#define RUNS 1000
#define SIZE 10000

int mark[SIZE];

int main(void)
{
   time_t start, finish;
   int i, loop, n, num;

   time(&start);

   /*  This loop finds the prime numbers between 2 and SIZE   */
   for (loop = 0; loop < RUNS; ++loop)
      {
      for (n = 0; n < SIZE; ++n)
         mark [n] = 0;
      /*  This loops marks all the composite numbers with -1  */
      for (num = 0, n = 2; n < SIZE; ++n)
         if ( ! mark[n])
         {
            for (i = 2 * n; i < SIZE; i += n)
                mark[i] = -1;
            ++num;
            }
      }
   time(&finish);
   printf("Program takes an average of %f seconds "
                  "to find %d primes.¥n",
                   difftime(finish,start)/RUNS, num);
}

/********************  Output should be similar:  *****************

The program takes an average of 0.106000 seconds to find 1229 primes.
*/
```

関連情報

- 41 ページの『asctime() ― 時間から文字ストリングへの変換』
- 43 ページの『asctime_r() ― 時間から文字ストリングへの変換 (再始動可能)』
- 75 ページの『ctime() ― 時間から文字ストリングへの変換』
- 77 ページの『ctime64() ― 時間から文字ストリングへの変換』
- 80 ページの『ctime64_r() ― 時間から文字ストリングへの変換 (再始動可能)』
- 78 ページの『ctime_r() ― 時間から文字ストリングへの変換 (再始動可能)』
- 88 ページの『difftime64() ― 時差の計算』
- 167 ページの『gmtime() ― 時間の変換』
- 169 ページの『gmtime64() ― 時間の変換』
- 173 ページの『gmtime64_r() ― 時間の変換 (再始動可能)』
- 171 ページの『gmtime_r() ― 時間の変換 (再始動可能)』
- 192 ページの『localtime() ― 時間の変換』
- 194 ページの『localtime64() ― 時間の変換』
- 197 ページの『localtime64_r() ― 時間の変換 (再始動可能)』
- 195 ページの『localtime_r() ― 時間の変換 (再始動可能)』

- 228 ページの『mktime() ― 地方時の変換』
- 230 ページの『mktime64() ― 地方時の変換』
- 387 ページの『strftime() ― 日付/時刻からストリングへの変換』
- 429 ページの『time() ― 現在時刻の判別』
- 430 ページの『time64() ― 現在時刻の判別』
- 19 ページの『<time.h>』

difftime64() ― 時差の計算

フォーマット

```
#include <time.h>
double difftime64(time64_t time2, time64_t time1);
```

言語レベル: ILE C Extension

スレッド・セーフ: はい。

説明

`difftime64()` 関数は *time2* と *time1* の差 (秒数) を計算します。

戻り値

`difftime64()` 関数は、*time1* から *time2* への経過時間 (秒数) を倍精度の数値として戻します。型 `time64_t` は <time.h> で定義されます。

`difftime64()` の使用例

次の例は `difftime64()` を使用するタイミング・アプリケーションを示しています。 2 から 10 000 までの素数を検索するのにかかる平均時間を計算します。

```
#include <time.h>
#include <stdio.h>

#define RUNS 1000
#define SIZE 10000

int mark[SIZE];

int main(void)
{
   time64_t start, finish;
   int i, loop, n, num;

   time64(&start);

   /*  This loop finds the prime numbers between 2 and SIZE   */
   for (loop = 0; loop < RUNS; ++loop)
      {
      for (n = 0; n < SIZE; ++n)
         mark [n] = 0;
      /*  This loops marks all the composite numbers with -1  */
      for (num = 0, n = 2; n < SIZE; ++n)
         if ( ! mark[n])
         {
            for (i = 2 * n; i < SIZE; i += n)
                mark[i] = -1;
            ++num;
            }
      }
   time64(&finish);
   printf("Program takes an average of %f seconds "
                 "to find %d primes.¥n",
                  difftime64(finish,start)/RUNS, num);
}

/********************  Output should be similar:  *****************

The program takes an average of 0.106000 seconds to find 1229 primes.
*/
```

関連情報

- 41 ページの『asctime() — 時間から文字ストリングへの変換』
- 43 ページの『asctime_r() — 時間から文字ストリングへの変換 (再始動可能)』
- 75 ページの『ctime() — 時間から文字ストリングへの変換』
- 77 ページの『ctime64() — 時間から文字ストリングへの変換』
- 80 ページの『ctime64_r() — 時間から文字ストリングへの変換 (再始動可能)』
- 78 ページの『ctime_r() — 時間から文字ストリングへの変換 (再始動可能)』
- 86 ページの『difftime() — 時差の計算』
- 167 ページの『gmtime() — 時間の変換』
- 169 ページの『gmtime64() — 時間の変換』
- 173 ページの『gmtime64_r() — 時間の変換 (再始動可能)』
- 171 ページの『gmtime_r() — 時間の変換 (再始動可能)』
- 192 ページの『localtime() — 時間の変換』
- 194 ページの『localtime64() — 時間の変換』
- 197 ページの『localtime64_r() — 時間の変換 (再始動可能)』
- 195 ページの『localtime_r() — 時間の変換 (再始動可能)』

- 228 ページの『mktime() ― 地方時の変換』
- 230 ページの『mktime64() ― 地方時の変換』
- 387 ページの『strftime() ― 日付/時刻からストリングへの変換』
- 429 ページの『time() ― 現在時刻の判別』
- 430 ページの『time64() ― 現在時刻の判別』
- 19 ページの『<time.h>』

div() ― 商および剰余の計算

フォーマット

```
#include <stdlib.h>
div_t div(int numerator, int denominator);
```

言語レベル: ANSI

スレッド・セーフ: はい。ただし、関数バージョンのみがスレッド・セーフです。マクロ・バージョンはスレッド・セーフではありません。

説明

div() 関数は、*numerator* を *denominator* で割った商および剰余を計算します。

戻り値

div() 関数は、商 int quot および剰余 int rem の両方を含む型 div_t の構造体を戻します。戻り値が表せない場合は、その値は予期できません。 *denominator* が 0 の場合は、例外が起こります。

div() の使用例

この例では、div() を使用して、2 つの被除数と 2 つの除数からなる 1 組のセットの商と剰余を計算します。

```
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
   int num[2] = {45,-45};
   int den[2] = {7,-7};
   div_t ans;   /* div_t is a struct type containing two ints:
                   'quot' stores quotient; 'rem' stores remainder */
   short i,j;

   printf("Results of division:¥n");
   for (i = 0; i < 2; i++)
      for (j = 0; j < 2; j++)
      {
         ans = div(num[i],den[j]);
         printf("Dividend: %6d  Divisor: %6d", num[i], den[j]);
         printf("  Quotient: %6d  Remainder: %6d¥n", ans.quot, ans.rem);
      }
}

/*****************  Output should be similar to:  *****************

Results of division:
Dividend:  45  Divisor:   7  Quotient:   6  Remainder:   3
Dividend:  45  Divisor:  -7  Quotient:  -6  Remainder:   3
Dividend: -45  Divisor:   7  Quotient:  -6  Remainder:  -3
Dividend: -45  Divisor:  -7  Quotient:   6  Remainder:  -3

**********************************************************/
```

関連情報

- 186 ページの『ldiv() — lldiv() — long 型整数および long long 型整数の除算の実行』
- 18 ページの『<stdlib.h>』

erf() - erfc() — 誤差関数の計算

フォーマット

```
#include <math.h>
double erf(double x);
double erfc(double x);
```

言語レベル: ILE C Extension

スレッド・セーフ: はい。

説明

`erf()` 関数は、以下の誤差関数を計算します。

$$2\pi^{-1/2}\int_{0}^{x} e^{-t2}dt$$

`erfc()` 関数は 1.0 - erf(x) の値を計算します。 x の値が大きい場合は、`erfc()` 関数を `erf()` の代わりに使用します。

戻り値

erf() 関数は、誤差関数を表す double 値を戻します。 erfc() 関数は、1.0 - erf を表す double 値を戻します。

erf() の使用例

次の例は erf() と erfc() を使用して 2 つの数値の誤差関数を計算します。

```
#include <stdio.h>
#include <math.h>

double smallx, largex, value;

int main(void)
{
   smallx = 0.1;
   largex = 10.0;

   value = erf(smallx);         /* value = 0.112463 */
   printf("Error value for 0.1: %lf¥n", value);

   value = erfc(largex);        /* value = 2.088488e-45 */
   printf("Error value for 10.0: %le¥n", value);
}

/*****************  Output should be similar to:  *****************

Error value for 0.1: 0.112463
Error value for 10.0: 2.088488e-45
*/
```

関連情報

- 53 ページの『ベッセル関数』
- 156 ページの『gamma() ― ガンマ関数』
- 9 ページの『<math.h>』

exit() ― プログラムの終了

フォーマット

```
#include <stdlib.h>
void exit(int status);
```

言語レベル: ANSI

スレッド・セーフ: はい。

説明

exit() 関数は、プログラムからホスト環境に制御を戻します。まず、atexit() 関数によって登録されたすべての関数を、逆の順番で呼び出します。つまり、最後に登録された関数を最初に呼び出します。プログラムを終了する前に、すべてのバッファーを削除し、すべてのオープン・ファイルをクローズします。

引数 *status* は 0 以上 255 以下の値か、またはマクロ EXIT_SUCCESS または EXIT_FAILURE のいずれかの値をとることができます。 *status* の値が、EXIT_SUCCESS または 0 の場合、通常の終了を示します。それ以外の場合は、別の status 値が戻されます。

注: SYSIFCOPT(*ASYNCSIGNAL) でコンパイルされた場合、`exit()` はシグナル・ハンドラー内で呼び出すことはできません。

戻り値

`exit()` 関数は制御と *status* の値の両方をオペレーティング・システムに戻します。

`exit()` の使用例

次の例は、ファイル myfile をオープンできない場合に、バッファーを削除し、すべてのオープン・ファイルをクローズした後でプログラムを終了します。

```
#include <stdio.h>
#include <stdlib.h>

FILE *stream;

int main(void)
{
   if ((stream = fopen("mylib/myfile", "r")) == NULL)
   {
      perror("Could not open data file");
      exit(EXIT_FAILURE);
   }
}
```

関連情報

- 38 ページの『abort() ― プログラムの停止』
- 48 ページの『atexit() ― プログラム終了の記録関数』
- 362 ページの『signal() ― 割り込みシグナルの処理』
- 18 ページの『<stdlib.h>』

exp() ― 指数関数の計算

フォーマット

```
#include <math.h>
double exp(double x);
```

言語レベル: ANSI

スレッド・セーフ: はい。

説明

`exp()` 関数は浮動小数点引数 *x* (e^x、ここで *e* は 2.17128128... と等しい) の指数値を計算します。

戻り値

オーバーフローが起きた場合、`exp()` 関数は HUGE_VAL を戻します。アンダーフローが起きた場合は 0 を戻します。オーバーフローの場合もアンダーフローの場合も errno は ERANGE に設定されます。errno の値も EDOM に設定される可能性があります。

`exp()` の使用例

次の例は *y* を *x* の指数関数として計算します。

```
#include <math.h>
#include <stdio.h>

int main(void)
{
   double x, y;

   x = 5.0;
   y = exp(x);

   printf("exp( %lf ) = %lf¥n", x, y);
}

/*****************  Output should be similar to:  *****************

exp( 5.000000 ) = 148.413159
*/
```

関連情報

- 198 ページの『log() ― 自然対数の計算』
- 199 ページの『log10() ― 基数 10 の対数の計算』
- 9 ページの『<math.h>』

fabs() ― 浮動小数点絶対値の計算

フォーマット

```
#include <math.h>
double fabs(double x);
```

言語レベル: ANSI

スレッド・セーフ: はい。

説明

`fabs()` 関数は浮動小数点引数 *x* の絶対値を計算します。

戻り値

`fabs()` 関数は絶対値を戻します。エラーの戻り値はありません。

`fabs()` の使用例

次の例は *y* を *x* の絶対値として計算します。

```
#include <math.h>
#include <stdio.h>

int main(void)
{
   double x, y;

   x = -5.6798;
   y = fabs(x);

   printf("fabs( %lf ) = %lf¥n", x, y);
}

/*******************  Output should be similar to:  ***************

fabs( -5.679800 ) = 5.679800
*/
```

関連情報

- 39 ページの『abs() ― 整数の絶対値の計算』
- 184 ページの『labs() ― llabs() ― long 型および long long 型整数の絶対値の計算』
- 9 ページの『<math.h>』

fclose() ― ストリームのクローズ

フォーマット

```
#include <stdio.h>
int fclose(FILE *stream);
```

言語レベル: ANSI

スレッド・セーフ: はい。

説明

`fclose()` 関数は *stream* で示されるストリームをクローズします。この関数は、ストリームをクローズする前に、そのストリームと関連付けられたすべてのバッファーを削除します。ストリームをクローズする場合、この関数はシステムが予約したすべてのバッファーを開放します。バイナリー・ストリームがクローズされる場合は、ファイル内の最後のレコードがそのレコードの終わりまでヌル文字 (¥0) で埋められます。

戻り値

ストリームが正常にクローズされた場合は、`fclose()` 関数は 0 を戻し、エラーが検出された場合は EOF を戻します。

errno の値は、次のいずれかに設定されます。

値　　意味

ENOTOPEN
ファイルはオープンされていません。

EIOERROR
リカバリー不能な入出力エラーが発生しました。

EIORECERR
リカバリー可能な入出力エラーが発生しました。

ESCANFAILURE
ファイルは走査失敗のマークを付けられました。

注: FILE ポインターが指すストレージは、fclose() 関数によって解放されます。fclose() 関数を使用した後は、FILE ポインターを使用するすべての試行が無効です。

fclose() の使用例

次の例は、ファイル myfile をオープンしてストリームとして読み込み、その後にこのファイルをクローズします。

```
#include <stdio.h>

#define NUM_ALPHA  26

int main(void)
{
  FILE *stream;
  char buffer[NUM_ALPHA];

  if (( stream = fopen("mylib/myfile", "r"))!= NULL )
  {
    fread( buffer, sizeof( char ), NUM_ALPHA, stream );
    printf( "buffer = %s¥n", buffer );
  }

  if (fclose(stream))   /* Close the stream. */
      perror("fclose error");
  else printf("File mylib/myfile closed successfully.¥n");
}
```

関連情報

- 101 ページの『fflush() ― ファイルへのバッファーの書き込み』
- 114 ページの『fopen() ― ファイルのオープン』
- 136 ページの『freopen() ― オープン・ファイルのリダイレクト』
- 17 ページの『<stdio.h>』

fdopen() ― ストリームとファイル記述子との関連付け

フォーマット

```
#include <stdio.h>
FILE *fdopen(int handle, char *type);
```

言語レベル: XPG4

スレッド・セーフ: はい。

統合ファイル・システム・インターフェース: この関数は、コンパイル・コマンドに対して SYSIFCOPT(*NOIFSIO) が指定されている場合には使用できません。

説明

fdopen() 関数は、入力または出力ストリームを、*handle* により識別されるファイルと関連付けます。*type* 変数は、ストリームに要求されるアクセス・タイプを指定する文字ストリングです。この変数には、位置パラメーターが 1 つ含まれ、オプションのキーワード・パラメーターが後に続きます。

位置パラメーターには次の値が可能です。

モード 説明

r
: テキスト・ファイルを読み込むためのストリームを作成します。ファイル・ポインターはファイルの先頭に設定されます。

w
: テキスト・ファイルに書き込むためのストリームを作成します。ファイル・ポインターはファイルの先頭に設定されます。

a
: テキスト・ファイルの最後に追加モードで書き込むためのストリームを作成します。ファイル・ポインターはファイルの最後に設定されます。

r+
: テキスト・ファイルを読み書きするためのストリームを作成します。ファイル・ポインターはファイルの先頭に設定されます。

w+
: テキスト・ファイルを読み書きするためのストリームを作成します。ファイル・ポインターはファイルの先頭に設定されます。

a+
: テキスト・ファイルの最後に追加モードで読み書きするためのストリームを作成します。ファイル・ポインターはファイルの最後に設定されます。

rb
: バイナリー・ファイルを読み込むためのストリームを作成します。ファイル・ポインターはファイルの先頭に設定されます。

wb
: バイナリー・ファイルに書き込むためのストリームを作成します。ファイル・ポインターはファイルの先頭に設定されます。

ab
: 追加モードでバイナリー・ファイルに書き込むためのストリームを作成します。ファイル・ポインターはファイルの最後に設定されます。

r+b *または* **rb+**
: バイナリー・ファイルを読み書きするためのストリームを作成します。ファイル・ポインターはファイルの先頭に設定されます。

w+b *または* **wb+**
: バイナリー・ファイルを読み書きするためのストリームを作成します。ファイル・ポインターはファイルの先頭に設定されます。

a+b *または* **ab+**
: 追加モードでバイナリー・ファイルを読み書きするためのストリームを作成します。ファイル・ポインターはファイルの最後に設定されます。

注: w, w+, wb, wb+, および w+b モードを使用する場合は注意してください。既存のファイルを破棄することがあります。

指定する *type* は、ファイルをオープンするために使用したアクセス方式と互換性がなければなりません。ファイルを O_APPEND フラグでオープンした場合は、ストリーム・モードは a, a+, ab, a+b, または ab+ でなければなりません。`fdopen()` 関数を使用するには、ファイル記述子が必要です。記述子を取得するには、POSIX 関数 `open()` を使用します。 O_APPEND フラグは `open()` 用のモードです。`open()` 用のモードは QSYSINC/H/FCNTL で定義されます。詳しい情報については、i5/OS Information Center の『API』のトピックを参照してください。

`fdopen()` に使用できるキーワード・パラメーターは、114 ページの『fopen() ― ファイルのオープン』 に説明されている、統合ファイル・システム用のキーワード・パラメーターと同じです。

`fdopen()` が NULL を戻した場合は、`close()` を使用してファイルをクローズします。 `fdopen()` が正常に実行された場合は、`fclose()` を使用してストリームとファイルをクローズする必要があります。

戻り値

fdopen() 関数は、オープン・ファイルにアクセスするために使用できるファイル構造体を指すポインターを戻します。 NULL ポインターの戻り値は、エラーを示します。

fdopen() の使用例

次の例はファイル sample.dat をオープンし、fdopen() を使用してストリームをファイルと関連付けます。次に、ストリームからバッファーに読み込みます。

```
/* compile with SYSIFCOPT(*IFSIO) */
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <string.h>

int main(void)
{
   long length;
   int fh;
   char buffer[20];
   FILE *fp;

   printf("¥nCreating sample.dat.¥n");
   if ((fp= fopen("/sample.dat", "w")) == NULL) {
       perror(" File was not created: ");
       exit(1);
   }
   fputs("Sample Program", fp);
   fclose(fp);

   memset(buffer, '¥0', 20);                              /* Initialize buffer*/

   if (-1 == (fh = open("/sample.dat", O_RDWR|O_APPEND))) {
      perror("Unable to open sample.dat");
      exit(1);
   }
   if (NULL == (fp = fdopen(fh, "r"))) {
      perror("fdopen failed");
      close(fh);
      exit(1);
   }
   if (14 != fread(buffer, 1, 14, fp)) {
      perror("fread failed");
      fclose(fp);
      exit(1);
   }
   printf("Successfully read from the stream the following:¥n%s.¥n", buffer);
   fclose(fp);
   return 1;

   /****************************************************************
    * The output should be:
    *
    * Creating sample.dat.
    * Successfully read from the stream the following:
    * Sample Program.
    */
}
```

関連情報

- 95 ページの『fclose() ― ストリームのクローズ』
- 114 ページの『fopen() ― ファイルのオープン』

- 140 ページの『fseek() ― fseeko() ― ファイル位置の位置変更』
- 142 ページの『fsetpos() ― ファイル位置の設定』
- 288 ページの『rewind() ― 現在のファイル位置の調整』
- 17 ページの『<stdio.h>』
- i5/OS Information Center の『API』トピックにある open API。
- i5/OS Information Center の『API』トピックにある close API。

feof() ― ファイル終了標識のテスト

フォーマット

```
#include <stdio.h>
int feof(FILE *stream);
```

言語レベル: ANSI

スレッド・セーフ: はい。

説明

`feof()` 関数は、指定された *stream* に対してファイル終了フラグを設定するかどうかを示します。ファイル終了フラグは複数の関数によって設定され、ファイル終了を示します。ファイル終了フラグは、このストリームに対して `rewind()` 関数、`fsetpos()` 関数、`fseek()` 関数、または `clearerr()` 関数を呼び出すことにより、クリアされます。

戻り値

`feof()` 関数は、EOF フラグが設定された場合にのみ、ゼロ以外の値を戻し、それ以外の場合は 0 を戻します。

`feof()` の使用例

次の例は、ファイル終了文字を読み込むまで、入力 stream を走査します。

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
   char string[100];
   FILE *stream;
   memset(string, 0, sizeof(string));
   stream = fopen("qcpple/qacsrc(feof)", "r");

   fscanf(stream, "%s", string);
   while (!feof(stream))
   {
      printf("%s¥n", string);
      memset(string, 0, sizeof(string));
      fscanf(stream, "%s", string);
   }
}
```

関連情報

- 65 ページの『clearerr() ― エラー標識のリセット』

- 『ferror() ― 読み取り/書き込みエラーのテスト』
- 140 ページの『fseek() ― fseeko() ― ファイル位置の位置変更』
- 142 ページの『fsetpos() ― ファイル位置の設定』
- 236 ページの『perror() ― エラー・メッセージの出力』
- 288 ページの『rewind() ― 現在のファイル位置の調整』
- 17 ページの『<stdio.h>』

ferror() ― 読み取り/書き込みエラーのテスト

フォーマット

```
#include <stdio.h>
int ferror(FILE *stream);
```

言語レベル: ANSI

スレッド・セーフ: はい。

説明

`ferror()` 関数は指定された *stream* からの読み取り中、または書き込み中のエラーをテストします。エラーが発生した場合、*stream* をクローズするか、`rewind()` 関数を呼び出すか、または `clearerr()` 関数を呼び出すまでは*stream* のエラー標識は設定されたままになります。

戻り値

`ferror()` 関数はゼロ以外の値を戻して、指定した *stream* のエラーを示します。0 の戻り値は、エラーが発生しなかったことを意味します。

ferror() の使用例

次の例は、データをストリームにプットしてから、書き込みエラーが発生しなかったかどうかを確認します。

```
#include <stdio.h>

int main(void)
{
   FILE *stream;
   char *string = "Important information";
   stream = fopen("mylib/myfile","w");

   fprintf(stream, "%s¥n", string);
   if (ferror(stream))
   {
      printf("write error¥n");
      clearerr(stream);
   }
   if (fclose(stream))
      perror("fclose error");
}
```

関連情報

- 65 ページの『clearerr() ― エラー標識のリセット』
- 99 ページの『feof() ― ファイル終了標識のテスト』

- 114 ページの『fopen() ― ファイルのオープン』
- 236 ページの『perror() ― エラー・メッセージの出力』
- 384 ページの『strerror() ― ランタイム・エラー・メッセージを指すポインターの設定』
- 17 ページの『<stdio.h>』

fflush() ― ファイルへのバッファーの書き込み

フォーマット

```
#include <stdio.h>
int fflush(FILE *stream);
```

言語レベル: ANSI

スレッド・セーフ: はい。

説明

`fflush()` 関数は、可能であれば指定された出力 *stream* に関連付けられたバッファーを空にするようにシステムに命じます。*stream* が入力に対してオープンである場合、`fflush()` 関数はすべての `ungetc()` 関数の効果を取り消します。呼び出しの後、*stream* はオープンしたままになります。

stream が NULL の場合は、システムはすべてのオープン・ストリームをフラッシュします。

注: ユーザーがストリームをクローズするとき、またはプログラムがストリームをクローズせずに正常終了するときには、システムはバッファーを自動的に削除します。

戻り値

`fflush()` 関数は、バッファーを正常に削除した場合は値 0 を戻します。エラーが発生した場合は EOF を戻します。

errno の値は、次のいずれかに設定されます。

値　　意味

ENOTOPEN
ファイルはオープンされていません。

ERECIO
ファイルはレコードの入出力用にオープンされています。

ESTDIN
`stdin` をオープンできません。

EIOERROR
リカバリー不能な入出力エラーが発生しました。

EIORECERR
リカバリー可能な入出力エラーが発生しました。

`fflush()` 関数は、type=record でオープンされたファイルではサポートされていません。

`fflush()` の使用例

次の例はストリーム・バッファーを削除します。

```
#include <stdio.h>

int main(void)
{
   FILE *stream;
   int ch;
   unsigned int result = 0;

   stream = fopen("mylib/myfile", "r");
   while ((ch = getc(stream)) != EOF && isdigit(ch))
      result = result * 10 + ch - '0';
   if (ch != EOF)
      ungetc(ch,stream);

   fflush(stream);           /* fflush undoes the effect of ungetc function
*/
   printf("The result is: %d¥n", result);
   if ((ch = getc(stream)) != EOF)
      printf("The character is: %c¥n", ch);
}
```

関連情報

- 95 ページの『fclose() ― ストリームのクローズ』
- 114 ページの『fopen() ― ファイルのオープン』
- 351 ページの『setbuf() ― バッファリングの制御』
- 439 ページの『ungetc() ― 入力ストリームへの文字のプッシュ』
- 17 ページの『<stdio.h>』

fgetc() ― 文字の読み取り

フォーマット

```
#include <stdio.h>
int fgetc(FILE *stream);
```

言語レベル: ANSI

スレッド・セーフ: はい。

説明

`fgetc()` 関数は 1 バイトの符号なし文字を、現在位置で入力 *stream* から読み取り、関連付けられたファイル・ポインターがある場合はこれを増やして、次の文字を指すようにします。

注: `fgetc()` 関数は `getc()` と同じですが、常に関数呼び出しとして定義され、マクロで置き換えられることはありません。

戻り値

`fgetc()` 関数は、整数として読み取られる文字を戻します。 EOF の戻り値はエラーか、またはファイル終了状態を示します。 EOF の値がエラーを示しているか、あるいはファイル終了を示しているかを判別するには、`feof()` 関数または `ferror()` 関数を使用してください。

errno の値は、次のいずれかに設定されます。

値 **意味**

EBADF
: ファイル・ポインター、または記述子が有効ではありません。

ECONVERT
: 変換エラーが発生しました。

ENOTREAD
: ファイルは読み取り操作用にオープンされていません。

EGETANDPUT
: 書き込み操作の後に許可されていない読み取り操作が発生しました。

ERECIO
: ファイルはレコードの入出力用にオープンしています。

ESTDIN
: `stdin` をオープンできません。

EIOERROR
: リカバリー不能な入出力エラーが発生しました。

EIORECERR
: リカバリー可能な入出力エラーが発生しました。

`fgetc()` 関数は、type=record でオープンされたファイルではサポートされていません。

fgetc() の使用例

次の例は、入力データ行をストリームから取得します。

```
#include <stdio.h>

#define  MAX_LEN  80

int main(void)
{
   FILE *stream;
   char buffer[MAX_LEN + 1];
   int i, ch;

   stream = fopen("mylib/myfile","r");

   for (i = 0; (i < (sizeof(buffer)-1) &&
         ((ch = fgetc(stream)) != EOF) && (ch != '¥n')); i++)
      buffer[i] = ch;

   buffer[i] = '¥0';

   if (fclose(stream))
      perror("fclose error");

   printf("line: %s¥n", buffer);
}

 /***********************************************************************
      If FILENAME contains:  one two three
      The output should be:
      line: one two three
   ***********************************************************************/
```

関連情報

- 99 ページの『feof() ― ファイル終了標識のテスト』
- 100 ページの『ferror() ― 読み取り/書き込みエラーのテスト』
- 107 ページの『fgetwc() ― ストリームのワイド文字の読み取り』
- 124 ページの『fputc() ― 文字の書き込み』
- 158 ページの『getc() - getchar() ― 文字の読み取り』
- 163 ページの『getwc() ― ストリームからのワイド文字の読み取り』
- 165 ページの『getwchar() ― STDIN からのワイド文字の取得』
- 17 ページの『<stdio.h>』

fgetpos() ― ファイル位置の取得

フォーマット

```
#include <stdio.h>
int fgetpos(FILE *stream, fpos_t *pos);
```

言語レベル: ANSI

スレッド・セーフ: はい

説明

`fgetpos()` 関数は、*stream* に関連するファイル・ポインターの現行の位置を、*pos* が指すオブジェクトに格納します。 *pos* が指す値を後で `fsetpos()` の呼び出しで使用して、*stream* の位置を変更することができます。

戻り値

`fgetpos()` 関数は、正常に終了した場合は 0 を戻し、エラーになるとゼロ以外を戻して errno をゼロ以外の値に設定します。

errno の値は、次のいずれかに設定されます。

値　　意味

EBADF
ファイル・ポインター、または記述子が有効ではありません。

EBADSEEK
シーク操作のオフセットが無効です。

ENODEV
誤ったデバイスに対して操作が試行されました。

ENOTOPEN
ファイルはオープンされていません。

ERECIO
ファイルはレコードの入出力用にオープンしています。

ESTDERR
`stderr` をオープンできません。

ESTDIN
`stdin` をオープンできません。

ESTDOUT
  stdout をオープンできません。

EIOERROR
  リカバリー不能な入出力エラーが発生しました。

EIORECERR
  リカバリー可能な入出力エラーが発生しました。

fgetpos() 関数は、type=record でオープンされたファイルではサポートされていません。

fgetpos() の使用例

次の例はファイル myfile をオープンして読み取り、現在のファイル・ポインターの位置を変数 *pos* に格納します。

```
#include <stdio.h>

FILE *stream;

int main(void)
{
   int retcode;
   fpos_t pos;

   stream = fopen("mylib/myfile", "rb");

   /* The value returned by fgetpos can be used by fsetpos */
   /* to set the file pointer if 'retcode' is 0            */

   if ((retcode = fgetpos(stream, Point-of-Sale)) == 0)
      printf("Current position of file pointer found¥n");
   fclose(stream);
}
```

関連情報

- 140 ページの『fseek() ― fseeko() ― ファイル位置の位置変更』
- 142 ページの『fsetpos() ― ファイル位置の設定』
- 144 ページの『ftell() ― ftello() ― 現在位置の取得』
- 17 ページの『<stdio.h>』

fgets() ― ストリングの読み取り

フォーマット

```
#include <stdio.h>
char *fgets (char *string, int n, FILE *stream);
```

言語レベル: ANSI

スレッド・セーフ: はい。

説明

fgets() 関数は、現在の *stream* 位置から最初の改行文字 (¥n) まで (改行文字を含める)、またはストリームの終わりまで、あるいは読み込まれた文字数が *n*-1 と同じになるまでの、最も早いものまでの文字を読

み取ります。fgets() 関数は結果を *string* に格納し、ストリングの終わりに NULL 文字 (¥0) を追加します。改行文字が読み込まれている場合、*string* には改行文字が含まれます。*n* が 1 である場合、*string* は空です。

戻り値

fgets() 関数は、正常に終了した場合、*string* バッファーを指すポインターを戻します。 NULL の戻り値はエラーか、またはファイル終了状態を示します。 NULL 値がエラーを示しているか、ファイルの終わりを示しているかを判別するには、feof() または ferror() 関数を使用します。いずれの場合も、ストリングの値は変わりません。

fgets() 関数は、type=record でオープンされたファイルではサポートされていません。

errno の値は、次のいずれかに設定されます。

値　　意味

EBADF
ファイル・ポインター、または記述子が有効ではありません。

ECONVERT
変換エラーが発生しました。

ENOTREAD
ファイルは読み取り操作用にオープンされていません。

EGETANDPUT
書き込み操作の後に許可されていない読み取り操作が発生しました。

ERECIO
ファイルはレコードの入出力用にオープンしています。

ESTDIN
stdin をオープンできません。

EIOERROR
リカバリー不能な入出力エラーが発生しました。

EIORECERR
リカバリー可能な入出力エラーが発生しました。

fgets() の使用例

次の例は、データ・ストリームから入出力の行を収集します。例では、ストリームから MAX_LEN - 1 文字まで、もしくは改行文字までを読み取ります。

```
#include <stdio.h>

#define  MAX_LEN  100

int main(void)
{
   FILE *stream;
   char line[MAX_LEN], *result;

   stream = fopen("mylib/myfile","rb");

   if ((result = fgets(line,MAX_LEN,stream)) != NULL)
       printf("The string is %s¥n", result);

   if (fclose(stream))
       perror("fclose error");
}
```

関連情報

- 99 ページの『feof() ― ファイル終了標識のテスト』
- 100 ページの『ferror() ― 読み取り/書き込みエラーのテスト』
- 109 ページの『fgetws() ― ストリームのワイド文字のストリングの読み取り』
- 127 ページの『fputs() ― ストリングの書き込み』
- 162 ページの『gets() ― 行の読み取り』
- 251 ページの『puts() ― ストリングの書き込み』
- 17 ページの『<stdio.h>』

fgetwc() ― ストリームのワイド文字の読み取り

フォーマット

```
#include <wchar.h>
#include <stdio.h>
wint_t fgetwc(FILE *stream);
```

言語レベル: ANSI

スレッド・セーフ: はい。

ロケール依存: この関数の振る舞いは、現行ロケールの LC_CTYPE カテゴリーの影響を受ける可能性があります。また、この振る舞いは、LOCALETYPE(*LOCALEUCS2) または LOCALETYPE(*LOCALEUTF) がコンパイル・コマンドに対して指定されている場合は、現行ロケールの LC_UNI_CTYPE カテゴリーの影響を受ける可能性もあります。この関数は、コンパイル・コマンドに対して LOCALETYPE(*CLD) が指定されている場合には使用できません。詳細については、 549 ページの『CCSID およびロケールの理解』を参照してください。

統合ファイル・システム・インターフェース: この関数は、コンパイル・コマンドに対して SYSIFCOPT(*NOIFSIO) が指定されている場合には使用できません。

ワイド文字関数: 詳細については、 553 ページの『ワイド文字』を参照してください。

説明

`fgetwc()` は次のマルチバイト文字を *stream* で指定される入力ストリームから読み取り、それをワイド文字に変換し、(定義されていれば) 関連付けられたストリームのファイル位置標識を進めます。

非ワイド文字関数を同じストリーム上で fgetwc() と共に使用すると、予期しない振る舞いが生じる結果となります。 fgetwc() を呼び出した後、EOF に達しない限り、ストリームの書き込み関数を呼び出す前にバッファーをフラッシュするか、またはストリーム・ポインターの位置を変更します。ストリームに対する書き込み操作の後、fgetwc() を呼び出す前にバッファーをフラッシュするか、またはストリーム・ポインターの位置を変更します。

注: 同じストリームに対する後続の読み取り操作間で現行ロケールが変更された場合は、未定義の結果が発生することがあります。

戻り値

fgetwc() 関数は、*stream* が指す入力ストリームのマルチバイト文字に対応する、次のワイド文字を戻します。ストリームが EOF に達すると、ストリームの EOF 標識が設定され、fgetwc() は WEOF を戻します。

読み取りエラーが発生すると、ストリームのエラー標識が設定され、fgetwc() 関数は WEOF を戻します。エンコード・エラーが発生した場合 (マルチバイト文字をワイド文字に変換中のエラー)、fgetwc() 関数は errno に EILSEQ を設定し、WEOF を戻します。

ferror() および feof() 関数を使用して読み取りエラーと EOF とを区別します。データの最後のバイトを超えて読み取ろうとしたときのみ、EOF に達します。データの最後のバイトまで (最後のバイトを含む) 読み取っても、EOF 標識はオンになりません。

errno の値は、次のいずれかに設定されます。

値 **意味**

EBADF
ファイル・ポインター、または記述子が有効ではありません。

ENOTREAD
ファイルは読み取り操作用にオープンされていません。

EGETANDPUT
書き込み操作の後に許可されていない読み取り操作が発生しました。

ERECIO
ファイルはレコードの入出力用にオープンしています。

ESTDIN
stdin をオープンできません。

EIOERROR
リカバリー不能な入出力エラーが発生しました。

EIORECERR
リカバリー可能な入出力エラーが発生しました。

EILSEQ
無効なマルチバイト文字シーケンスが検出されました。

ECONVERT
変換エラーが発生しました。

fgetwc() の使用例

この例はファイルをオープンし、各ワイド文字を読み取り、その文字を出力します。

```
#include <stdio.h>
#include <stdlib.h>
#include <wchar.h>
#include <errno.h>

int main(void)
{
   FILE   *stream;
   wint_t wc;

   if (NULL == (stream = fopen("fgetwc.dat", "r"))) {
      printf("Unable to open: ¥"fgetwc.dat¥"¥n");
      exit(1);
   }

   errno = 0;
   while (WEOF != (wc = fgetwc(stream)))
      printf("wc = %lc¥n", wc);

   if (EILSEQ == errno) {
      printf("An invalid wide character was encountered.¥n");
      exit(1);
   }
   fclose(stream);
   return 0;
}
* * * End of File * * *
```

関連情報

- 102 ページの『fgetc() ― 文字の読み取り』
- 128 ページの『fputwc() ― 文字の書き込み』
- 『fgetws() ― ストリームのワイド文字のストリングの読み取り』
- 158 ページの『getc() - getchar() ― 文字の読み取り』
- 163 ページの『getwc() ― ストリームからのワイド文字の読み取り』
- 165 ページの『getwchar() ― STDIN からのワイド文字の取得』
- 17 ページの『<stdio.h>』
- 20 ページの『<wchar.h>』

fgetws() ― ストリームのワイド文字のストリングの読み取り

フォーマット

```
#include <wchar.h>
#include <stdio.h>
wchar_t *fgetws(wchar_t *wcs, int n, FILE *stream);
```

言語レベル: ANSI

スレッド・セーフ: はい。

ロケール依存: この関数の振る舞いは、現行ロケールの LC_CTYPE カテゴリーの影響を受ける可能性があります。また、この振る舞いは、LOCALETYPE(*LOCALEUCS2) または LOCALETYPE(*LOCALEUTF) がコンパイル・コマンドに対して指定されている場合は、現行ロケールの LC_UNI_CTYPE カテゴリーの影

響を受ける可能性もあります。この関数は、コンパイル・コマンドに対して LOCALETYPE(*CLD) が指定されている場合には使用できません。詳細については、 549 ページの『CCSID およびロケールの理解』を参照してください。

統合ファイル・システム・インターフェース: この関数は、コンパイル・コマンドに対して SYSIFCOPT(*NOIFSIO) が指定されている場合には使用できません。

ワイド文字関数: 詳細については、 553 ページの『ワイド文字』を参照してください。

説明

`fgetws()` 関数は、 *stream* が指すストリームからワイド文字を読み取ります。読み取られる文字数の上限は *n* より 1 つ少ない数となります。`fgetws()` 関数は WEOF の後か、または改行ワイド文字 (これは、保持されます) を読み取った後、文字の読み取りを停止します。また、最後のワイド文字が配列へ読み取られた直後に NULL ワイド文字を追加します。 `fgetws()` 関数はエラーがない限り、ファイル位置を進めます。エラーが発生した場合、ファイル位置は予期できません。

非ワイド文字関数を `fgetws()` 関数と共に同じストリーム上で使用すると、予期しない振る舞いが生じる結果となります。 `fgetws()` 関数を呼び出した後、WEOF に達しない限り、ストリームの書き込み関数を呼び出す前にバッファーをフラッシュするか、ストリーム・ポインターの位置を変更してください。ストリームに対する書き込み操作の後、`fgetws()` 関数を呼び出す前にバッファーをフラッシュするか、ストリーム・ポインターの位置を変更してください。

注: 同じストリームに対する後続の読み取り操作間で現行ロケールが変更された場合は、未定義の結果が発生することがあります。

戻り値

正常終了の場合は、`fgetws()` 関数はワイド文字ストリング *wcs* を指すポインターを戻します。ワイド文字が *wcs* に読み取られる前に WEOF が検出された場合は、*wcs* の内容は変更されないままで、`fgetws()` 関数は NULL ポインターを戻します。データがストリング・バッファーにすでに読み取られた後で WEOF に達した場合は、`fgetws()` 関数はストリング・バッファーを指すポインターを戻し、正常終了したことを示します。この後に続く呼び出しは、データを読み取らずに WEOF に達するため、NULL を戻します。

読み取りエラーが発生すると、*wcs* の内容は確定できず、`fgetws()` 関数は NULL を戻します。エンコード・エラーが発生した場合 (ワイド文字をマルチバイト文字に変換中に)、`fgetws()` 関数は errno に EILSEQ を設定し、NULL を戻します。

n が 1 に等しいと、*wcs* バッファーは最後の NULL 文字分の空きしかなく、ストリームからは何も読み取られません。 (そのような操作でも、読み取り操作であるとみなされます。したがって、最初にバッファーがフラッシュされたり、ストリーム・ポインターが位置変更されない限り、直後で書き込み操作を行うことはできません)。 *n* が 1 より大きいときは、入出力エラーが起こるか、データがストリームから読み取られる前に WEOF に達した場合にのみ、`fgetws()` 関数が失敗します。

`ferror()` および `feof()` 関数を使用して読み取りエラーと WEOF とを区別します。 WEOF に達するのは、データの最後のバイトを超えて読み取ろうとした場合のみです。データの最後のバイトまで (最後のバイトを含む) 読み取っても、WEOF 標識はオンになりません。

`fgetws()` の errno 値について詳しくは、107 ページの『fgetwc() ― ストリームのワイド文字の読み取り』を参照してください。

`fgetws()` の使用例

この例はファイルをオープンし、ファイルの内容を読み取ってからそのファイルの内容を出力します。

```
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <wchar.h>

int main(void)
{
   FILE    *stream;
   wchar_t  wcs[100];

   if (NULL == (stream = fopen("fgetws.dat", "r"))) {
      printf("Unable to open: ¥"fgetws.dat¥"¥n");
      exit(1);
   }

   errno = 0;
   if (NULL == fgetws(wcs, 100, stream)) {
      if (EILSEQ == errno) {
         printf("An invalid wide character was encountered.¥n");
         exit(1);
      }
      else if (feof(stream))
              printf("End of file reached.¥n");
           else
              perror("Read error.¥n");
   }
   printf("wcs = ¥"%ls¥"¥n", wcs);
   fclose(stream);
   return 0;

   /************************************************************
      Assuming the file fgetws.dat contains:

      This test string should not return -1

      The output should be similar to:

      wcs = "This test string should not return -1"
   ************************************************************/
}
```

関連情報

- 102 ページの『fgetc() ― 文字の読み取り』
- 105 ページの『fgets() ― ストリングの読み取り』
- 107 ページの『fgetwc() ― ストリームのワイド文字の読み取り』
- 130 ページの『fputws() ― ワイド文字ストリングの書き込み』
- 17 ページの『<stdio.h>』
- 20 ページの『<wchar.h>』

fileno() ― ファイル・ハンドルの判別

フォーマット

```
#include <stdio.h>
int fileno(FILE *stream);
```

言語レベル: XPG4

スレッド・セーフ: はい。

統合ファイル・システム・インターフェース: この関数は、コンパイル・コマンドに対して SYSIFCOPT(*NOIFSIO) が指定されている場合には使用できません。

説明

fileno() 関数は、現在 *stream* と関連付けられているファイル・ハンドルを判別します。

戻り値

環境変数 QIBM_USE_DESCRIPTOR_STDIO が Yes に設定されている場合、fileno() 関数は stdin に対して 0 を、stdout に対して 1 を、そして stderr に対して 2 を戻します。

QIBM_USE_DESCRIPTOR_STDIO を No に設定すると、ILE C セッション・ファイルの stdin、stdout、および stderr は、それらに関連付けられているファイル記述子を持ちません。その場合は、fileno() 関数は -1 を戻します。

errno の値は、EBADF に設定できます。

fileno() の使用例

次の例では stderr データ・ストリームのファイル・ハンドルを判別します。

```
/* Compile with SYSIFCOPT(*IFSIO)         */
 #include <stdio.h>

 int main (void)
   {
   FILE *fp;
   int result;

   fp = fopen ("stderr","w");

   result = fileno(fp);
   printf("The file handle associated with stderr is %d.¥n", result);
   return 0;

   /*****************************************************************
    * The output should be:
    *
    * The file handle associated with stderr is -1.
    ****************************************************************/
  }
```

関連情報

- 114 ページの『fopen() ― ファイルのオープン』
- 136 ページの『freopen() ― オープン・ファイルのリダイレクト』
- 17 ページの『<stdio.h>』

floor() ― 整数の検索 <=引数

フォーマット

```
#include <math.h>
double floor(double x);
```

言語レベル: ANSI

スレッド・セーフ: はい。

説明

floor() 関数は、*x* 以下の最大整数を計算します。

戻り値

floor() 関数は、浮動小数点の結果を double 値として戻します。

floor() の結果は範囲エラーにはなりません。

floor() の使用例

次の例では、*y* に 2.8 以下の最大整数の値を、*z* に -2.8 以下の最大整数の値を割り当てます。

```
#include <math.h>
#include <stdio.h>

int main(void)
{
   double y, z;

   y = floor(2.8);
   z = floor(-2.8);

   printf("floor(  2.8 ) = %lf¥n", y);
   printf("floor( -2.8 ) = %lf¥n", z);
}
/*******************  Output should be similar to:  ***************

floor(  2.8 ) = 2.000000
floor( -2.8 ) = -3.000000
*/
```

関連情報

- 64 ページの『ceil() ― 整数の検索 >= 引数』
- 『fmod() ― 浮動小数点の剰余の計算』
- 9 ページの『<math.h>』

fmod() ― 浮動小数点の剰余の計算

フォーマット

```
#include <math.h>
double fmod(double x, double y);
```

言語レベル: ANSI

スレッド・セーフ: はい。

説明

fmod() 関数は、*x/y* の浮動小数点の剰余を計算します。結果の絶対値は常に *y* の絶対値よりも小さくなります。結果は *x* と同じ符号を持ちます。

戻り値

fmod() 関数は *x/y* の浮動小数点の剰余を戻します。*y* がゼロか、または *x/y* によってオーバーフローが生じた場合は、fmod() は 0 を戻します。errno の値は EDOM に設定される可能性があります。

fmod() の使用例

次の例は *x/y* の剰余としての *z* を計算します。ここで、*x/y* は、剰余 -1 を持つ -3 です。

```
#include <math.h>
#include <stdio.h>

int main(void)
{
   double x, y, z;

   x = -10.0;
   y = 3.0;
   z = fmod(x,y);      /* z = -1.0 */

   printf("fmod( %lf, %lf) = %lf¥n", x, y, z);
}

/*******************  Output should be similar to:  ***************

fmod( -10.000000, 3.000000) = -1.000000
*/
```

関連情報

- 64 ページの『ceil() ― 整数の検索 >= 引数』
- 94 ページの『fabs() ― 浮動小数点絶対値の計算』
- 112 ページの『floor() ― 整数の検索 <=引数』
- 9 ページの『<math.h>』

fopen() ― ファイルのオープン

フォーマット

```
#include <stdio.h>
FILE *fopen(const char *filename, const char *mode);
```

言語レベル: ANSI

スレッド・セーフ: はい。

説明

fopen() 関数は、*filename* により指定されるファイルをオープンします。 *mode* パラメーターは、そのファイルに要求されたアクセス・タイプを指定する文字ストリングです。 *mode* 変数には、オプションのキーワード・パラメーターが後に続く、位置パラメーターが 1 つ含まれます。

注: プログラムが SYSIFCOPT(*IFSIO) または SYSIFCOPT(*IFS64IO) でコンパイルされている場合で、fopen() が統合ファイル・システムでファイルを作成すると、ファイルの所有者、所有者のグループ、およびパブリックに、ファイルに対する読み取り権限、書き込み権限、および実行権限が与えられます。

位置パラメーターには次の値が可能です。

モード 説明

r 読み取り用にテキスト・ファイルをオープンするファイルが存在している必要があります。

w 書き込み用にテキスト・ファイルを作成する。所定のファイルが存在している場合、それが論理ファイルでない限り、その内容は破棄されます。

a ファイルの終わりに書き込む付加モードで、テキスト・ファイルをオープンする。fopen() 関数は、ファイルが存在しておらず、論理ファイルでない場合にファイルを作成します。

r+ 読み取りおよび書き込み用にテキスト・ファイルをオープンする。ファイルが存在している必要があります。

w+ 読み取りおよび書き込み用にテキスト・ファイルを作成する。所定のファイルが存在している場合、それが論理ファイルでない限り、その内容はクリアされます。

a+ 読み取りおよびファイルの終わりでの更新用に、付加モードでテキスト・ファイルをオープンする。fopen() 関数は、ファイルが存在していないときにはファイルを作成します。

rb 読み取り用にバイナリー・ファイルをオープンするファイルが存在している必要があります。

wb 書き込み用に空のバイナリー・ファイルを作成する。ファイルが存在している場合、それが論理ファイルでない限り、内容はクリアされます。

ab ファイルの終わりでの書き込み用に、付加モードでバイナリー・ファイルをオープンする。fopen 関数は、ファイルが存在していないときにはファイルを作成します。

r+b *または* **rb+**
読み取りおよび書き込み用にバイナリー・ファイルをオープンする。ファイルが存在している必要があります。

w+b *または* **wb+**
読み取りおよび書き込み用に空のバイナリー・ファイルを作成する。ファイルが存在している場合、それが論理ファイルでない限り、その内容はクリアされます。

a+b *または* **ab+**
ファイルの終わりでの書き込み用に、付加モードでバイナリー・ファイルをオープンする。fopen() 関数は、ファイルが存在していないときにはファイルを作成します。

注:

1. fopen() 関数は、属性 type=record および ab+, rb+, または wb+ でオープンされたファイルではサポートされていません。
2. w、w+、wb、w+b、および wb+ パラメーターを使用する場合には注意が必要です。同じ名前の既存ファイルにあるデータは失われます。

テキスト・ファイル には出力可能文字と制御文字が含まれ、これらの文字で行が構成されています。コンパイラーによっては、おそらく最終行を除き、各行は改行文字で終了します。システムは出力テキスト・ストリームに制御文字を挿入または変換することができます。 fopen() 関数モード ″a″ および ″a+″ は QSYS.LIB ファイル・システムには使用できません。すべてのモードで、テキスト・ファイル用に QSYS.LIB ファイル・システムを使用する場合には実装制限があります。ファイルの開始を超えた検索は、テキスト・モードでオープンされたストリームの操作には信頼できません。

注: fopen() を使用して QSYS.LIB ファイル・システムでファイルを作成する場合、ライブラリー名を *LIBL またはブランクに指定するとファイルが QTEMP ライブラリーに作成されてしまいます。

テキスト・ファイルが存在していない場合、次のコマンドを使用して作成できます。

CRTSRCPF FILE(MYLIB/MYFILE) RCDLEN(LRECL) MBR(MYMBR) SYSTEM(*FILETYPE)

注: テキスト・ストリームへのデータ出力は入力時の同じデータと等しくない場合があります。 QSYS.LIB ファイル・システムはデータベース・ファイルをメンバーのディレクトリーとして扱います。 fopen() 関数を使用する場合、メンバーが動的に作成される前にデータベース・ファイルが存在している必要があります。

統合ファイル・システムの現在のファイル・システムの制限について詳しくは、i5/OS Information Center の統合ファイル・システムのトピックにある『ラージ・ファイル・サポート』を参照してください。 2 GB を超える統合ファイル・システムのファイルについては、ご使用のアプリケーションのプログラム・アクセスに 64 ビット C ランタイム関数を許可する必要があります。次のメソッドを使用してプログラム・アクセスを許可できます。

- コンパイル・コマンドで SYSIFCOPT(*IFS64IO) を指定します。これにより、ネイティブ C コンパイラーが _IFS64_IO_ を定義します。こうすると、マクロ _LARGE_FILES および _LARGE_FILE_API が定義されます。
- プログラム・ソースで、またはコンパイル・コマンドに DEFINE('_LARGE_FILES') を指定して、マクロ _LARGE_FILES を定義します。既存の C ランタイム関数およびコード内の関連データ型はすべて自動的に 64 ビット・バージョンにマップされるか、再定義されます。
- プログラム・ソースで、またはコンパイル・コマンドに DEFINE('_LARGE_FILE_API') を指定して、マクロ _LARGE_FILE_API を定義します。これにより、新しい 64 ビット C ランタイム関数およびデータ型のセットが見えるようになります。アプリケーションは使用する C ランタイム関数の名前を、既存バージョンと 64 ビット・バージョンの両方について明示的に指定する必要があります。

64 ビット C ランタイム関数には次のものがあります。int fgetpos64()、FILE *fopen64()、FILE *freopen64()、FILE *wfopen64()、int fsetpos64(FILE *, const fpost64_t *)、FILE *tmpfile64()、int fseeko(FILE *, off_t, int)、int fseeko64(FILE *, off64_t, int)、off_t ftello(FILE *)、off64_t ftello64()。

バイナリー・ファイル には、一連の文字が含まれます。バイナリー・ファイルでは、システムは入力または出力の際に制御文字を変換しません。

バイナリー・ファイルが存在していない場合、次のコマンドを使用して作成できます。

CRTPF FILE(MYLIB/MYFILE) RCDLEN(LRECL) MBR(MYMBR) MAXMBRS(*NOMAX) SYSTEM(*FILETYPE)

ファイルを a、a+、ab、a+b または ab+ モードでオープンする場合は、すべての書き込み操作はファイルの終わりで行われます。 fseek() 関数または rewind() 関数を使用してファイル・ポインターの位置を変更できますが、書き込み関数は操作を実行する前にファイル・ポインターをファイルの終わりに移動し直します。このアクションは、既存データへの上書きを防ぐためのものです。

更新モード (2 または 3 番目の位置に + を使用して) を指定する場合は、ファイルの読み書き両方ができます。ただし、読み取りと書き込みの切り替えを行うとき、fseek()、fsetpos()、rewind()、または fflush() などの位置設定関数を介在させる必要があります。出力は、ファイルの終わりが検出された場合に直ちに入力に続くことができます。

非統合ファイル・システムのキーワード・パラメーター

blksize=*value*
: レコードの物理ブロックの最大長をバイトで指定する。

lrecl=*value*
: 固定長レコードの長さと、可変長レコードの最大長を、バイトで指定する。

recfm=*value*
: 指定できる *value* は、以下の通りです。

F 固定長、非ブロック化レコード

FB 固定長、ブロック化レコード

V 可変長、非ブロック化レコード

VB 可変長、ブロック化レコード

VBS テープ・ファイル用の可変長で、ブロック化したスパン・レコード

VS テープ・ファイル用の可変長で、非ブロック化したスパン・レコード

D テープ・ファイル用 ASCII D 形式の、可変長で、非ブロック化した非スパン・レコード

DB テープ・ファイル用 ASCII D 形式の、可変長で、ブロック化した非スパン・レコード

U テープ・ファイルの不定形式

FA 固定長で、出力ファイル用に先頭文字書式制御データを使用

注: ファイルが CTLCHAR(*FCFC) を使用して作成されている場合、先頭文字の書式制御が使用されます。ファイルが CTLCHAR(*NONE) を使用して作成されている場合、先頭文字書式制御は使用されません。

commit=*value*
: 指定できる *value* は、以下の通りです。

N このパラメーターは、このファイルがコミットメント制御下でオープンされないことを識別します。これはデフォルトです。

Y このパラメーターは、このファイルがコミットメント制御下でオープンされることを識別します。

ccsid=*value*
: i5/OS オペレーティング・システムでサポートされていない CCSID が指定された場合、これはデータ管理によって無視されます。

コンパイル・コマンドに LOCALETYPE(*LOCALEUTF) が指定された場合は、デフォルト値は LC_CTYPE CCSID の値です。これは、現行ロケール設定によって決まります。ロケール設定について詳しくは、354 ページの『setlocale() ー ロケールの設定』を参照してください。コンパイル・コマンドに LOCALETYPE(*LOCALEUTF) が指定されていない場合、デフォルト値はジョブの CCSID の値です。ファイルの CCSID の値について詳しくは、550 ページの『ファイルの CCSID』を参照してください。

arrseq=*value*
: *value* は、以下のいずれかになります。

N このパラメーターは、ファイルが作成された方法で、このファイルが処理されていることを識別します。これはデフォルトです。

Y このパラメーターは、このファイルが到着順で処理されることを識別します。

indicators=*value*

指定できる *value* は、以下の通りです。

N この値は、ディスプレイ、ICF、またはプリンター・ファイルの標識がファイル・バッファーに格納されていることを識別します。これはデフォルトです。

Y このパラメーターは、ディスプレイ、ICF、またはプリンター・ファイルの標識が、ファイル・バッファーではなく、別の標識領域に格納されていることを識別します。ファイル・バッファーは、書き込みと読み込みのときにシステムがユーザー・プログラムとオペレーション・システム間でデータを転送するために使用する領域です。 ICF ファイルを処理する場合には標識を別の標識領域に格納する必要があります。

type=*value*

指定できる *value* は、以下の通りです。

memory このパラメーターは、C プログラムからのみ使用できるメモリー・ファイルとして、このファイルを識別します。これはデフォルトです。

record このパラメーターは、順次レコード入出力用にファイルがオープンされることを指定します。ファイルはバイナリー・ファイルとしてオープンされる必要があります。そうでない場合、`fopen()` 関数は失敗します。読み取り操作と書き込み操作は `fread()` 関数と `fwrite()` 関数によって行います。

統合ファイル・システムのキーワード・パラメーター

type=*value*

指定できる *value* は、以下の通りです。

record ファイルは順次レコード入出力用にオープンされます。(ファイルはバイナリー・ストリームとしてオープンされる必要があります。)

ccsid=*value*

ccsid はコード・ページ値に変換されます。デフォルトとして、コード・ページとしてジョブ CCSID の値を使用します。CCSID とコード・ページ・オプションの両方を指定することはできません。 CCSID オプションは i5/OS およびデータ管理ベースのストリーム入出力との互換性を提供します。

注: 混合データ (データに 1 バイト文字と 2 バイト文字の両方が含まれている) は、テキストのファイル・データ処理モードではサポートされていません。混合データは、バイナリーのファイル処理モードでサポートされています。

ccsid キーワードを指定する場合は、o_ccsid キーワードやコード・ページ・キーワードを指定できません。

変換されたデータが拡大したり縮小する可能性があるため、データ・サイズや現行のファイル・オフセットを推測することは危険です。例えば、100 バイトの物理サイズを持つファイルで、アプリケーションがファイルから 100 バイトを読み取った後のファイル・オフセットは 50 しかない場合があります。含まれている CCSID によっては、ファイル全体を読み取るために、アプリケーションが 200 バイト以上を読み取る必要がある場合があります。このため、`ftell()`、`fseek()`、`fgetpos()`、`fsetpos()` などのファイル位置決め関数が機能しないことがあります。これらの関数はエラー ENOTSUP が発生して失敗します。関数の読み取りは、デフォルトのように、バッファリングがオンの場合にも機能しません。バッファリングをオフにするには、_IONBF キーワードを指定して setvbuf 関数を使用します。

次の 3 つの状態すべてが起こった場合、fopen() 関数は ECONVERT エラーを発生して失敗することがあります。

- ファイル・データ処理モードがテキストである。
- コード・ページが指定されていない。
- ジョブの CCSID が「混合データ」(1 バイト文字と 2 バイト文字の両方が含まれているデータ) である。

o_ccsid=*value*

コンパイル・コマンドに LOCALETYPE(*LOCALEUTF) が指定された場合は、デフォルト値は LC_CTYPE CCSID の値です。これは、現行ロケール設定によって決まります。ロケール設定について詳しくは、354 ページの『setlocale() ― ロケールの設定』を参照してください。コンパイル・コマンドに LOCALETYPE(*LOCALEUTF) が指定されていない場合、デフォルト値はジョブの CCSID の値です。ファイルの CCSID の値について詳しくは、550 ページの『ファイルの CCSID』を参照してください。

このパラメーターは、指定された値がコード・ページに変換されない点を除き、*ccsid* パラメーターと類似しています。また、混合データもサポートされています。ファイルが作成されると、指定された CCSID でタグ付けされます。ファイルがすでに存在している場合、データは読み取り操作時に、ファイルの CCSID から指定された CCSID に変換されます。書き込み操作では、データは指定された CCSID にあると仮定され、ファイルの CCSID に変換されます。

変換されたデータが拡大したり縮小する可能性があるため、データ・サイズや現行のファイル・オフセットを推測することは危険です。例えば、100 バイトの物理サイズを持つファイルで、アプリケーションがファイルから 100 バイトを読み取った後のファイル・オフセットは 50 しかない場合があります。含まれている CCSID によっては、ファイル全体を読み取るために、アプリケーションが 200 バイト以上を読み取る必要がある場合があります。このため、ftell()、fseek()、fgetpos()、fsetpos() などのファイル位置決め関数が機能しません。これらの関数は、ENOTSUP を発生して失敗します。関数の読み取りは、デフォルトのように、バッファリングがオンの場合にも機能しません。バッファリングをオフにするには、_IONBF キーワードを指定して setvbuf 関数を使用します。

o_ccsid の使用例

```
/* Create a file that is tagged with CCSID 37 */
if ((fp = fopen("/MYFILE" , "w, o_ccsid=37")) == NULL) {
   printf("Failed to open file with o_ccsid=37¥n");
}

fclose(fp);

/* Now reopen the file with CCSID 13488, because your application
 wants to deal with the data in UNICODE */

if ((fp = fopen("/MYFILE" , "r+, o_ccsid=13488")) == NULL) {
   printf("Failed to open file with o_ccsid=13488¥n");
}
/* Turn buffering off because read functions do not work when
buffering is on */

if (setbuf(fp, NULL, _IONBF, 0) != 0){
    printf("Unable to turn buffering off¥n");
}
/* Because you opened with o_ccsid = 13488, you must provide
all input data as unicode.
If this program is compiled with LOCALETYPE(*LOCALEUCS2),
L constrants will be unicode. */
```

```
funcreturn = fputws(L"ABC", fp); /* Write a unicode ABC to the file. */

if (funcreturn < 0) {
   printf("Error with 'fputws' on line %d¥n", __LINE__);
}
/* Because the file was tagged with CCSID 37, the unicode ABC was
converted to EBCDIC ABC when it was written to the file. */
```

codepage=*value*
: *value* により指定されるコード・ページが使用されます。

  コード・ページ・キーワードを指定する場合は、ccsid キーワードや o_ccsid キーワードは指定できません。

  オープンするファイルが存在しておらず、オープン・モードでファイルを作成するように指定した場合は、ファイルが作成されて計算されたコード・ページでタグ付けされます。ファイルがすでに存在している場合は、ファイルから読み取られるデータがファイル・コード・ページから計算されたコード・ページに、読み取り操作中に変換されます。ファイルに書き込まれるデータは、計算されたコード・ページにあり、書き込み操作中にファイルのコード・ページに変換されると仮定されます。

crln=*value*
: 指定できる *value* は、以下の通りです。

  Y 使用される行の終了文字は、復帰改行文字 [CR] と改行文字 [NL] の組み合わせです。データが読み取られるときに、復帰改行文字 [CR] が string 関数のためにストリップされます。データがファイルに書き込まれるときには、復帰改行文字 [CR] がそれぞれの改行文字 [NL] の前に追加されます。行の終了文字処理は、ファイルがテキスト・モードでオープンされている場合にのみ発生します。これはデフォルトです。

  N 使用される行の終了文字は、改行文字 [NL] のみです。

キーワード・パラメーターは大文字と小文字の区別がなく、コンマで区切ります。

パラメーターが一致しない場合、一般に `fopen()` 関数は失敗します。

戻り値

`fopen()` 関数は、オープン・ファイルにアクセスするために使用できる **FILE** 構造体の型を指すポインターを戻します。

注: ストリーム・ファイル (type = record) を record I/O 関数と一緒に使用する場合、**FILE** ポインターを **RFILE** ポインターにキャストする必要があります。

NULL ポインターの戻り値は、エラーを示します。

errno の値は、次のいずれかに設定されます。

値　　**意味**

EBADMODE
: 指定されたファイル・モードが無効です。

EBADNAME
: 指定されたファイル名が無効です。

ECONEVRT
: 変換エラー。

ENOENT
: ファイルまたはライブラリーがありません。

ENOMEM
: ストレージ割り振り要求が失敗しました。

ENOTOPEN
: ファイルはオープンされていません。

EIOERROR
: リカバリー不能な入出力エラーが発生しました。

EIORECERR
: リカバリー可能な入出力エラーが発生しました。

ESCANFAILURE
: ファイルは走査失敗のマークを付けられました。

`fopen()` に渡されたモード・ストリングが正しい場合、ファイル・タイプにかかわらず、`fopen()` は errno を EBADMODE に設定しません。

`fopen()` に渡されたモード・ストリングが有効でない場合、ファイル・タイプにかかわらず、`fopen()` は errno を EBADMODE に設定します。

`fopen()` に渡されたモード・ストリングは正しいが、その指定されたファイル・タイプについては無効な場合は、ファイル・タイプにかかわらず、`fopen()` は errno を ENOTOPEN、EIOERROR、または EIORECERR に設定します。

`fopen()` の使用例

この例は、読み取り用ファイルのオープンを試行します。

```
#include <stdio.h>
#define  MAX_LEN  60

int main(void)
{
   FILE *stream;
   fpos_t pos;
   char line1[MAX_LEN];
   char line2[MAX_LEN];
   char *result;
   char ch;
   int num;

   /* The following call opens a text file for reading.   */
   if ((stream = fopen("mylib/myfile", "r")) == NULL)
      printf("Could not open data file¥n");
   else if ((result = fgets(line1,MAX_LEN,stream)) != NULL)
           {
            printf("The string read from myfile: %s¥n", result);
            fclose(stream);
           }

   /* The following call opens a fixed record length file */
   /* for reading and writing.                             */
   if ((stream = fopen("mylib/myfile2", "rb+, lrecl=80,  ¥
                 blksize=240, recfm=f")) == NULL)
         printf("Could not open data file¥n");
   else {
         fgetpos(stream, Point-of-Sale);
         if (!fread(line2,sizeof(line2),1,stream))
            perror("fread error");
         else printf("1st record read from myfile2: %s¥n", line2);

         fsetpos(stream, Point-of-Sale);     /* Reset pointer to start of file */
         fputs(result, stream);     /* The line read from myfile is   */
                                    /* written to myfile2.            */
         fclose(stream);
        }
}
```

関連情報

- 95 ページの『fclose() ― ストリームのクローズ』
- 101 ページの『fflush() ― ファイルへのバッファーの書き込み』
- 132 ページの『fread() ― 項目の読み取り』
- 136 ページの『freopen() ― オープン・ファイルのリダイレクト』
- 140 ページの『fseek() ― fseeko() ― ファイル位置の位置変更』
- 142 ページの『fsetpos() ― ファイル位置の設定』
- 152 ページの『fwrite() ― 項目の書き込み』
- 288 ページの『rewind() ― 現在のファイル位置の調整』
- 521 ページの『wfopen() ―オープン・ファイル』
- 17 ページの『<stdio.h>』
- open() i5/OS Information Center の 『API』にある API。

fprintf() ― フォーマット済みデータのストリームへの書き込み

フォーマット

```
#include <stdio.h>
int fprintf(FILE *stream, const char *format-string, argument-list);
```

言語レベル: ANSI

スレッド・セーフ: はい。

ロケール依存: この関数の振る舞いは、現行ロケールの LC_CTYPE カテゴリーおよび LC_NUMERIC カテゴリーの影響を受ける可能性があります。また、この振る舞いは、LOCALETYPE(*LOCALEUCS2) または LOCALETYPE(*LOCALEUTF) がコンパイル・コマンドに対して指定されている場合は、現行ロケールの LC_UNI_CTYPE カテゴリーの影響を受ける可能性もあります。詳細については、 549 ページの『CCSID およびロケールの理解』を参照してください。

説明

`fprintf()` 関数は、一連の文字および値をフォーマット設定し、出力 *stream* に書き込みます。`fprintf()` 関数は、もしあれば、*関数リスト* の各エントリーを変換し、それを*フォーマット・ストリング* の対応する書式指定に従ってストリームに書き込みます。

format-string は、`printf()` 関数の *format-string* 引数と同じ形式および機能を持っています。

戻り値

`fprintf()` 関数は、出力エラーが起こった場合に、出力されたバイト数または負の値を戻します。

`fprintf()` の errno 値について詳しくは、238 ページの『printf() ― 定様式の文字の出力』を参照してください。

`fprintf()` の使用例

次の例は、配列 count の各整数のアスタリスクの行をファイル myfile に送信します。各行に出力されるアスタリスクの数は、配列内の整数に相当します。

```
#include <stdio.h>

int count [10] = {1, 5, 8, 3, 0, 3, 5, 6, 8, 10};

int main(void)
{
   int i,j;
   FILE *stream;

   stream = fopen("mylib/myfile", "w");
                 /* Open the stream for writing */
   for (i=0; i < sizeof(count) / sizeof(count[0]); i++)
   {
      for (j = 0; j < count[i]; j++)
         fprintf(stream,"*");
                 /* Print asterisk              */
         fprintf(stream,"¥n");
                 /* Move to the next line       */
   }
   fclose (stream);
}

/*******************  Output should be similar to:  ***************

*
*****
********
***

***
*****
******
********
**********
*/
```

関連情報

- 138 ページの『fscanf() ― フォーマット済みデータの読み取り』
- 149 ページの『fwprintf() ― ワイド文字としてのデータのフォーマット設定とストリームへの書き込み』
- 238 ページの『printf() ― 定様式の文字の出力』
- 368 ページの『sprintf() ― フォーマット設定データのバッファーへの出力』
- 444 ページの『vfprintf() ― ストリームへの引数データの出力』
- 452 ページの『vprintf() ― 引数データの出力』
- 456 ページの『vsprintf() ― 引数データのバッファーへの出力』
- 17 ページの『<stdio.h>』

fputc() ― 文字の書き込み

フォーマット

```
#include <stdio.h>
int fputc(int c, FILE *stream);
```

言語レベル: ANSI

スレッド・セーフ: はい。

説明

`fputc()` 関数は *c* を unsigned char に変換してから *c* を出力 *stream* の現在位置に書き込み、正しくファイル位置を進めます。ストリームが追加モードの 1 つでオープンされている場合は、文字はストリームの最後に付け加えられます。

`fputc()` 関数は `putc()` と同じですが、常に関数呼び出しとして定義され、マクロで置き換えられることはありません。

戻り値

`fputc()` 関数は書き込まれる文字を戻します。EOF の戻り値は、エラーを示します。

errno の値は、次のいずれかに設定されます。

値　　意味

ECONVERT
　　変換エラーが発生しました。

ENOTWRITE
　　ファイルは書き込み操作用にオープンされません。

EPUTANDGET
　　読み取り操作の後に、許可されていない書き込み操作が発生しました。

ERECIO
　　ファイルはレコードの入出力用にオープンしています。

ESTDERR
　　`stderr` をオープンできません。

ESTDOUT
　　`stdout` をオープンできません。

EIOERROR
　　リカバリー不能な入出力エラーが発生しました。

EIORECERR
　　リカバリー可能な入出力エラーが発生しました。

`fputc()` 関数は、type=record でオープンされたファイルではサポートされていません。

`fputc()` の使用例

次の例は buffer の内容を *myfile* と呼ばれるファイルに書き込みます。

注: 出力が for ステートメントの 2 番目の式内の副次作用として発生するため、このステートメント本体は NULL です。

```
#include <stdio.h>

#define NUM_ALPHA  26

int main(void)
{
  FILE * stream;
  int i;
  int ch;

  char buffer[NUM_ALPHA + 1] = "abcdefghijklmnopqrstuvwxyz";

  if (( stream = fopen("mylib/myfile", "w"))!= NULL )
  {
    /* Put buffer into file */
    for ( i = 0; ( i < sizeof(buffer) ) &&
           ((ch = fputc( buffer[i], stream)) != EOF ); ++i );
    fclose( stream );
  }
  else
    perror( "Error opening myfile" );

}
```

関連情報

- 102 ページの『fgetc() ― 文字の読み取り』
- 249 ページの『putc() - putchar() ― 文字の書き込み』
- 17 ページの『<stdio.h>』

_fputchar - 文字の書き込み

フォーマット

```
#include <stdio.h>
int _fputchar(int c);
```

言語レベル: Extension

スレッド・セーフ: はい。

説明

`_fputchar` は、単一文字 c を現在位置の STDOUT ストリームに書き込みます。これは次の fputc 呼び出しと同じです。

```
fputc(c, stdout);
```

移植性を考慮する場合、_fputchar の代わりに ANSI/ISO fputc 関数を使用してください。

戻り値

`_fputchar` は書き込まれる文字を戻します。戻り値 `EOF` は、書き込みエラーが発生したことを示します。`ferror` および `feof` を使用して、これがエラー状態であるか、ファイル終了であるかを示します。

`_fputchar` の errno の値については、124 ページの『fputc() ― 文字の書き込み』を参照してください。

`_fputchar()` の使用例

次の例は、バッファーの内容を STDOUT に書き込みます。

```
#include <stdio.h>
int main(void)
{
    char buffer[80];
    int i,ch = 1;
    for (i = 0; i < 80; i++)
       buffer[i] = 'c';
    for (i = 0; (i < 80) && (ch != EOF); i++)
       ch = _fputchar(buffer[i]);
    printf("\n");
    return 0;
}
```

The output should be similar to:

```
cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
```

関連情報

- 158 ページの『getc() - getchar() ― 文字の読み取り』
- 124 ページの『fputc() ― 文字の書き込み』
- 249 ページの『putc() - putchar() ― 文字の書き込み』
- 17 ページの『<stdio.h>』

fputs() ― ストリングの書き込み

フォーマット

```
#include <stdio.h>
int fputs(const char *string, FILE *stream);
```

言語レベル: ANSI

スレッド・セーフ: はい。

説明

`fputs()` 関数はストリング を現在位置で出力 *stream* に書き込みます。ストリングの最後のヌル文字 (¥0) は書き込みません。

戻り値

エラーが発生すると、`fputs()` 関数は EOF を戻します。それ以外の場合は、負の値以外の値を戻します。

`fputs()` 関数は、type=record でオープンされたファイルではサポートされていません。

`fputs()` の errno 値について詳しくは、124 ページの『fputc() ― 文字の書き込み』を参照してください。

`fputs()` の使用例

次の例はストリングをストリームに書き込みます。

```
#include <stdio.h>

#define NUM_ALPHA  26

int main(void)
{
  FILE * stream;
  int num;

  /* Do not forget that the '¥0' char occupies one character */
  static char buffer[NUM_ALPHA + 1] = "abcdefghijklmnopqrstuvwxyz";

  if ((stream = fopen("mylib/myfile", "w")) != NULL )
  {
     /* Put buffer into file */
     if ( (num = fputs( buffer, stream )) != EOF )
     {
       /* Note that fputs() does not copy the ¥0 character */
       printf( "Total number of characters written to file = %i¥n", num );
       fclose( stream );
     }
     else   /* fputs failed */
       perror( "fputs failed" );
  }
  else
     perror( "Error opening myfile" );
}
```

関連情報

- 105 ページの『fgets() ― ストリングの読み取り』
- 130 ページの『fputws() ― ワイド文字ストリングの書き込み』
- 162 ページの『gets() ― 行の読み取り』
- 251 ページの『puts() ― ストリングの書き込み』
- 17 ページの『<stdio.h>』

fputwc() ― 文字の書き込み

フォーマット

```
#include <wchar.h>
#include <stdio.h>
wint_t fputwc(wint_t wc, FILE *stream);
```

言語レベル: ANSI

スレッド・セーフ: はい。

ロケール依存: この関数の振る舞いは、現行ロケールの LC_CTYPE カテゴリーの影響を受ける可能性があります。また、この振る舞いは、LOCALETYPE(*LOCALEUCS2) または LOCALETYPE(*LOCALEUTF) がコンパイル・コマンドに対して指定されている場合は、現行ロケールの LC_UNI_CTYPE カテゴリーにも影響を受ける可能性があります。この関数は、コンパイル・コマンドに対して LOCALETYPE(*CLD) が指定されている場合には使用できません。詳細については、 549 ページの『CCSID およびロケールの理解』を参照してください。

統合ファイル・システム・インターフェース: この関数は、コンパイル・コマンドに対して SYSIFCOPT(*NOIFSIO) が指定されている場合には使用できません。

ワイド文字関数: 詳細については、 553 ページの『ワイド文字』 を参照してください。

説明

fputwc() 関数はワイド文字 *wc* を、*stream* で指定される出力ストリームへ、現在位置で書き込みます。また、ファイル位置標識を正しく進めます。ファイルが位置指定要求をサポートできない場合、またはストリームが追加モードでオープンされた場合は、文字はストリームへ追加されます。

非ワイド文字関数を同じストリーム上で fputwc() 関数と共に使用すると、予期しない振る舞いが生じる結果となります。 fputwc() 関数を呼び出した後、ストリームの読み取り関数を呼び出す前にバッファーを削除するか、ストリーム・ポインターの位置を変更します。ストリームを読み取った後、EOF に達しない限り、fputwc() 関数を呼び出す前にバッファーを削除するか、ストリーム・ポインターの位置を変更します。

注: 同じストリームに対する連続した操作の間で現行ロケールが変更された場合は、予期しない結果が発生することがあります。

戻り値

fputwc() 関数は、書き込まれるワイド文字を戻します。書き込みエラーが発生すると、ストリームのエラー標識が設定され、fputwc() 関数は WEOF を戻します。ワイド文字からマルチバイト文字への変換中にエンコード・エラーが発生した場合は、fputwc() は errno に EILSEQ を設定し、WEOF を戻します。

putwc() の errno 値について詳しくは、124 ページの『fputc() ― 文字の書き込み』を参照してください。

fputwc() の使用例

次の例はファイルをオープンし、fputwc() 関数を使用してワイド文字をファイルに書き込みます。

```
#include <stdio.h>
#include <stdlib.h>
#include <wchar.h>
#include <errno.h>

int main(void)
{
   FILE    *stream;
   wchar_t *wcs = L"A character string.";
   int     i;

   if (NULL == (stream = fopen("fputwc.out", "w")))
   {
      printf("Unable to open: ¥"fputwc.out¥".¥n");
      exit(1);
   }

   for (i = 0; wcs[i] != L'¥0'; i++) {
      errno = 0;
      if (WEOF == fputwc(wcs[i], stream)) {
         printf("Unable to fputwc() the wide character.¥n"
                "wcs[%d] = 0x%.4lx¥n", i, wcs[i]);
         if (EILSEQ == errno)
            printf("An invalid wide character was encountered.¥n");
         exit(1);
      }
   }
   fclose(stream);
   return 0;

   /***************************************************************
      The output file fputwc.out should contain:

      A character string.
   ***************************************************************/
}
```

関連情報

- 107 ページの『fgetwc() ― ストリームのワイド文字の読み取り』
- 124 ページの『fputc() ― 文字の書き込み』
- 128 ページの『fputwc() ― 文字の書き込み』
- 249 ページの『putc() - putchar() ― 文字の書き込み』
- 254 ページの『putwchar() ― ワイド文字の stdout への書き込み』
- 252 ページの『putwc() ― ワイド文字の書き込み』
- 17 ページの『<stdio.h>』
- 20 ページの『<wchar.h>』

fputws() ― ワイド文字ストリングの書き込み

フォーマット

```
#include <wchar.h>
#include <stdio.h>
int fputws(const wchar_t *wcs, FILE *stream);
```

言語レベル: XPG4

スレッド・セーフ: はい。

ロケール依存: この関数の振る舞いは、現行ロケールの LC_CTYPE カテゴリーの影響を受ける可能性があります。また、この振る舞いは、LOCALETYPE(*LOCALEUCS2) または LOCALETYPE(*LOCALEUTF) がコンパイル・コマンドに対して指定されている場合は、現行ロケールの LC_UNI_CTYPE カテゴリーにも影響を受ける可能性があります。この関数は、コンパイル・コマンドに対して LOCALETYPE(*CLD) が指定されている場合には使用できません。詳細については、 549 ページの『CCSID およびロケールの理解』を参照してください。

統合ファイル・システム・インターフェース: この関数は、コンパイル・コマンドに対して SYSIFCOPT(*NOIFSIO) が指定されている場合には使用できません。

ワイド文字関数: 詳細については、 553 ページの『ワイド文字』 を参照してください。

説明

`fputws()` 関数はワイド文字ストリング *wcs* を *stream* に書き込みます。最後の NULL ワイド文字は書き込みません。

非ワイド文字関数を同じストリーム上で `fputws()` 関数と共に使用すると、予期しない振る舞いが生じる結果となります。 `fputws()` 関数を呼び出した後、ストリームの読み取り関数を呼び出す前にバッファーをフラッシュするか、ストリーム・ポインターの位置を変更してください。読み取り操作の後、EOF に達しない限り、`fputws()` 関数を呼び出す前にバッファーをフラッシュするか、ストリーム・ポインターの位置を変更してください。

注: 同じストリームに対する連続した操作の間で現行ロケールが変更された場合は、予期しない結果が発生することがあります。

戻り値

`fputws()` 関数は正常に終了した場合は、負の数以外の数を戻します。書き込みエラーが発生すると、ストリームのエラー標識が設定され、`fputws()` 関数は -1 を戻します。ワイド文字からマルチバイト文字への変換中にエンコード・エラーが発生した場合は、`fputws()` 関数は errno に EILSEQ を設定し、-1 を戻します。

`fputws()` の errno 値について詳しくは、124 ページの『fputc() ― 文字の書き込み』を参照してください。

`fputws()` の使用例

次の例はファイルをオープンし、`fgetws()` 関数を使用してワイド文字ストリングをファイルに書き込みます。

```
#include <stdio.h>
#include <stdlib.h>
#include <wchar.h>
#include <errno.h>

int main(void)
{
   FILE    *stream;
   wchar_t *wcs = L"This test string should not return -1";

   if (NULL == (stream = fopen("fputws.out", "w"))) {
      printf("Unable to open: ¥"fputws.out¥".¥n");
      exit(1);
   }

   errno = 0;
   if (EOF == fputws(wcs, stream)) {
      printf("Unable to complete fputws() function.¥n");
      if (EILSEQ == errno)
         printf("An invalid wide character was encountered.¥n");
      exit(1);
   }
   fclose(stream);
   return 0;

   /*************************************************************
      The output file fputws.out should contain:

      This test string should not return -1
   *************************************************************/
}
```

関連情報

- 109 ページの『fgetws() ― ストリームのワイド文字のストリングの読み取り』
- 127 ページの『fputs() ― ストリングの書き込み』
- 128 ページの『fputwc() ― 文字の書き込み』
- 251 ページの『puts() ― ストリングの書き込み』
- 17 ページの『<stdio.h>』
- 20 ページの『<wchar.h>』

fread() ― 項目の読み取り

フォーマット

```
#include <stdio.h>
size_t fread(void *buffer, size_t size, size_t count, FILE *stream);
```

言語レベル: ANSI

スレッド・セーフ: はい。

説明

`fread()` 関数は入力 *stream* から *size* 長の *count* 項目までを読み取り、それらを指定された *buffer* に格納します。ファイル内の位置は、読み取られたバイト数だけ前に進みます。

戻り値

fread() 関数は、正常に読み取られた完全な項目の数を戻します。これは、エラーが発生した場合、または *count* に達する前にファイル終了になった場合は *count* よりも小さくなる場合があります。*size* または *count* が 0 の場合、fread() 関数は 0 を戻し、配列の内容とストリームの状態は変更されないままになります。

errno の値は、次のいずれかに設定されます。

値 **意味**

EGETANDPUT
書き込み操作の後に許可されていない読み取り操作が発生しました。

ENOREC
レコードが見つかりません。

ENOTREAD
ファイルは読み取り操作用にオープンされていません。

ERECIO
ファイルはレコードの入出力用にオープンしています。

ESTDIN
stdin をオープンできません。

ETRUNC
操作で切り捨てが発生しました。

EIOERROR
リカバリー不能な入出力エラーが発生しました。

EIORECERR
リカバリー可能な入出力エラーが発生しました。

ferror() および feof() 関数を使用して読み取りエラーとファイル終了とを区別します。

レコード入力に対して fread() を使用するときには、バイト数を取得するために、*size* を 1 に、および *count* を予想されるレコード最大長に設定します。レコード長が分からない場合は、*size* を 1 に、*count* を大きな値に設定する必要があります。レコード入出力を使用する場合は、一度に 1 レコードしか読み取ることができません。

fread() の使用例

次の例は NUM_ALPHA 文字をファイル *myfile* から読み出します。fread() または fopen() でエラーが発生すると、メッセージが出力されます。

```
#include <stdio.h>

#define NUM_ALPHA  26

int main(void)
{
  FILE * stream;
  int num;       /* number of characters read from stream */

  /* Do not forget that the '¥0' char occupies one character too! */
  char buffer[NUM_ALPHA + 1];

  if (( stream = fopen("mylib/myfile", "r"))!= NULL )
  {
    memset(buffer, 0, sizeof(buffer));
    num = fread( buffer, sizeof( char ), NUM_ALPHA, stream );
    if ( num ) {  /* fread success */
      printf( "Number of characters has been read = %i¥n", num );
      printf( "buffer = %s¥n", buffer );
      fclose( stream );
    }
    else {  /* fread failed */
      if ( ferror(stream) )    /* possibility 1 */
        perror( "Error reading myfile" );
      else if ( feof(stream))  /* possibility 2 */
        perror( "EOF found" );
    }
  }
  else
    perror( "Error opening myfile" );

}
```

関連情報

- 99 ページの『feof() — ファイル終了標識のテスト』
- 100 ページの『ferror() — 読み取り/書き込みエラーのテスト』
- 114 ページの『fopen() — ファイルのオープン』
- 152 ページの『fwrite() — 項目の書き込み』
- 17 ページの『<stdio.h>』

free() — ストレージ・ブロックの解放

フォーマット

```
#include <stdlib.h>
void free(void *ptr);
```

言語レベル: ANSI

スレッド・セーフ: はい。

説明

free() 関数はストレージのブロックを解放します。*ptr* 引数は、以前に calloc()、malloc()、realloc()、_C_TS_calloc()、_C_TS_malloc()、_C_TS_realloc()、または _C_TS_malloc64() 関数の呼び出しで予約されたブロックを指します。解放されるバイト数は、ストレージのブロックを予約 (realloc() 関数の場合は再割り振り) するときに指定したバイト数です。*ptr* が NULL の場合、free() は戻るだけです。

注:

1. すべてのヒープ・ストレージは、呼び出しルーチンの活動化グループと関連付けられます。したがって、ストレージの割り振りと割り振り解除は同じ活動化グループ内で行ってください。 1 つの活動化グループ内でヒープ・ストレージを割り振ったり、異なる活動化グループからそのストレージを割り振り解除したりすることは、有効ではありません。活動化グループについて詳しくは、「*ILE Concepts*」のマニュアルを参照してください。
2. `calloc()`、`malloc()`、または `realloc()` によって割り振られなかったストレージのブロック (または以前に解放されたストレージ) を解放しようとすると、その後ストレージを予約するときに影響が生じて予期しない結果につながることがあります。 ILE バインド可能 API CEEGTST で割り振られたストレージは `free()` で解放できます。

| C ソース・コードを変更せずに、単一レベル・ストア・ストレージの代わりに**テラスペース**・ストレージ
| を使用する場合、コンパイラー・コマンドで TERASPACE(*YES *TSIFC) パラメーターを指定します。こ
| れにより、`free()` ライブラリー関数が `_C_TS_free()` (テラスペース・ストレージでの `free()` ライブラリ
| ー関数のカウンター・パート) にマップされます。

| **注:** C2M1211 メッセージまたは C2M1212 メッセージが `free()` 関数から生成された場合、詳細について
| は、 579 ページの『C2M1211/C2M1212 メッセージの問題の診断』を参照してください。

戻り値

戻り値はありません。

`free()` の使用例

次の例は `calloc()` 関数を使用して *x* 配列エレメントのためにストレージを割り振った後、`free()` 関数を呼び出してこれらを解放します。

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
  long * array;    /* start of the array            */
  long * index;    /* index variable                */
  int    i;        /* index variable                */
  int  num;        /* number of entries of the array */

  printf( "Enter the size of the array¥n" );
  scanf( "%i", &num );

  /* allocate num entries */
  if ( (index = array = calloc( num, sizeof( long ))) != NULL )
  {
    for ( i = 0; i < num; ++i )           /* put values in array    */
       *index++ = i;                      /* using pointer notation */

    free( array );                        /* deallocates array      */
  }
  else
  { /* Out of storage */
    perror( "Error: out of storage" );
    abort();
  }
}
```

関連情報

- 58 ページの『calloc() ― ストレージの予約と初期化』

- 69 ページの『_C_Quickpool_Debug() ― 高速プール・メモリー・マネージャー特性の変更』
- 71 ページの『_C_Quickpool_Init() ― 高速プール・メモリー・マネージャーの初期化』
- 73 ページの『_C_Quickpool_Report() ― 高速プール・メモリー・マネージャー・レポートの生成』
- 564 ページの『ヒープ・メモリー』
- 203 ページの『malloc() ― ストレージ・ブロックの予約』
- 276 ページの『realloc() ― 予約ストレージ・ブロック・サイズの変更』
- 18 ページの『<stdlib.h>』

freopen() ― オープン・ファイルのリダイレクト

フォーマット

```
#include <stdio.h>
FILE *freopen(const char *filename, const char *mode, FILE *stream);
```

言語レベル: ANSI

スレッド・セーフ: はい。

説明

`freopen()` 関数は、現在 *stream* と関連付けられているファイルをクローズし、*filename* により指定されるファイルに *stream* を再割り当てします。`freopen()` 関数は *stream* と関連付けられている新規ファイルを指定の *mode* (ファイルに対して要求されたアクセス・タイプを指定する文字ストリング) でオープンします。また、`freopen()` 関数を使用して標準のストリーム・ファイル `stdin`、`stdout`、および `stderr` を指定したファイルにリダイレクトできます。

データベース・ファイルでは、*filename* が空ストリングの場合に `freopen()` 関数はストリームをクローズしてから、これを新しいファイルまたはデバイスに再割り当てしないで、新しいオープン・モードで再オープンします。例えば、ファイル名を指定しないで `freopen()` 関数を使用し、ストリームをリダイレクトしないで標準ストリームのモードをテキストからバイナリーに変更することができます。

```
fp = freopen("", "rb", stdin);
```

同じメソッドを使用してモードをバイナリーからテキストへ戻すこともできます。

SYSIFCOPT(*IFSIO) で作成したモジュールの空ストリングとして、`freopen()` 関数を *filename* と共に使用することはできません。

戻り値

`freopen()` 関数は、新しくオープンしたストリームを指すポインターを戻します。エラーが発生すると、`freopen()` 関数はオリジナル・ファイルをクローズし、NULL ポインター値を戻します。

errno の値は、次のいずれかに設定されます。

値 **意味**

EBADF
ファイル・ポインター、または記述子が有効ではありません。

EBADMODE
指定されたファイル・モードが無効です。

EBADNAME
: 指定されたファイル名が無効です。

ENOENT
: ファイルまたはライブラリーがありません。

ENOTOPEN
: ファイルはオープンされていません。

EIOERROR
: リカバリー不能な入出力エラーが発生しました。

EIORECERR
: リカバリー可能な入出力エラーが発生しました。

freopen() の使用例

次の例では *stream1* データ・ストリームをクローズし、そのストリーム・ポインターを再割り当てします。 *stream1* および *stream2* は同じ値を持ちますが、必ずしも *stream* と同じ値ではありません。

```
#include <stdio.h>
#define  MAX_LEN  100

int main(void)
{
   FILE *stream, *stream1, *stream2;
   char line[MAX_LEN], *result;
   int  i;

   stream = fopen("mylib/myfile","r");
   if ((result = fgets(line,MAX_LEN,stream)) != NULL)
       printf("The string is %s¥n", result);

   /* Change all spaces in the line to '*'. */
   for (i=0; i<=sizeof(line); i++)
     if (line[i] == ' ')
         line[i] = '*';

   stream1 = stream;
   stream2 = freopen("", "w+", stream1);
   fputs( line, stream2 );
   fclose( stream2);
}
```

関連情報

- 95 ページの『fclose() ― ストリームのクローズ』
- 114 ページの『fopen() ― ファイルのオープン』
- 17 ページの『<stdio.h>』

---

frexp() ― 浮動小数点値の分離

フォーマット

```
#include <math.h>
double frexp(double x, int *expptr);
```

言語レベル: ANSI

スレッド・セーフ: はい。

説明

frexp() 関数は浮動小数点値 *x* を小数部の項 *m* と指数部の項 *n* に分けます。これは、$x=m*2^n$ のように行われ、*m* の絶対値は 0.5 以上かつ 1.0 未満、または 0 です。frexp() 関数は整数の指数 *n* を *expptr* が指すロケーションに格納します。

戻り値

frexp() 関数は小数部の項 *m* を戻します。*x* が 0 の場合、frexp() は小数部指数部の両方に対して 0 を戻します。小数部には引数 *x* と同じ符号が付きます。frexp() 関数の結果は範囲エラーを持つことができません。

frexp() の使用例

次の例は *x* の浮動小数点値 16.4 をその小数部 0.5125 と指数部 5 に分けます。小数部を *y* に、指数部を *n* に格納します。

```
#include <math.h>
#include <stdio.h>

int main(void)
{
   double x, m;
   int n;

   x = 16.4;
   m = frexp(x, n);

   printf("The mantissa is %lf and the exponent is %d¥n", m, n);
}

/*******************  Output should be similar to:  ***************

The mantissa is 0.512500 and the exponent is 5
*/
```

関連情報

- 185 ページの『ldexp() ― 2 のべき乗の乗算』
- 232 ページの『modf() ― 浮動小数点値の分離』
- 9 ページの『<math.h>』

fscanf() ― フォーマット済みデータの読み取り

フォーマット

```
#include <stdio.h>
int fscanf (FILE *stream, const char *format-string, argument-list);
```

言語レベル: ANSI

スレッド・セーフ: はい。

ロケール依存: この関数の振る舞いは、現行ロケールの LC_CTYPE カテゴリーおよび LC_NUMERIC カテゴリーの影響を受ける可能性があります。また、この振る舞いは、LOCALETYPE(*LOCALEUCS2) または LOCALETYPE(*LOCALEUTF) がコンパイル・コマンドに対して指定されている場合は、現行ロケール

の LC_UNI_CTYPE カテゴリーの影響を受ける可能性もあります。詳細については、 549 ページの『CCSID およびロケールの理解』を参照してください。

説明

fscanf() 関数は、指定された *stream* の現在位置から、*argument-list* のエントリー (存在する場合) で指定されたロケーションにデータを読み取ります。*argument-list* の各エントリーは、*format-string* の型指定子に対応する型を持つ変数を指すポインターでなければなりません。

format-string は入力フィールドの解釈を制御し、scanf() 関数の *format-string* 引数と同じ形式と機能を持ちます。

戻り値

fscanf() 関数は正常に変換し、割り当てたフィールド数を戻します。戻り値に、fscanf() 関数が読み取ったが割り当てなかったフィールドは含まれません。

変換の前に入力障害が起こった場合、戻り値は EOF です。

fscanf() の使用例

次の例はファイル *myfile* をオープンして読み取り、ストリング、long 型整数値、文字、および浮動小数点値についてこのファイルを走査します。

```
#include <stdio.h>

#define  MAX_LEN  80

int main(void)
{
   FILE *stream;
   long l;
   float fp;
   char s[MAX_LEN + 1];
   char c;

   stream = fopen("mylib/myfile", "r");

   /* Put in various data. */

   fscanf(stream, "%s", &s [0]);
   fscanf(stream, "%ld", &l);
   fscanf(stream, "%c", &c);
   fscanf(stream, "%f", &fp);

   printf("string = %s¥n", s);
   printf("long double = %ld¥n", l);
   printf("char = %c¥n", c);
   printf("float = %f¥n", fp);
}

/***************  If myfile contains  ************************
**************** abcdefghijklmnopqrstuvwxyz 343.2 ***********
**********************  expected output is:  *********************

string = abcdefghijklmnopqrstuvwxyz
long double = 343
char = .
float = 2.000000
*/
```

関連情報

- 122 ページの『fprintf() ― フォーマット済みデータのストリームへの書き込み』
- 153 ページの『fwscanf() ― ワイド文字を使用したストリームからのデータの読み取り』
- 344 ページの『scanf() ― データの読み取り』
- 371 ページの『sscanf() ― データの読み取り』
- 425 ページの『swscanf() ― ワイド文字データの読み取り』
- 528 ページの『wscanf() ― ワイド文字書式ストリングを使用したデータの読み取り』
- 17 ページの『<stdio.h>』

fseek() ― fseeko() ― ファイル位置の位置変更

フォーマット

```
#include <stdio.h>
int fseek(FILE *stream, long int offset, int origin);
int fseeko(FILE *stream, off_t offset, int origin);
```

言語レベル: ANSI

スレッド・セーフ: はい。

統合ファイル・システム・インターフェース: `fseeko()` 関数は、コンパイル・コマンドに対して SYSIFCOPT(*NOIFSIO) が指定されている場合は使用できません。

説明

`fseek()` および `fseeko()` 関数は *stream* と関連付けられているファイルの現在位置を、ファイル内の新しいロケーションに変更します。 *stream* での次の操作は、新しい位置で行われます。更新用にオープンされた *stream* では、次の操作は読み取り、または書き込み操作のいずれかです。

`fseeko()` 関数は、オフセット引数が型 `off_t` である点を除き、`fseek()` と同じです。

origin は、<stdio.h> で定義された以下の定数のいずれかでなければなりません。

オリジン
: **定義**

SEEK_SET
: ファイルの始め

SEEK_CUR
: ファイル・ポインターの現在位置

SEEK_END
: ファイルの終わり

バイナリー・ストリームでは、ファイルの終わりを越えて位置を変更することもできます。ファイルの先頭よりも前に位置指定しようとすると、エラーが生じます。正常終了した場合は、*origin* が SEEK_END である場合でも、`fseek()` または `fseeko()` 関数はファイル終了標識をクリアし、同じストリームに対する直前の `ungetc()` 関数の影響を取り消します。

注: テキスト・モードでオープンされたストリームでは、`fseek()` 関数および `fseeko()` 関数の使用が制限されています。これは、一部のシステム変換 (復帰改行と改行間の変換など) が予期できない結果を生

み出すことがあるからです。テキスト・モードでオープンされたストリーム上での機能が信頼できる fseek() および fseeko() 操作は、発信元のすべての値に対して 0 のオフセットでシークするか、ftell() または ftello() 関数への呼び出しから戻されたオフセット値で、ファイルの先頭からシークすることのみです。ftell() および ftello() 関数への呼び出しは、それらの関数の制限に従います。

戻り値

fseek() または fseeko 関数は、正常にポインターを移動した場合には 0 を戻します。ゼロ以外の戻り値はエラーを示します。端末やプリンターのような、シークできないデバイスでは、戻り値はゼロ以外の値です。

errno の値は、次のいずれかに設定されます。

値　　意味

EBADF
ファイル・ポインターまたは記述子が無効です。

EBADSEEK
シーク操作のオフセットが無効です。

ENODEV
誤ったデバイスに対して操作が試行されました。

ENOTOPEN
ファイルはオープンされていません。

ERECIO
ファイルはレコードの入出力用にオープンしています。

ESTDERR
stderr をオープンできません。

ESTDIN
stdin をオープンできません。

ESTDOUT
stdout をオープンできません。

EIOERROR
リカバリー不能な入出力エラーが発生しました。

EIORECERR
リカバリー可能な入出力エラーが発生しました。

fseek() 関数および fseeko() 関数は、type=record でオープンされたファイルではサポートされていません。

fseek() の使用例

次の例は読み取り用にファイル myfile をオープンします。入力操作を実行した後、fseek() はファイル・ポインターをファイルの先頭に移動します。

```
#include <stdio.h>
#define  MAX_LEN  10

int main(void)
{
   FILE *stream;
   char buffer[MAX_LEN + 1];
   int  result;
   int  i;
   char ch;

   stream = fopen("mylib/myfile", "r");
   for (i = 0; (i  < (sizeof(buffer)-1) &&
       ((ch = fgetc(stream)) != EOF) && (ch != '¥n')); i++)
          buffer[i] = ch;

   result = fseek(stream, 0L, SEEK_SET);  /* moves the pointer to the */
                                          /* beginning of the file    */
   if (result == 0)
      printf("Pointer successfully moved to the beginning of the file.¥n");
   else
      printf("Failed moving pointer to the beginning of the file.¥n");
}
```

関連情報

- 144 ページの『ftell() ― ftello() ― 現在位置の取得』
- 104 ページの『fgetpos() ― ファイル位置の取得』
- 『fsetpos() ― ファイル位置の設定』
- 288 ページの『rewind() ― 現在のファイル位置の調整』
- 439 ページの『ungetc() ― 入力ストリームへの文字のプッシュ』
- 140 ページの『fseek() ― fseeko() ― ファイル位置の位置変更』
- 17 ページの『<stdio.h>』

fsetpos() ― ファイル位置の設定

フォーマット

```
#include <stdio.h>
int fsetpos(FILE *stream, const fpos_t *pos);
```

言語レベル: ANSI

スレッド・セーフ: はい。

説明

`fsetpos()` 関数は、*stream* と関連付けられたファイル位置を、*pos* で指された値に応じて、ファイル内の新しい位置へ移動します。*pos* の値は、`fgetpos()` ライブラリー関数の直前の呼び出しによって取得されています。

正常終了した場合、`fsetpos()` はファイル終了標識をクリアし、直前の `ungetc()` 関数の同じストリームに対する影響を取り消します。

`fsetpos()` 呼び出しの後、ストリームに対する更新モードでの次の操作は、入力または出力であることが可能です。

戻り値

`fsetpos()` がファイルの現在位置を正常に変更した場合、値 0 を戻します。ゼロ以外の戻り値はエラーを示します。

errno の値は、次のいずれかに設定されます。

値 **意味**

EBADF
ファイル・ポインターまたは記述子が無効です。

EBADPOS
指定された位置が無効です。

EINVAL
引数に指定された値が不正です。プログラムを *IFSIO でコンパイルし、かつ、QSYS ファイル・システムで操作している場合は、この errno を受信することがあります。例えば、次のようになります。`"/qsys.lib/qtemp.lib/myfile.file/mymem.mbr"`

ENODEV
誤ったデバイスに対して操作が試行されました。

ENOPOS
指定された位置にレコードがありません。

ERECIO
ファイルはレコードの入出力用にオープンしています。

ESTDERR
`stderr` をオープンできません。

ESTDIN
`stdin` をオープンできません。

ESTDOUT
`stdout` をオープンできません。

EIOERROR
リカバリー不能な入出力エラーが発生しました。

EIORECERR
リカバリー可能な入出力エラーが発生しました。

`fsetpos()` 関数は、type=record でオープンされたファイルではサポートされていません。また、`fsetpos()` 関数は以下の場合のみ、ファイルの先頭への位置の設定をサポートすることができます。

- プログラムが *IFSIO でコンパイルされている
- QSYS ファイル・システム内のファイルに対して操作している。

`fsetpos()` の使用例

次の例は読み取り用にファイル mylib/myfile をオープンします。入力操作を実行した後、`fsetpos()` はファイル・ポインターをファイルの先頭に移動し、先頭のバイトを再読み取りします。

```
#include  <stdio.h>

FILE *stream;

int main(void)
{
   int retcode;
   fpos_t pos;
   char ptr[20];  /* existing file 'mylib/myfile' has 20 byte records */
   int  i;

   /* Open file, get position of file pointer, and read first record */

   stream = fopen("mylib/myfile", "rb");
   fgetpos(stream,Point-of-Sale);
   if (!fread(ptr,sizeof(ptr),1,stream))
       perror("fread error");
   else printf("1st record: %s¥n", ptr);

   /* Perform another read operation on the second record       */
   /*  - the value of 'pos' changes                              */
   if (!fread(ptr,sizeof(ptr),1,stream))
       perror("fread error");
   else printf("2nd record: %s¥n", ptr);

   /* Re-set pointer to start of file and re-read first record  */
   fsetpos(stream,Point-of-Sale);
   if (!fread(ptr,sizeof(ptr),1,stream))
       perror("fread error");
   else printf("1st record again: %s¥n", ptr);

   fclose(stream);
}
```

関連情報

- 104 ページの『fgetpos() ― ファイル位置の取得』
- 140 ページの『fseek() ― fseeko() ― ファイル位置の位置変更』
- 『ftell() ― ftello() ― 現在位置の取得』
- 288 ページの『rewind() ― 現在のファイル位置の調整』
- 17 ページの『<stdio.h>』

ftell() ― ftello() ― 現在位置の取得

フォーマット

```
#include <stdio.h>
long int ftell(FILE *stream);
off_t ftello(FILE *stream);
```

言語レベル: ANSI

スレッド・セーフ: はい。

統合ファイル・システム・インターフェース: `ftello()` 関数は、コンパイル・コマンドに対して SYSIFCOPT(*NOIFSIO) が指定されている場合は使用できません。

説明

ftell() および ftello() 関数は、*stream* と関連付けられたファイルの現在位置を検索します。固定長バイナリー・ファイルの場合、戻される値は *stream* の先頭に相対的なオフセットです。

QSYS ライブラリー・システムのファイルの場合、ftell() および ftello() 関数は、固定形式のバイナリー・ファイルに対しては相対値を、他のファイル・タイプに対してはエンコード値を戻します。このエンコード値は、ファイルの先頭以外の位置に対する fseek() 関数および fseeko() 関数の呼び出しで使用される必要があります。

戻り値

ftell() 関数および ftello() 関数はファイルの現在位置を戻します。エラーの場合、ftell() および ftello() はそれぞれ -1, long へキャスト および off_t を戻し、errno にゼロ以外の値を設定します。

errno の値は、次のいずれかに設定されます。

値　　意味

ENODEV
誤ったデバイスに対して操作が試行されました。

ENOTOPEN
ファイルはオープンされていません。

ENUMMBRS
ファイルは複数メンバー処理用にオープンしています。

ENUMRECS
レコードが多すぎます。

ERECIO
ファイルはレコードの入出力用にオープンしています。

ESTDERR
stderr をオープンできません。

ESTDIN
stdin をオープンできません。

ESTDOUT
stdout をオープンできません。

EIOERROR
リカバリー不能な入出力エラーが発生しました。

EIORECERR
リカバリー可能な入出力エラーが発生しました。

ftell() 関数および ftello() 関数は、type=record でオープンされたファイルではサポートされていません。

ftell() の使用例

次の例は読み取り用にファイル **mylib/myfile** をオープンします。バッファーの半分を埋めるに十分な文字を読み取り、ストリームおよびバッファー内の位置を出力します。

```
#include  <stdio.h>

#define NUM_ALPHA  26
#define NUM_CHAR    6

int main(void)
{
  FILE * stream;
  int i;
  char ch;

  char buffer[NUM_ALPHA];
  long position;

  if (( stream = fopen("mylib/myfile", "r")) != NULL )
  {
    /* read into buffer */
    for ( i = 0; ( i  < NUM_ALPHA/2 ) && ((buffer[i] = fgetc(stream)) != EOF ); ++i )
        if (i==NUM_CHAR-1)  /* We want to be able to position the   */
                                 /* file pointer to the character in */
                                 /* position NUM_CHAR                */
           position = ftell(stream);

    buffer[i] = '¥0';
  } printf("Current file position is %d¥n", position);
    printf("Buffer contains: %s¥n", buffer);
}
```

関連情報

- 140 ページの『fseek() ― fseeko() ― ファイル位置の位置変更』
- 104 ページの『fgetpos() ― ファイル位置の取得』
- 114 ページの『fopen() ― ファイルのオープン』
- 142 ページの『fsetpos() ― ファイル位置の設定』
- 17 ページの『<stdio.h>』

fwide() ― ストリーム指向の決定

フォーマット

```
#include <stdio.h>
#include <wchar.h>
int fwide(FILE *stream, int mode);
```

言語レベル: ANSI

スレッド・セーフ: はい。

ロケール依存: この関数は、コンパイル・コマンドに対して LOCALETYPE(*CLD) が指定されている場合には使用できません。

統合ファイル・システム・インターフェース: この関数は、コンパイル・コマンドに対して SYSIFCOPT(*NOIFSIO) が指定されている場合には使用できません。

説明

`fwide()` 関数は、*stream* によって指定されたストリームの指向を決定します。*mode* が 0 より大きい場合、`fwide()` 関数はまず、ストリームをワイド指向にしようとします。 *mode* が 0 より小さい場合、`fwide()` 関数はまず、ストリームをバイト指向にしようとします。それ以外の場合は、*mode* は 0 で、`fwide()` 関数はストリームの指向を変更しません。

注: ストリームの指向がすでに決定されている場合、`fwide()` 関数はその指向を変更しません。

戻り値

呼び出し後にストリームがワイド指向であれば、`fwide()` 関数は 0 より大きな値を戻します。ストリームがバイト指向であれば、0 より小さい値を戻します。ストリームに指向がなければ、0 を戻します。

`fwide()` の使用例

```
#include <stdio.h>
#include <math.h>
#include <wchar.h>

void check_orientation(FILE *stream)
{
   int rc;
   rc = fwide(stream,0);      /* check the orientation */
   if (rc<0) {
      printf("Stream has byte orientation.¥n");
   } else if (rc>0)  {
      printf("Stream has wide orientation.¥n");
   } else {
      printf("Stream has no orientation.¥n");
   }
   return;
}

int main(void)
{
   FILE *stream;
   /* Demonstrate that fwide can be used to set the orientation,
      but cannot change it once it has been set.   */
   stream = fopen("test.dat","w");
   printf("After opening the file: ");
   check_orientation(stream);
   fwide(stream, -1);        /* Make the stream byte oriented */
   printf("After fwide(stream, -1): ");
   check_orientation(stream);
   fwide(stream,  1);        /* Try to make the stream wide oriented */
   printf("After fwide(stream,  1): ");
   check_orientation(stream);
   fclose(stream);
   printf("Close the stream¥n");
   /* Check that a wide character output operation sets the orientation
      as expected.   */
   stream = fopen("test.dat","w");
   printf("After opening the file: ");
   check_orientation(stream);
   fwprintf(stream, L"pi = %.5f¥n", 4* atan(1.0));
   printf("After fwprintf( ): ");
   check_orientation(stream);
   fclose(stream);
   return 0;
   /*******************************************************************
      The output should be similar to :
      After opening the file: Stream has no orientation.
      After fwide(stream, -1): Stream has byte orientation.
      After fwide(stream,  1): Stream has byte orientation.
      Close the stream
      After opening the file: Stream has no orientation.
      After fwprintf( ): Stream has wide orientation.
   *******************************************************************/
}
```

関連情報

- 107 ページの『fgetwc() ― ストリームのワイド文字の読み取り』
- 109 ページの『fgetws() ― ストリームのワイド文字のストリングの読み取り』
- 128 ページの『fputwc() ― 文字の書き込み』
- 130 ページの『fputws() ― ワイド文字ストリングの書き込み』
- 17 ページの『<stdio.h>』
- 20 ページの『<wchar.h>』

fwprintf() ― ワイド文字としてのデータのフォーマット設定とストリームへの書き込み

フォーマット

```
#include <stdio.h>
#include <wchar.h>
int fwprintf(FILE *stream, const wchar_t *format, argument-list);
```

言語レベル: ANSI

スレッド・セーフ: はい。

ロケール依存: この関数の振る舞いは、LOCALETYPE(*LOCALEUCS2) または LOCALETYPE (*LOCALEUTF) がコンパイル・コマンドに対して指定されている場合は、現行ロケールの LC_CTYPE および LC_NUMERIC カテゴリーの影響を受ける可能性があります。この関数は、コンパイル・コマンドに対して LOCALETYPE(*CLD) が指定されている場合には使用できません。詳細については、 549 ページの『CCSID およびロケールの理解』を参照してください。

統合ファイル・システム・インターフェース: この関数は、コンパイル・コマンドに対して SYSIFCOPT(*NOIFSIO) が指定されている場合には使用できません。

ワイド文字関数: 詳細については、 553 ページの『ワイド文字』 を参照してください。

説明

fwprintf() 関数は、*format* により示されたワイド・ストリングの制御下で、*stream* により示されたストリームに出力を書き込みます。書式ストリングは、後続の引数が出力用にどのように変換されるかを指定します。

fwprintf() 関数は、書式中の対応するワイド文字指定子に応じて、*argument-list* 内の各エントリーを変換します。

書式に対して引数が不十分な場合、振る舞いは予期できません。引数が残っているのに、書式がすべて使用されている場合、fwprintf() 関数は余分の引数を評価しますが、それ以外の場合は無視します。 fwprintf() 関数は、書式ストリングの終わりになると戻ります。

書式はゼロ以上のディレクティブで構成されています。一般にはワイド文字 (% 以外) および変換指定です。変換指定は、書式ストリングでワイド文字ストリングにより置換されたかのように処理されます。ワイド文字ストリングはゼロ以上の後続の引数を取り出し、該当する場合は、対応する変換指定子に応じてその引数を変換した結果です。 fwprintf() 関数は次に、展開されたワイド文字書式ストリングを出力ストリームに書き込みます。

fwprintf() 関数の書式には、printf() の書式ストリングと同じ形式と機能がありますが、以下の点が異なります。

- %c (l 接頭部なし) は、btowc() 関数を呼び出して変換したかのように、整数引数を wchar_t に変換します。
- %s (l 接頭部なし) は、mbrtowc() 関数を呼び出して変換したかのように、マルチバイト文字の配列を wchar_t の配列に変換します。精度に対して、より短い出力が指定されていない限り、配列は終了 NULL 文字に達するまで (終了 NULL 文字は含まれません) 書き込まれます。
- %ls および %S は wchar_t の配列を書き込みます。この配列は、精度に対してより短い出力が指定されていない限り、終了 NULL 文字まで (終了 NULL 文字は含まれません) 書き込まれます。

- %c、%s、%ls、および %S に対して指定された任意の幅すなわち精度は、バイト数ではなく、文字数を示します。

変換指定が無効な場合、その振る舞いは予期できません。

任意の引数またはポイントが和集合または集合の場合 (%s 変換を使用した char 型配列、 %ls 変換を使用した wchar_t 型配列、または %p 変換を使用したポインターを除く)、振る舞いは予期できません。

いかなる場合も、存在していないフィールド幅または短いフィールド幅で、フィールドが切り捨てられることはありません。変換結果がフィールド幅よりも広い場合、そのフィールドは変換結果を含めるように拡張されます。

注: ワイド文字を書き込む場合、ファイルはバイナリー・モードでオープンするか、**o_ccsid** または **codepage** パラメーターでオープンする必要があります。これにより、ワイド文字に対して変換が発生しないことが保証されます。

戻り値

fwprintf() 関数は、送信されたワイド文字の数を戻します。出力エラーが発生すると、負の値を戻します。

fwprintf() の使用例

```
#include <stdio.h>
#include <wchar.h>
#include <locale.h>
int count [10] = {1, 5, 8, 3, 0, 3, 5, 6, 8, 10};
int main(void)
{
   int i,j;
   FILE *stream;                      /* Open the stream for writing */
   if (NULL == (stream = fopen("/QSYS.LIB/LIB.LIB/WCHAR.FILE/WCHAR.MBR","wb")))
      perror("fopen error");
   for (i=0; i < sizeof(count) / sizeof(count[0]); i++)
   {
      for (j = 0; j < count[i]; j++)
         fwprintf(stream, L"*");      /* Print asterisk              */
         fwprintf(stream, L"¥n");     /* Move to the next line       */
   }
   fclose (stream);
}
/* The member WCHAR of file WCHAR will contain:
   *
   *****
   ********
   ***
   ***
   *****
   ******
   ********
   **********
 */
```

fwprintf() を使用する Unicode 例

```
#include <stdio.h>
#include <stdlib.h>
#include <locale.h>
/* This program is compile LOCALETYPE(*LOCALEUCS2) and             */
/* SYSIFCOPT(*IFSIO)                                               */
int main(void)
{
    FILE *stream;
    wchar_t wc = 0x0058;     /* UNICODE X */
    char c1 = 'c';
    char *s1 = "123";
    wchar_t ws[4];
    setlocale(LC_ALL,
     "/QSYS.LIB/EN_US.LOCALE"); /* a CCSID 37 locale */
    ws[0] = 0x0041;        /* UNICODE A   */
    ws[1] = (wchar_t)0x0042;        /* UNICODE B   */
    ws[2] = (wchar_t)0x0043;        /* UNICODE C   */
    ws[3] = (wchar_t)0x0000;

    stream = fopen("myfile.dat", "wb+");

    /* lc and ls take wide char as input and just copies then  */
    /* to the file.  So the file would look like this        */
    /* after the below fwprintf statement:                   */
    /* 0058002000200020004100420043                          */
    /* 0020 is UNICODE blank                                 */

    fwprintf(stream, L"%lc   %ls",wc,ws);
    /* c and s take multibyte as input and produce UNICODE   */
    /* In this case c1 and s1 are CCSID 37 characters based  */
    /* on the setlocale above.  So the characters are        */
    /* converted from CCSID 37 to UNICODE and will look      */
    /* like this in hex after the following fwprintf         */
    /* statement:   0063002000200020003100320033             */
    /* 0063 is a UNICODE c  0031 is a UNICODE 1 and so on    */

    fwprintf(stream, L"%c   %s",c1,s1);

    /* Now lets try width and precision.  6ls means write  */
    /* 6 wide characters so we will pad with 3 UNICODE     */
    /* blanks and %.2s means write no more then 2 wide     */
    /* characters.  So we get an output that looks like    */
    /* this:  00200020002000410042004300310032             */

    fwprintf(stream, L"%6ls%.2s",ws,s1);
}
```

関連情報

- 122 ページの『fprintf() ― フォーマット済みデータのストリームへの書き込み』
- 238 ページの『printf() ― 定様式の文字の出力』
- 444 ページの『vfprintf() ― ストリームへの引数データの出力』
- 452 ページの『vprintf() ― 引数データの出力』
- 56 ページの『btowc() ― 1 バイト文字のワイド文字への変換』
- 210 ページの『mbrtowc() ― マルチバイト文字からワイド文字への変換 (再始動可能)』
- 447 ページの『vfwprintf() ― 引数データのワイド文字としてのフォーマット設定とストリームへの書き込み』
- 459 ページの『vswprintf() ― ワイド文字のフォーマット設定とバッファーへの書き込み』
- 527 ページの『wprintf() ― データのワイド文字としてのフォーマット設定と出力』
- 15 ページの『<stdarg.h>』

- 20 ページの『<wchar.h>』

fwrite() ― 項目の書き込み

フォーマット

```
#include <stdio.h>
size_t fwrite(const void *buffer, size_t size, size_t count,
              FILE *stream);
```

言語レベル: ANSI

スレッド・セーフ: はい。

説明

`fwrite()` 関数は最大数 *count* の項目を、それぞれ *size* バイト長まで、*buffer* から出力 *stream* に書き込みます。

戻り値

`fwrite()` 関数は、正常に書き込まれた完全な項目の数を戻します。これは、エラーが発生した場合は *count* より少なくなることがあります。

レコード出力に `fwrite()` を使用するときには、*size* を 1 に設定し、*count* を書き込まれるバイト数を取得するためのレコード長に設定します。レコード入出力を使用する場合は、一度に 1 レコードしか書き込むことができません。

errno の値は、次のいずれかに設定されます。

値　**意味**

ECONVERT
変換エラーが発生しました。

ENOTWRITE
ファイルは書き込み操作用にオープンされません。

EPAD 書き込み操作で埋め込みが発生しました。

EPUTANDGET
読み取り操作の後、正しくない書き込み操作が発生しました。

ESTDERR
`stderr` をオープンできません。

ESTDIN
`stdin` をオープンできません。

ESTDOUT
`stdout` をオープンできません。

ETRUNC
入出力操作で切り捨てが発生しました。

EIOERROR
リカバリー不能な入出力エラーが発生しました。

EIORECERR
リカバリー可能な入出力エラーが発生しました。

fwrite() の使用例

次の例は NUM long 整数をストリームにバイナリー・フォーマットで書き込みます。

```
#include  <stdio.h>
#define  NUM  100

int main(void)
{
   FILE *stream;
   long list[NUM];
   int numwritten;
   int i;

   stream = fopen("MYLIB/MYFILE", "w+b");

   /* assign values to list[] */
   for (i=0;  i<=NUM; i++)
       list[i]=i;

   numwritten = fwrite(list, sizeof(long), NUM, stream);
   printf("Number of items successfully written = %d¥n", numwritten);
}
```

関連情報

- 114 ページの『fopen() ― ファイルのオープン』
- 132 ページの『fread() ― 項目の読み取り』
- 17 ページの『<stdio.h>』

fwscanf() ― ワイド文字を使用したストリームからのデータの読み取り

フォーマット

```
#include <stdio.h>
#include <wchar.h>
int fwscanf(FILE *stream, const wchar_t *format, argument-list);
```

言語レベル: ANSI

スレッド・セーフ: はい。

ロケール依存: この関数の振る舞いは、現行ロケールの LC_CTYPE カテゴリーおよび LC_NUMERIC カテゴリーの影響を受ける可能性があります。また、LOCALETYPE(*LOCALEUCS2) または LOCALETYPE(*LOCALEUTF) がコンパイル・コマンドに対して指定されている場合は、現行ロケールの LC_UNI_CTYPE カテゴリーおよび LC_UNI_NUMERIC カテゴリーにも影響を受ける可能性があります。この関数は、コンパイル・コマンドに対して LOCALETYPE(*CLD) が指定されている場合には使用できません。詳細については、 549 ページの『CCSID およびロケールの理解』を参照してください。

統合ファイル・システム・インターフェース: この関数は、コンパイル・コマンドに対して SYSIFCOPT(*NOIFSIO) が指定されている場合には使用できません。

ワイド文字関数: 詳細については、 553 ページの『ワイド文字』 を参照してください。

説明

fwscanf() 関数は、*format* により指されたワイド・ストリングの制御下で、*stream* により指されたストリームから入力データを読み取ります。書式ストリングは許容入力シーケンスおよびこれらが割り当てられるための変換方法を指定します。変換済み入力データを受信するために、fwscanf() 関数は後続の引数をオブジェクトを指すポインターとして使用します。

argument-list の各引数は、書式の型指定子に対応する型を持つ変数を指さなければなりません。

書式に対して引数が不十分な場合、振る舞いは予期できません。引数が残っているのに、書式がすべて使用されている場合、fwscanf() 関数は余分の引数を評価しますが、それ以外の場合は無視します。

書式は、ゼロ以上のディレクティブ、すなわち、1 つ以上の空白文字および普通のワイド文字 (% と空白文字は含まない) または変換指定で構成されます。各変換指定の先頭には、% が入ります。

この書式には scanf() 関数の書式ストリングと同じ形式と機能がありますが、以下の点が異なります。
- %c (l 接頭部なし) は、wcrtomb() の呼び出しにより変換したかのように、1 つ以上の wchar_t 文字 (文字数は精度によって異なります) をマルチバイト文字に変換します。
- %lc および %C は、1 つ以上の wchar_t 文字 (文字数は精度によって異なります) を wchar_t の配列に変換します。
- %s (l 接頭部なし) は、wcrtomb() 関数の呼び出しにより変換したかのように、空白文字ではない wchar_t 文字のシーケンスをマルチバイト文字に変換します。配列には NULL 終了文字が含まれます。
- %ls および %S は wchar_t の配列 (NULL 終了ワイド文字を含む) を wchar_t の配列にコピーします。

データが stdin のもので、stdin が上書きされていなかった場合、そのデータはジョブの CCSID にあると仮定されます。データは書式仕様の要求により変換されます。読み取られているファイルがファイル・モード rb でオープンされない場合には、無効な変換エラーが発生することがあります。

変換指定が無効な場合、その振る舞いは予期できません。入力中に fwscanf() 関数がファイル終了になった場合、変換は終了します。 fwscanf() 関数が現行ディレクティブ (先頭の空白文字が許可されている場合は、先頭の空白文字以外) に一致するすべての文字を読み取る前にファイル終了が発生すると、入力障害により現行ディレクティブの実行が終了します。それ以外の場合は、現行ディレクティブの実行がマッチング失敗により終了しない限り、それに続くディレクティブ (%n がある場合は、これ以外) の実行は入力失敗により終了します。

ディレクティブによりマッチングされない限り、fwscanf() 関数は末尾の空白文字 (改行ワイド文字を含む) を未読のままにします。%n ディレクティブによるもの以外の、リテラル・マッチと抑止された代入の正常終了を判別することはできません。

戻り値

fwscanf() 関数は、割り当てられた入力項目数を戻しますが、これは、以前のマッチングが失敗した場合、提供された数より少なくなることがあります。

変換の前に入力障害が起こった場合、fwscanf() 関数は EOF を戻します。

fwscanf() の使用例

次の例は、入力用にファイル **myfile.dat** をオープンし、その後このファイルをストリング、long 型整数値、文字、および浮動小数点値について走査します。

```
#include <stdio.h>
#include <wchar.h>

#define  MAX_LEN       80

int main(void)
{
   FILE *stream;
   long l;
   float fp;
   char s[MAX_LEN+1];
   char c;

   stream = fopen("myfile.dat", "r");

    /* Read data from file. */

   fwscanf(stream, L"%s", &s[0]);
   fwscanf(stream, L"%ld", &l);
   fwscanf(stream, L"%c", &c);
   fwscanf(stream, L"%f", &fp);

   printf("string = %s¥n", s);
   printf("long integer = %ld¥n", l);
   printf("char = %c¥n", c);
   printf("float = %f¥n", fp);
   return 0;

   /***********************************************
      If myfile.dat contains:
      abcdefghijklmnopqrstuvwxyz 343.2.

      The output should be:

      string = abcdefghijklmnopqrstuvwxyz
      long integer = 343
      char = .
      float = 2.000000
   ***********************************************/
}
```

fwscanf() を使用する Unicode 例

次の例は **unicode.dat** から Unicode ストリングを読み取り、それを画面に出力します。この例は、LOCALETYPE(*LOCALEUCS2) SYSIFCOPT(*IFSIO) でコンパイルされています。

```
#include   <stdio.h>
#include   <wchar.h>
#include   <locale.h>
void main(void)
{
FILE *stream;
wchar_t buffer[20];
stream=fopen("unicode.dat","rb");

fwscanf(stream,L"%ls", buffer);
wprintf(L"The string read was :%ls\n",buffer);

fclose(stream);
}

/* If the input in unicode.dat is :
 ABC
 and ABC is in unicode which in hex would be 0x0041, 0x0042, 0x0043
 then the output will be similar to:
 The string read was :ABC
*/
```

関連情報

- 138 ページの『fscanf() — フォーマット済みデータの読み取り』
- 149 ページの『fwprintf() — ワイド文字としてのデータのフォーマット設定とストリームへの書き込み』
- 344 ページの『scanf() — データの読み取り』
- 423 ページの『swprintf() — ワイド文字のフォーマット設定とバッファーへの書き込み』
- 425 ページの『swscanf() — ワイド文字データの読み取り』
- 528 ページの『wscanf() — ワイド文字書式ストリングを使用したデータの読み取り』
- 17 ページの『<stdio.h>』
- 20 ページの『<wchar.h>』

gamma() — ガンマ関数

フォーマット

```
#include <math.h>
double gamma(double x);
```

言語レベル: ILE C Extension

スレッド・セーフ: はい。

説明

`gamma()` 関数は、以下のように G(x) (ln(|G(x)|)) の絶対値の自然対数を計算します。

$$G(x) = \int_{0}^{\infty} e^{-t} \times t^{x-1} dt$$

引数 *x* は正の実数値でなければなりません。

戻り値

gamma() 関数は、ln(|G(x)|) の値を戻します。 *x* が負の値の場合、errno は EDOM に設定されます。結果がオーバーフローになると、gamma() は HUGE_VAL を戻し、 errno に ERANGE を設定します。

gamma() の使用例

この例では、gamma() を使用して ln(|G(x)|) を計算します。ここでは x = 42 とします。

```
#include <math.h>
#include <stdio.h>

int main(void)
{
   double x=42, g_at_x;

   g_at_x = exp(gamma(x));       /* g_at_x = 3.345253e+49 */
   printf ("The value of G(%4.2lf) is %7.2e¥n", x, g_at_x);
}

/************************  Output should be similar to: **********

The value of G(42.00) is 3.35e+49
*/
```

関連情報

- 53 ページの『ベッセル関数』
- 91 ページの『erf() - erfc() — 誤差関数の計算』
- 9 ページの『<math.h>』

_gcvt - 浮動小数点からストリングへの変換

フォーマット

```
#include <stdlib.h>
char *_gcvt(double value, int ndec, char *buffer);
```

注: _gcvt 関数は C++ のみがサポート対象となっており、C はサポート対象外です。

言語レベル: Extension

スレッド・セーフ: はい。

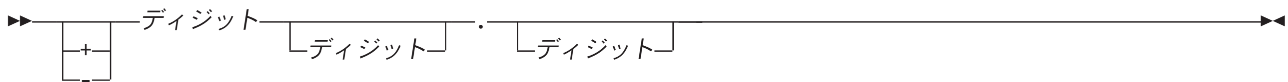
ロケール依存: この関数の振る舞いは、現行ロケールの LC_CTYPE カテゴリーおよび LC_NUMERIC カテゴリーの影響を受ける可能性があります。詳細については、 549 ページの『CCSID およびロケールの理解』を参照してください。

説明

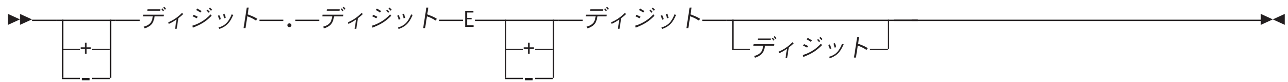
_gcvt() は、浮動小数点値をバッファーがポイントした文字列に変換します。バッファーは、変換された値、および _gcvt() によってストリングの終わりに自動的に追加されるヌル文字 (¥0) を保持するのに十分な大きさでなければなりません。オーバーフローに対する対応策はありません。

_gcvt() は、FORTRAN F フォーマットで *ndec* 有効数字を生成しようと試みます。それに失敗した場合、FORTRAN E フォーマットで *ndec* 有効数字を生成します。後続ゼロは、重要ではない場合、変換で抑制される可能性があります。

FORTRAN F 番号の形式は以下のとおりです。



FORTRAN E 番号の形式は以下のとおりです。



_gcvt は、無限大値もストリング INFINITY に変換します。

戻り値

_gcvt() は、ディジットのストリングを指すポインターを戻します。変換を実行するためにメモリーを割り振ることができない場合、_gcvt() は空ストリングを戻し、errno に ENOMEM を設定します。

_gcvt() の使用例

この例では、値 -3.1415e3 を文字ストリングに変換し、変換された文字ストリングを文字配列 buffer1 に置きます。

```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
    char buffer1[10];
    _gcvt(-3.1415e3, 7, buffer1);
    printf("The first result is %s \n", buffer1);
    return 0;
}
```

The output should be:

```
The first result is -3141.5
```

関連情報

- 18 ページの『<stdlib.h>』

getc() - getchar() — 文字の読み取り

フォーマット

```
#include <stdio.h>
int getc(FILE *stream);
int getchar(void);
```

言語レベル: ANSI

スレッド・セーフ: いいえ。#undef getc または #undef getchar は、getc または getchar 関数のマクロ・バージョンの代わりに、これらの関数の呼び出しを許可します。関数はスレッド・セーフです。

説明

getc() 関数は、現在の *stream* 位置から単一文字を読み取り、*stream* 位置を次の文字に進めます。getchar() 関数は getc(stdin) と同じです。

`getc()` 関数と `fgetc()` 関数の違いは、`getc()` の場合、引数の評価が複数回にわたって行われるように実装可能なことです。したがって、`getc()` に対する *stream* 引数は、副次作用のある式にはしないでください。

戻り値

`getc()` および `getchar()` 関数は、読み取られた文字を戻します。 EOF に戻り値がある場合は、エラーまたは EOF 条件を表します。 `ferror()` または `feof()` を使用すると、エラーまたは EOF 条件のいずれが発生したかを判別することができます。

errno の値は、次のいずれかに設定されます。

値 **意味**

EBADF
ファイル・ポインター、または記述子が有効ではありません。

ECONVERT
変換エラーが発生しました。

EGETANDPUT
書き込み操作後に、無許可の読み取り操作が発生しました。

EIOERROR
リカバリー不能な入出力エラーが発生しました。

EIORECERR
リカバリー可能な入出力エラーが発生しました。

`getc()` および `getchar()` 関数はレコード・モードではサポートされません。

`getc()` の使用例

この例では、入力データ行を `stdin` ストリームから取得しています。 `getc(stdin)` を `getchar()` の代わりに for ステートメントで使用して、`stdin` から入力データ行を取得できます。

```
#include  <stdio.h>

#define LINE 80

int main(void)
{
  char buffer[LINE+1];
  int i;
  int ch;

  printf( "Please enter string¥n" );

  /* Keep reading until either:
     1. the length of LINE is exceeded  or
     2. the input character is EOF  or
     3. the input character is a new-line character
   */

  for ( i = 0; ( i  < LINE ) && (( ch = getchar()) != EOF) &&
               ( ch !='¥n' ); ++i )
    buffer[i] = ch;

  buffer[i] = '¥0';  /* a string should always end with '¥0' ! */

  printf( "The string is %s¥n", buffer );
}
```

関連情報

- 102 ページの『fgetc() ― 文字の読み取り』
- 107 ページの『fgetwc() ― ストリームのワイド文字の読み取り』
- 162 ページの『gets() ― 行の読み取り』
- 163 ページの『getwc() ― ストリームからのワイド文字の読み取り』
- 165 ページの『getwchar() ― STDIN からのワイド文字の取得』
- 249 ページの『putc() - putchar() ― 文字の書き込み』
- 439 ページの『ungetc() ― 入力ストリームへの文字のプッシュ』
- 17 ページの『<stdio.h>』

getenv() ― 環境変数の検索

フォーマット

```
#include <stdlib.h>
char *getenv(const char *varname);
```

言語レベル: ANSI

スレッド・セーフ: はい。

ジョブ CCSID インターフェース: この関数に送信される文字データは、すべてジョブの CCSID 内にあると想定されます。この関数によって戻された文字データは、すべてジョブの CCSID 内にあります。詳細については、 549 ページの『CCSID およびロケールの理解』を参照してください。

説明

`getenv()` 関数は、*varname* に対応するエントリー用の環境変数のリストを検索します。

戻り値

getenv() 関数は、現行環境の *varname* で指定されている値を含むストリングへのポインターを戻します。getenv() が環境ストリングを見つけられない場合は NULL が戻され、エラーを示す errno が設定されます。

getenv() の使用例

```
#include  <stdlib.h>
#include  <stdio.h>

/* Where the environment variable 'PATH' is set to a value. */

int main(void)
{
   char *pathvar;

   pathvar = getenv("PATH");
   printf("pathvar=%s",pathvar);
}
```

関連情報

- 18 ページの『<stdlib.h>』
- 250 ページの『putenv() ― 環境変数の変更/追加』
- i5/OS Information Center の『API』トピックの環境変数 API。

_GetExcData() ― 例外データの取得

フォーマット

```
#include <signal.h>
void _GetExcData(_INTRPT_Hndlr_Parms_T *parms);
```

言語レベル: ILE C Extension

スレッド・セーフ: はい。

ジョブ CCSID インターフェース: この関数に送信される文字データは、すべてジョブの CCSID 内にあると想定されます。この関数によって戻された文字データは、すべてジョブの CCSID 内にあります。詳細については、 549 ページの『CCSID およびロケールの理解』を参照してください。

説明

_GetExcData() 関数は、現行の例外情報を C シグナル・ハンドラー内から戻します。 _GetExcData() 関数の呼び出し元は、十分なストレージをタイプ _INTRPT_Hndlr_Parms_T の構造に割り当てなくてはなりません。_GetExcData() 関数が外部シグナル・ハンドラーから呼び出された場合、*parms* によってポイントされたストレージは更新されません。

この関数は、SYSIFCOPT(*ASYNCSIGNAL) をコンパイル・コマンドで指定した場合に使用できません。SYSIFCOPT(*ASYNCSIGNAL) が指定された場合、ILE C signal() 関数との間で確立されたシグナル・ハンドラーは、シグナル・ハンドラー呼び出しの原因となった可能性のある例外情報にアクセスする方法を持てなくなります。ただし、sigaction() 関数との間で確立された拡張シグナル・ハンドラーは、この例外情報に対するアクセス権限を持っています。拡張シグナル・ハンドラーには、以下の関数プロトタイプがあります。

```
void func( int signo, siginfo_t *info, void *context )
```

例外情報は siginfo_t 構造体に追加され、次にこの構造体は拡張シグナル・ハンドラーへの 2 次パラメーターとして渡されます。

siginfo_t 構造体は signal.h 内で定義されます。siginfo_t 構造体の si_sigdata フィールドの後には、例外関連データが続きます。 sigdata_t 構造体の se_data フィールドからアドレス指定することができます。

siginfo_t 構造体に付加される例外データのフォーマットは、except.h 内の _INTRPT_Hndlr_Parms_T 構造体によって定義されます。

戻り値

戻り値はありません。

_GetExcData() の使用例

この例では、MI ライブラリー関数からの例外が、シグナル・ハンドリング関数を使用してモニターおよび処理される方法を示します。シグナル・ハンドラー my_signal_handler は、rslvsp() 関数が 0x2201 例外をシグナル通知する前に登録されます。 SIGSEGV シグナルが発呼されると、シグナル・ハンドラーは呼び出されます。 0x2201 例外が発生すると、シグナル・ハンドラーは QUSRCRTS API を呼び出してスペースを作成します。

```
#include  <signal.h>
#include  <QSYSINC/MIH/RSLVSP>
#include  <QSYSINC/H/QUSCRTUS>
#include  <string.h>

#define CREATION_SIZE  65500

void my_signal_handler(int sig) {

   _INTRPT_Hndlr_Parms_T excp_data;
   int                   error_code = 0;

   /* Check the message id for exception 0x2201 */
   _GetExcData(&excp_data);

   if (!memcmp(excp_data.Msg_Id, "MCH3401", 7))
      QUSCRTUS("MYSPACE   QTEMP     ",
               "MYSPACE   ",
               CREATION_SIZE,
               "¥0",
               "*ALL      ",
               "MYSPACE example for Programmer's Reference        ",
               "*YES      ",
               &error_code);
}
```

関連情報

- 362 ページの『signal() ― 割り込みシグナルの処理』
- 4 ページの『<except.h>』

gets() ― 行の読み取り

フォーマット

```
#include <stdio.h>
char *gets(char *buffer);
```

言語レベル: ANSI

スレッド・セーフ: はい。

説明

`gets()` 関数は、標準入力ストリーム `stdin` から行を読み取り、バッファーに保管します。行は、最初の改行文字 (¥n) または EOF まで (その文字を含まない) のすべての文字で構成されています。読み込みが成功した場合、次に `gets()` 関数は読み取られた行を戻す前に、改行文字をヌル文字 (¥0) に置き換えます。

戻り値

正常に実行された場合、`gets()` 関数は引数を戻します。 NULL ポインターの戻り値がある場合、エラー、または文字が読み取られていない EOF 条件を表します。 `ferror()` 関数または `feof()` 関数を使用すると、発生した条件を判別することができます。エラーが発生すると、バッファーに保管される値は未定義なものになります。 EOF 条件が発生すると、バッファーは変更されません。

`gets()` の使用例

この例では、入力データの行を `stdin` から取得しています。

```
#include  <stdio.h>

#define MAX_LINE 100

int main(void)
{
   char line[MAX_LINE];
   char *result;

   printf("Please enter a string:¥n");
   if ((result = gets(line)) != NULL)
      printf("The string is: %s¥n", line);
   else if (ferror(stdin))
      perror("Error");
}
```

関連情報

- 105 ページの『fgets() — ストリングの読み取り』
- 109 ページの『fgetws() — ストリームのワイド文字のストリングの読み取り』
- 99 ページの『feof() — ファイル終了標識のテスト』
- 100 ページの『ferror() — 読み取り/書き込みエラーのテスト』
- 127 ページの『fputs() — ストリングの書き込み』
- 158 ページの『getc() - getchar() — 文字の読み取り』
- 251 ページの『puts() — ストリングの書き込み』
- 17 ページの『<stdio.h>』

getwc() — ストリームからのワイド文字の読み取り

フォーマット

```
#include <stdio.h>
#include <wchar.h>
wint_t getwc(FILE *stream);
```

言語レベル: ANSI

スレッド・セーフ: はい。

ロケール依存: この関数の振る舞いは、現行ロケールの LC_CTYPE カテゴリーの影響を受ける可能性があります。また、この振る舞いは、LOCALETYPE(*LOCALEUCS2) または LOCALETYPE(*LOCALEUTF) がコンパイル・コマンドに対して指定されている場合は、現行ロケールの LC_UNI_CTYPE カテゴリーにも影響を受ける可能性があります。この関数は、コンパイル・コマンドに対して LOCALETYPE(*CLD) が指定されている場合には使用できません。詳細については、 549 ページの『CCSID およびロケールの理解』を参照してください。

統合ファイル・システム・インターフェース: この関数は、コンパイル・コマンドに対して SYSIFCOPT(*NOIFSIO) が指定されている場合には使用できません。

ワイド文字関数: 詳細については、 553 ページの『ワイド文字』 を参照してください。

説明

`getwc()` 関数は、次のマルチバイト文字を stream から読み取ってからワイド文字に変換し、ストリームに関連するファイル位置標識を進めます。

`getwc()` 関数は、マクロとして設定された場合、*stream* を複数回評価できます。これを除けば `fgetwc()` 関数と同じです。 したがって、引数は副次作用をとる式であってはなりません。

同じストリームに対する後続の読み取り操作間で現行ロケールが変更された場合は、未定義の結果が発生することがあります。非ワイド文字関数を `getwc()` 関数と共に同じストリーム上で使用すると、予期しない振る舞いが生じる結果となります。

`getwc()` 関数の呼び出し後、EOF に到達していない限り、ストリームに対する書き込み関数を呼び出す前にバッファーをフラッシュするか、またはストリーム・ポインターの位置を変更します。ストリームに対する書き込み操作の後、`getwc()` 関数を呼び出す前にバッファーをフラッシュするか、ストリーム・ポインターの位置を変更してください。

戻り値

`getwc()` 関数は、入力ストリームから次のワイド文字を戻すか、または WEOF を戻します。エラーが発生した場合、`getwc()` 関数はエラー標識を設定します。 `getwc()` 関数は EOF に遭遇すると、EOF 指標を設定します。マルチバイト文字のワイド文字への変換中にエンコード・エラーが発生すると、`getwc()` 関数は errno に EILSEQ を設定します。

`ferror()` または `feof()` 関数を使用すると、エラーまたは EOF 条件のいずれが発生したかを判別することができます。データの最後のバイトを超えて読み取ろうとしたときのみ、EOF に達します。データの最後のバイトまで (最後のバイトを含む) 読み取っても、EOF 標識はオンになりません。

`getwc()` の errno 値について詳しくは、107 ページの『fgetwc() ー ストリームのワイド文字の読み取り』を参照してください。

`getwc()` の使用例

```
#include <stdio.h>
#include <stdlib.h>
#include <wchar.h>
#include <errno.h>

int main(void)
{
   FILE   *stream;
   wint_t wc;

   if (NULL == (stream = fopen("getwc.dat", "r"))) {
      printf("Unable to open: ¥"getwc.dat¥"¥n");
      exit(1);
   }

   errno = 0;
   while (WEOF != (wc = getwc(stream)))
      printf("wc = %lc¥n", wc);

   if (EILSEQ == errno) {
      printf("An invalid wide character was encountered.¥n");
      exit(1);
   }
   fclose(stream);
   return 0;

   /********************************************************
      Assuming the file getwc.dat contains:

      Hello world!

      The output should be similar to:

      wc = H
      wc = e
      wc = l
      wc = l
      wc = o
      :
   ********************************************************/
}
```

関連情報

- 107 ページの『fgetwc() ― ストリームのワイド文字の読み取り』
- 『getwchar() ― STDIN からのワイド文字の取得』
- 158 ページの『getc() - getchar() ― 文字の読み取り』
- 252 ページの『putwc() ― ワイド文字の書き込み』
- 440 ページの『ungetwc() ― 入力ストリームへのワイド文字のプッシュ』
- 17 ページの『<stdio.h>』
- 20 ページの『<wchar.h>』

getwchar() ― STDIN からのワイド文字の取得

フォーマット

```
#include <wchar.h>
wint_t getwchar(void);
```

言語レベル: ANSI

スレッド・セーフ: はい。

ロケール依存: この関数の振る舞いは、現行ロケールの LC_CTYPE カテゴリーの影響を受ける可能性があります。また、この振る舞いは、LOCALETYPE(*LOCALEUCS2) または LOCALETYPE(*LOCALEUTF) がコンパイル・コマンドに対して指定されている場合は、現行ロケールの LC_UNI_CTYPE カテゴリーにも影響を受ける可能性があります。この関数は、コンパイル・コマンドに対して LOCALETYPE(*CLD) が指定されている場合には使用できません。詳細については、 549 ページの『CCSID およびロケールの理解』を参照してください。

統合ファイル・システム・インターフェース: この関数は、コンパイル・コマンドに対して SYSIFCOPT(*NOIFSIO) が指定されている場合には使用できません。

ワイド文字関数: 詳細については、 553 ページの『ワイド文字』 を参照してください。

説明

`getwchar()` 関数は、次のマルチバイト文字を STDIN から読み取ってからワイド文字に変換し、`stdin` に関連するファイル位置標識を進めます。 `getwchar()` 関数への呼び出しは、`getwc(stdin)` への呼び出しと同等です。

同じストリームに対する後続の読み取り操作間で現行ロケールが変更された場合は、未定義の結果が発生することがあります。非ワイド文字関数を `getwchar()` 関数と共に `stdin` で使用すると、予期しない振る舞いが生じる結果となります。

戻り値

`getwchar()` 関数は、`stdin` から次のワイド文字を戻すか、または WEOF を戻します。 `getwchar()` 関数は EOF に遭遇すると、ストリームに EOF 指標を設定し、WEOF を戻します。読み取りエラーが発生すると、ストリームのエラー標識が設定され、`getwchar()` 関数は WEOF を戻します。マルチバイト文字のワイド文字への変換中にエンコード・エラーが発生すると、`getwchar()` 関数は errno に EILSEQ を設定し、WEOF を戻します。

`ferror()` または `feof()` 関数を使用すると、エラーまたは EOF 条件のいずれが発生したかを判別することができます。データの最後のバイトを超えて読み取ろうとしたときのみ、EOF に達します。データの最後のバイトまで (最後のバイトを含む) 読み取っても、EOF 標識はオンになりません。

`getwchar()` の errno 値について詳しくは、107 ページの『fgetwc() ― ストリームのワイド文字の読み取り』を参照してください。

`getwchar()` の使用例

この例では、`getwchar()` を使用してワイド文字をキーボードから読み取り、次に、そのワイド文字を出力します。

```
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <wchar.h>

int main(void)
{
   wint_t  wc;

   errno = 0;
   while (WEOF != (wc = getwchar()))
      printf("wc = %lc¥n", wc);

   if (EILSEQ == errno) {
      printf("An invalid wide character was encountered.¥n");
      exit(1);
   }
   return 0;

   /****************************************************************
      Assuming you enter: abcde

      The output should be:

      wc = a
      wc = b
      wc = c
      wc = d
      wc = e
   ****************************************************************/
}
```

関連情報

- 102 ページの『fgetc() ― 文字の読み取り』
- 107 ページの『fgetwc() ― ストリームのワイド文字の読み取り』
- 109 ページの『fgetws() ― ストリームのワイド文字のストリングの読み取り』
- 158 ページの『getc() - getchar() ― 文字の読み取り』
- 163 ページの『getwc() ― ストリームからのワイド文字の読み取り』
- 440 ページの『ungetwc() ― 入力ストリームへのワイド文字のプッシュ』
- 20 ページの『<wchar.h>』

gmtime() ― 時間の変換

フォーマット

```
#include <time.h>
struct tm *gmtime(const time_t *time);
```

言語レベル: ANSI

スレッド・セーフ: いいえ。代わりに gmtime_r() を使用します。

説明

gmtime() 関数は、*time* 値を秒単位で分割し、<time.h> で定義された *tm* 構造体に保管します。 *time* の値は、通常、time() 関数を呼び出して取得します。

tm 構造体のフィールドには、以下のものがあります。

tm_sec 秒 (0 から 61)

tm_min
: 分 (0 から 59)

tm_hour
: 時間 (0 から 23)

tm_mday
: 日 (1 から 31)

tm_mon
: 月 (0 から 11、1 月が 0 になる)

tm_year
: 年 (現在の年から 1900 を差し引く)

tm_wday
: 曜日 (0 から 6、日曜日が 0 になる)

tm_yday
: ユリウス日付 (0 から 365、1 月 1 日が 0 になる)

tm_isdst
: 夏時間 (DST) が無効な場合は 0。夏時間 (DST) が有効な場合は正の値。情報が使用可能でない場合は負の値。

戻り値

gmtime() 関数は、結果である tm 構造体へポインターを戻します。

注:

1. tm_sec の範囲 (0 から 61) は、2 うるう秒までを考慮に入れています。
2. gmtime() および localtime() 関数は、静的に割り振られた共通のバッファーを変換に使用します。これらの関数の 1 つを呼び出すごとに、以前の呼び出しの結果を変更します。
3. カレンダー時間とは、エポックである UTC (1970 年 1 月 1 日 00:00:00) から数えた秒数です。

gmtime() の使用例

この例では、gmtime() 関数を使用して time_t 表記を協定世界時 (UTC) 文字ストリングへ調整し、次にそれを asctime() 関数を使用して出力可能なストリングに変換します。

```
#include <stdio.h>
#include <time.h>

int main(void)
{
   time_t ltime;

      time(&ltime);
  printf ("Coordinated Universal Time is %s\n",
            asctime(gmtime(&ltime)));
}

/************************  Output should be similar to:  **********

Coordinated Universal Time is Wed Aug 18 21:01:44 1993
*/
```

関連情報

- 41 ページの『asctime() ― 時間から文字ストリングへの変換』
- 43 ページの『asctime_r() ― 時間から文字ストリングへの変換 (再始動可能)』
- 75 ページの『ctime() ― 時間から文字ストリングへの変換』
- 77 ページの『ctime64() ― 時間から文字ストリングへの変換』
- 80 ページの『ctime64_r() ― 時間から文字ストリングへの変換 (再始動可能)』
- 78 ページの『ctime_r() ― 時間から文字ストリングへの変換 (再始動可能)』
- 『gmtime64() ― 時間の変換』
- 173 ページの『gmtime64_r() ― 時間の変換 (再始動可能)』
- 171 ページの『gmtime_r() ― 時間の変換 (再始動可能)』
- 192 ページの『localtime() ― 時間の変換』
- 194 ページの『localtime64() ― 時間の変換』
- 197 ページの『localtime64_r() ― 時間の変換 (再始動可能)』
- 195 ページの『localtime_r() ― 時間の変換 (再始動可能)』
- 228 ページの『mktime() ― 地方時の変換』
- 230 ページの『mktime64() ― 地方時の変換』
- 354 ページの『setlocale() ― ロケールの設定』
- 429 ページの『time() ― 現在時刻の判別』
- 430 ページの『time64() ― 現在時刻の判別』
- 19 ページの『<time.h>』

gmtime64() ― 時間の変換

フォーマット

```
#include <time.h>
struct tm *gmtime64(const tim64_t *time);
```

言語レベル: ILE C Extension

スレッド・セーフ: いいえ。代わりに `gmtime64_r()` を使用します。

説明

`gmtime64()` 関数は、*time* 値を秒単位で分割し、<time.h> で定義された *tm* 構造体に保管します。 *time* 値は、通常、`time64()` 関数を呼び出して取得します。

tm 構造体のフィールドには、以下のものがあります。

tm_sec 秒 (0 から 61)

tm_min
分 (0 から 59)

tm_hour
時間 (0 から 23)

tm_mday
日 (1 から 31)

tm_mon
: 月 (0 から 11、1 月が 0 になる)

tm_year
: 年 (現在の年から 1900 を差し引く)

tm_wday
: 曜日 (0 から 6、日曜日が 0 になる)

tm_yday
: ユリウス日付 (0 から 365、1 月 1 日が 0 になる)

tm_isdst
: 夏時間 (DST) が無効な場合は 0。夏時間 (DST) が有効な場合は正の値。情報が使用可能でない場合は負の値。

戻り値

gmtime64() 関数は、結果である tm 構造体へポインターを戻します。

注:

1. tm_sec の範囲 (0 から 61) は、2 うるう秒までを考慮に入れています。
2. gmtime64() および localtime64() 関数は、静的に割り振られた共通のバッファーを変換に使用します。これらの関数の 1 つを呼び出すごとに、以前の呼び出しの結果を変更します。asctime_r()、ctime64_r()、gmtime64_r()、および localtime64_r() 関数は、戻りストリングの保持用に静的に割り振られた共通のバッファーを使用しません。これらの関数は、再入可能性が望ましい場合に、asctime()、ctime64()、gmtime64()、および localtime64() 関数の代わりに使用可能です。
3. カレンダー時間とは、エポックである UTC (1970 年 1 月 1 日 00:00:00) から数えた秒数です。

gmtime64() の使用例

この例では、gmtime64() 関数を使用して time64_t 表記を協定世界時 (UTC) 文字ストリングに調整し、次にそれを asctime() 関数を使用して出力可能なストリングに変換します。

```
#include <stdio.h>
#include <time.h>

int main(void)
{
   time64_t ltime;

      time64(&ltime);
  printf ("Universal Coordinate Time is %s",
            asctime(gmtime64(&ltime)));
}

/************************  Output should be similar to:  **********

Universal Coordinate Time is Wed Aug 18 21:01:44 1993
*/
```

関連情報

- 41 ページの『asctime() — 時間から文字ストリングへの変換』
- 43 ページの『asctime_r() — 時間から文字ストリングへの変換 (再始動可能)』
- 75 ページの『ctime() — 時間から文字ストリングへの変換』
- 77 ページの『ctime64() — 時間から文字ストリングへの変換』

- 80 ページの『ctime64_r() ― 時間から文字ストリングへの変換 (再始動可能)』
- 78 ページの『ctime_r() ― 時間から文字ストリングへの変換 (再始動可能)』
- 167 ページの『gmtime() ― 時間の変換』
- 『gmtime_r() ― 時間の変換 (再始動可能)』
- 173 ページの『gmtime64_r() ― 時間の変換 (再始動可能)』
- 192 ページの『localtime() ― 時間の変換』
- 194 ページの『localtime64() ― 時間の変換』
- 197 ページの『localtime64_r() ― 時間の変換 (再始動可能)』
- 195 ページの『localtime_r() ― 時間の変換 (再始動可能)』
- 228 ページの『mktime() ― 地方時の変換』
- 230 ページの『mktime64() ― 地方時の変換』
- 354 ページの『setlocale() ― ロケールの設定』
- 429 ページの『time() ― 現在時刻の判別』
- 430 ページの『time64() ― 現在時刻の判別』
- 19 ページの『<time.h>』

gmtime_r() ― 時間の変換 (再始動可能)

フォーマット

```
#include <time.h>
struct tm *gmtime_r(const time_t *time, struct tm *result);
```

言語レベル: XPG4

スレッド・セーフ: はい。

説明

この関数は、`gmtime()` の再始動可能バージョンです。

`gmtime_r()` 関数は、*time* 値を秒単位で分割し、*result* に保管します。*result* は、<time.h> で定義された *tm* 構造体を指すポインターです。 time 値は、通常、`time()` 関数を呼び出して取得します。

tm 構造体のフィールドには、以下のものがあります。

tm_sec 秒 (0 から 61)

tm_min
: 分 (0 から 59)

tm_hour
: 時間 (0 から 23)

tm_mday
: 日 (1 から 31)

tm_mon
: 月 (0 から 11、1 月が 0 になる)

tm_year
: 年 (現在の年から 1900 を差し引く)

tm_wday
: 曜日 (0 から 6、日曜日が 0 になる)

tm_yday
: ユリウス日付 (0 から 365、1 月 1 日が 0 になる)

tm_isdst
: 夏時間 (DST) が無効な場合は 0。夏時間 (DST) が有効な場合は正の値。情報が使用可能でない場合は負の値。

戻り値

gmtime_r() 関数は、結果である tm 構造体へポインターを戻します。

注:

1. tm_sec の範囲 (0 から 61) は、2 うるう秒までを考慮に入れています。
2. gmtime() および localtime() 関数は、静的に割り振られた共通のバッファーを変換に使用します。これらの関数の 1 つを呼び出すごとに、以前の呼び出しの結果を変更します。asctime_r()、ctime_r()、gmtime_r()、および localtime_r() 関数は、戻りストリングの保持用に静的に割り振られた共通のバッファーを使用しません。これらの関数は、再入可能性が望ましい場合に、asctime()、ctime()、gmtime()、および localtime() 関数の代わりに使用可能です。
3. カレンダー時間とは、エポックである UTC (1970 年 1 月 1 日 00:00:00) から数えた秒数です。

gmtime_r() の使用例

この例では、gmtime_r() 関数を使用して time_t 表記を協定世界時 (UTC) 文字ストリングへ調整し、次にそれを asctime_r() 関数を使用して出力可能なストリングに変換します。

```
#include <stdio.h>
#include <time.h>

int main(void)
{
   time_t ltime;
   struct tm mytime;
   char buf[50];

   time(&ltime)
 printf ("Coordinated Universal Time is %s¥n",
            asctime_r(gmtime_r(&ltime, &mytime), buf));
}

/************************  Output should be similar to:  **********

Coordinated Universal Time is Wed Aug 18 21:01:44 1993
*/
```

関連情報

- 41 ページの『asctime() — 時間から文字ストリングへの変換』
- 43 ページの『asctime_r() — 時間から文字ストリングへの変換 (再始動可能)』
- 75 ページの『ctime() — 時間から文字ストリングへの変換』
- 77 ページの『ctime64() — 時間から文字ストリングへの変換』
- 80 ページの『ctime64_r() — 時間から文字ストリングへの変換 (再始動可能)』
- 78 ページの『ctime_r() — 時間から文字ストリングへの変換 (再始動可能)』

- 167 ページの『gmtime() ― 時間の変換』
- 169 ページの『gmtime64() ― 時間の変換』
- 『gmtime64_r() ― 時間の変換 (再始動可能)』
- 192 ページの『localtime() ― 時間の変換』
- 194 ページの『localtime64() ― 時間の変換』
- 197 ページの『localtime64_r() ― 時間の変換 (再始動可能)』
- 195 ページの『localtime_r() ― 時間の変換 (再始動可能)』
- 228 ページの『mktime() ― 地方時の変換』
- 230 ページの『mktime64() ― 地方時の変換』
- 429 ページの『time() ― 現在時刻の判別』
- 430 ページの『time64() ― 現在時刻の判別』
- 19 ページの『<time.h>』

gmtime64_r() ― 時間の変換 (再始動可能)

フォーマット

```
#include <time.h>
struct tm *gmtime64_r(const time64_t *time, struct tm *result);
```

言語レベル: ILE C Extension

スレッド・セーフ: はい。

説明

この関数は、`gmtime64()` の再始動可能バージョンです。

`gmtime64_r()` 関数は、*time* 値を秒単位で分割し、*result* に保管します。*result* は、<time.h> で定義された *tm* 構造体を指すポインターです。 *time* 値は、通常、`time64()` 関数を呼び出して取得します。

tm 構造体のフィールドには、以下のものがあります。

tm_sec 秒 (0 から 61)

tm_min
: 分 (0 から 59)

tm_hour
: 時間 (0 から 23)

tm_mday
: 日 (1 から 31)

tm_mon
: 月 (0 から 11、1 月が 0 になる)

tm_year
: 年 (現在の年から 1900 を差し引く)

tm_wday
: 曜日 (0 から 6、日曜日が 0 になる)

tm_yday
: ユリウス日付 (0 から 365、1 月 1 日が 0 になる)

tm_isdst
: 夏時間 (DST) が無効な場合は 0。夏時間 (DST) が有効な場合は正の値。情報が使用可能でない場合は負の値。

戻り値

gmtime64_r() 関数は、結果である tm 構造体へポインターを戻します。

注:

1. tm_sec の範囲 (0 から 61) は、2 うるう秒までを考慮に入れています。
2. gmtime64() および localtime64() 関数は、静的に割り振られた共通のバッファーを変換に使用します。これらの関数の 1 つを呼び出すごとに、以前の呼び出しの結果を変更します。asctime_r()、ctime64_r()、gmtime64_r()、および localtime64_r() 関数は、戻りストリングの保持用に静的に割り振られた共通のバッファーを使用しません。これらの関数は、再入可能性が望ましい場合に、asctime()、ctime64()、gmtime64()、および localtime64() 関数の代わりに使用可能です。
3. カレンダー時間とは、エポックである UTC (1970 年 1 月 1 日 00:00:00) から数えた秒数です。

gmtime64_r() の使用例

この例では、gmtime64_r() 関数を使用して time64_t 表記を協定世界時 (UTC) 文字ストリングへ調整し、次にそれを asctime_r() 関数を使用して出力可能なストリングに変換します。

```
#include <stdio.h>
#include <time.h>

int main(void)
{
   time64_t ltime;
   struct tm mytime;
   char buf[50];

   time64(&ltime)
 printf ("Universal Coordinate Time is %s",
           asctime_r(gmtime64_r(&ltime, &mytime), buf));
}

/************************  Output should be similar to:  **********

Universal Coordinate Time is Wed Aug 18 21:01:44 1993
*/
```

関連情報

- 41 ページの『asctime() — 時間から文字ストリングへの変換』
- 43 ページの『asctime_r() — 時間から文字ストリングへの変換 (再始動可能)』
- 75 ページの『ctime() — 時間から文字ストリングへの変換』
- 77 ページの『ctime64() — 時間から文字ストリングへの変換』
- 80 ページの『ctime64_r() — 時間から文字ストリングへの変換 (再始動可能)』
- 78 ページの『ctime_r() — 時間から文字ストリングへの変換 (再始動可能)』
- 167 ページの『gmtime() — 時間の変換』
- 169 ページの『gmtime64() — 時間の変換』

- 171 ページの『gmtime_r() — 時間の変換 (再始動可能)』
- 192 ページの『localtime() — 時間の変換』
- 194 ページの『localtime64() — 時間の変換』
- 197 ページの『localtime64_r() — 時間の変換 (再始動可能)』
- 195 ページの『localtime_r() — 時間の変換 (再始動可能)』
- 228 ページの『mktime() — 地方時の変換』
- 230 ページの『mktime64() — 地方時の変換』
- 429 ページの『time() — 現在時刻の判別』
- 430 ページの『time64() — 現在時刻の判別』
- 19 ページの『<time.h>』

hypot() — 斜辺の計算

フォーマット

```
#include <math.h>
double hypot(double side1, double side2);
```

言語レベル: ILE C Extension

スレッド・セーフ: はい。

説明

`hypot()` 関数は、2 つの辺 *side1* および *side2* の長さに基づいて、直角三角形の斜辺の長さを計算します。 `hypot()` 関数の呼び出しは次の式と同じです。

```
sqrt(side1 * side1 + side2 * side2);
```

戻り値

`hypot()` 関数は、斜辺の長さを戻します。結果がオーバーフローの場合、`hypot()` は errno に ERANGE を設定し、値 HUGE_VAL を戻します。結果がアンダーフローの場合、`hypot()` は errno に ERANGE を設定し、ゼロを戻します。 errno の値も EDOM に設定される可能性があります。

hypot() の使用例

この例では、サイドが 3.0 および 4.0 の直角三角形の斜辺を計算します。

```
#include <math.h>

int main(void)
{
   double x, y, z;

   x = 3.0;
   y = 4.0;
   z = hypot(x,y);

   printf("The hypotenuse of the triangle with sides %lf and %lf"
          " is %lf¥n", x, y, z);
}

/********************  Output should be similar to:  **************

The hypotenuse of the triangle with sides 3.000000 and 4.000000 is 5.000000
*/
```

関連情報

- 369 ページの『sqrt() ― 平方根の計算』
- 9 ページの『<math.h>』

isalnum() - isxdigit() ― 整数値のテスト

フォーマット

```
#include <ctype.h>
int isalnum(int c);
/* Test for upper- or lowercase letters, or decimal digit */
int isalpha(int c);
/* Test for alphabetic character */
int iscntrl(int c);
/* Test for any control character */
int isdigit(int c);
/* Test for decimal digit */
int isgraph(int c);
/* Test for printable character excluding space */
int islower(int c);
/* Test for lowercase */
int isprint(int c);
/* Test for printable character including space */
int ispunct(int c);
/* Test for any nonalphanumeric printable character */
/* excluding space */
int isspace(int c);
/* Test for whitespace character */
int isupper(int c);
/* Test for uppercase */
int isxdigit(int c);
/* Test for hexadecimal digit */
```

言語レベル: ANSI

スレッド・セーフ: はい。

ロケール依存: これらの関数の振る舞いは、現行ロケールの LC_CTYPE カテゴリーの影響を受ける可能性があります。詳細については、 549 ページの『CCSID およびロケールの理解』を参照してください。

説明

リストされている <ctype.h> 関数は、整数値を使用して文字をテストします。

戻り値

これらの関数は、整数がテスト条件を満たしている場合は非ゼロの値を戻し、テスト条件を満たしていない場合はゼロの値を戻します。整変数 *c* は、符号なし char 型 として表されていなければなりません。

注: EOF は入力値として妥当です。

<ctype.h> 関数の使用例

この例では、コード 0x0 とコード UPPER_LIMIT の間のすべての文字を分析し、英字の場合は A、英数字の場合は AN、大文字の場合は U、小文字の場合は L、数字の場合は D、16 進数字の場合は X、スペースの場合は S、句読点の場合は PU、出力可能文字の場合は PR、GRAPHIC 文字の場合は G、および制御文字の場合は C を出力します。この例では、出力可能な場合にコードを出力します。

この例では 256 行の表が出力され、テストした属性を持つ 0 から 255 までの文字が示されます。

```
#include <stdio.h>
#include <ctype.h>

#define UPPER_LIMIT   0xFF

int main(void)
{
   int ch;

   for ( ch = 0; ch  <= UPPER_LIMIT; ++ch )
   {
      printf("%3d ", ch);
      printf("%#04x ", ch);
      printf("%3s ", isalnum(ch)  ? "AN" : " ");
      printf("%2s ", isalpha(ch)  ? "A"  : " ");
      printf("%2s",  iscntrl(ch)  ? "C"  : " ");
      printf("%2s",  isdigit(ch)  ? "D"  : " ");
      printf("%2s",  isgraph(ch)  ? "G"  : " ");
      printf("%2s",  islower(ch)  ? "L"  : " ");
      printf(" %c",  isprint(ch)  ? ch   : ' ');
      printf("%3s",  ispunct(ch)  ? "PU" : " ");
      printf("%2s",  isspace(ch)  ? "S"  : " ");
      printf("%3s",  isprint(ch)  ? "PR" : " ");
      printf("%2s",  isupper(ch)  ? "U"  : " ");
      printf("%2s",  isxdigit(ch) ? "X"  : " ");

      putchar('¥n');
   }
}
```

関連情報

- 434 ページの『tolower() - toupper() ― 英大/小文字の変換』
- 179 ページの『isblank() ―ブランクまたはタブ文字のテスト』
- 3 ページの『<ctype.h>』

isascii() ― 表示可能文字の ASCII 値としてのテスト

フォーマット

```
#include <ctype.h>
int isascii(int c);
```

言語レベル: XPG4

スレッド・セーフ: はい。

ロケール依存: この関数の振る舞いは、現行ロケールの LC_CTYPE カテゴリーの影響を受ける可能性があります。この関数は、コンパイル・コマンドに対して LOCALETYPE(*CLD) が指定されている場合には使用できません。詳細については、 549 ページの『CCSID およびロケールの理解』を参照してください。

説明

指定した文字が現行ロケールで有効な 7 ビット US-ASCII 文字として表示可能な場合に、`isascii()` 関数はテストを行います。

戻り値

c が現行ロケールの 7 ビット US-ASCII 文字セットで表示可能な場合、`isascii()` 関数は非ゼロを戻します。それ以外の場合は、0 を戻します。

`isascii()` の使用例

この例では、0x7c から 0x82 までの整数をテストし、整数が 7 ビット US-ASCII 文字セット内の文字で表示可能な場合は対応する文字を出力します。

```
#include <stdio.h>
#include <ctype.h>

int main(void)
{
   int ch;

   for (ch = 0x7c; ch <= 0x82; ch++) {
      printf("%#04x    ", ch);
      if (isascii(ch))
         printf("The character is %c¥n", ch);
      else
         printf("Cannot be represented by an ASCII character¥n");
   }
   return 0;
}
   /************************************************
      The output should be:

 0x7c    The character is @
 0x7d    The character is '
 0x7e    The character is =
 0x7f    The character is "
 0x80    Cannot be represented by an ASCII character
 0x81    The character is a
 0x82    The character is b

   ************************************************/
```

関連情報

- 176 ページの『isalnum() - isxdigit() — 整数値のテスト』
- 180 ページの『iswalnum() から iswxdigit() — ワイド整数値のテスト』
- 433 ページの『toascii() — 文字から ASCII で表現可能な文字への変換』
- 434 ページの『tolower() - toupper() — 英大/小文字の変換』
- 436 ページの『towlower() -towupper() — ワイド文字の英大/小文字の変換』
- 3 ページの『<ctype.h>』

isblank() ーブランクまたはタブ文字のテスト

フォーマット

```
#include <ctype.h>
int isblank(int c);
```

注: isblank() 関数は C++ のみがサポート対象となっており、C はサポート対象外です。

言語レベル: Extended

スレッド・セーフ: はい。

説明

isblank() 関数は、文字が EBCDIC スペースまたは EBCDIC タブ文字のいずれかの場合にのみテストを行います。

戻り値

isblank() 関数は、*c* が EBCDIC スペースまたは EBCDIC タブ文字のいずれかの場合に非ゼロを戻し、その他の場合に 0 を戻します。

isblank() の使用例

この例では、isblank() を使用していくつかの文字をテストします。

```
#include <stdio.h>
#include <ctype.h>

int main(void)
{
   char *buf = "a b¥tc";
   int i;

   for (i = 0; i < 5; i++) {
     if (isblank(buf[i]))
        printf("Character %d is not a blank.¥n", i);
      else
        printf("Character %d is a blank¥n", i);
   }
   return 0;
}

/*************************************
     The output should be

 Character 0 is not a blank.
 Character 1 is a blank.
 Character 2 is not a blank.
 Character 3 is a blank.
 Character 4 is not a blank.

*************************************/
```

関連情報

- 176 ページの『isalnum() - isxdigit() ― 整数値のテスト』
- 180 ページの『iswalnum() から iswxdigit() ― ワイド整数値のテスト』
- 177 ページの『isascii() ― 表示可能文字の ASCII 値としてのテスト』
- 434 ページの『tolower() - toupper() ― 英大/小文字の変換』

- 436 ページの『towlower() -towupper() ― ワイド文字の英大/小文字の変換』
- 3 ページの『<ctype.h>』

iswalnum() から iswxdigit() ― ワイド整数値のテスト

フォーマット

```
#include <wctype.h>
int iswalnum(wint_t wc);
int iswalpha(wint_t wc);
int iswcntrl(wint_t wc);
int iswdigit(wint_t wc);
int iswgraph(wint_t wc);
int iswlower(wint_t wc);
int iswprint(wint_t wc);
int iswpunct(wint_t wc);
int iswspace(wint_t wc);
int iswupper(wint_t wc);
int iswxdigit(wint_t wc);
```

言語レベル: ANSI

スレッド・セーフ: はい。

ロケール依存: コンパイル・コマンドで LOCALETYPE(*LOCALE) が指定される場合、これらの関数の振る舞いは、現行ロケールの LC_CTYPE カテゴリーの影響を受ける可能性があります。これらの関数の振る舞いは、LOCALETYPE(*LOCALEUCS2) オプションまたは LOCALETYPE(*LOCALEUTF) オプションのいずれかをコンパイル・コマンドで指定した場合、現行ロケールの LC_UNI_CTYPE カテゴリーによって影響を受ける可能性があります。これらの関数は、コンパイル・コマンドで LOCALETYPE(*CLD) が指定される場合には使用できません。詳細については、 549 ページの『CCSID およびロケールの理解』を参照してください。

ワイド文字関数: 詳細については、 553 ページの『ワイド文字』を参照してください。

説明

上記の関数はすべて <wctype.h> 内で宣言されており、指定されているワイド整数値をテストします。

wc の値は、現行ロケール内で有効な文字に対応するワイド文字コードでなければならないか、または、マクロ WEOF の値と等しくなければなりません。引数の値がその他の場合、振る舞いは予期できません。

このグループ内の各関数の説明が以下に続きます。

iswalnum()
: ワイド英数字をテストします。

iswalpha()
: 現行ロケールの alpha クラスで定義されている、ワイド英字をテストします。

iswcntrl()
: 現行ロケールの cntrl クラスで定義されている、ワイド制御文字をテストします。

iswdigit()
: 現行ロケールの digit クラスで定義されている、0 から 9 までのワイド 10 進数字をテストします。

iswgraph()
: 現行ロケールの graph クラスで定義されている、ワイド印字文字 (スペースではなく) をテストします。

iswlower()
: 原稿ロケールの lower クラスで定義されているワイド小文字、または `iswcntrl()`、`iswdigit()`、`iswspace()` 関数のどれもが真ではないワイド小文字をテストします。

iswprint()
: 現行ロケールの print クラスで定義されている、すべてのワイド印字文字をテストします。

iswpunct()
: 現行ロケールの punct クラスで定義されている、ワイド非英数字、非スペース文字をテストします。

iswspace()
: 現行ロケールの space クラスで定義されている、ワイド空白文字をテストします。

iswupper()
: 現行ロケールの upper クラスで定義されている、ワイド大文字をテストします。

iswxdigit()
: 現行ロケールの xdigit クラスで定義されている、0 から 9、a から f、または A から F までのワイド 16 進数字をテストします。

戻り値

これらの関数は、ワイド整数がテスト値を満たしている場合は非ゼロの値を戻し、満たしていない場合は 0 の値を戻します。*wc* の値は、ワイド符号なし char として表現可能なものでなくてはいけません。WEOF は入力値として妥当です。

例

```
#include <stdio.h>
#include <wctype.h>

int main(void)
{
   int wc;

   for (wc=0; wc <= 0xFF; wc++) {
      printf("%3d", wc);
      printf(" %#4x ", wc);
      printf("%3s", iswalnum(wc)  ? "AN" : " ");
      printf("%2s", iswalpha(wc)  ? "A"  : " ");
      printf("%2s", iswcntrl(wc)  ? "C"  : " ");
      printf("%2s", iswdigit(wc)  ? "D"  : " ");
      printf("%2s", iswgraph(wc)  ? "G"  : " ");
      printf("%2s", iswlower(wc)  ? "L"  : " ");
      printf(" %c", iswprint(wc)  ? wc   : ' ');
      printf("%3s", iswpunct(wc)  ? "PU" : " ");
      printf("%2s", iswspace(wc)  ? "S"  : " ");
      printf("%3s", iswprint(wc)  ? "PR" : " ");
      printf("%2s", iswupper(wc)  ? "U"  : " ");
      printf("%2s", iswxdigit(wc) ? "X"  : " ");

      putchar('¥n');
   }
}
```

関連情報

- 20 ページの『<wctype.h>』

iswctype() ― 文字プロパティーのテスト

フォーマット

```
#include <wctype.h>
int iswctype(wint_t wc, wctype_t wc_prop);
```

言語レベル: ANSI

スレッド・セーフ: はい。

ロケール依存: コンパイル・コマンドで LOCALETYPE(*LOCALE) が指定される場合、この関数の振る舞いは、現行ロケールの LC_CTYPE カテゴリーの影響を受ける可能性があります。この関数の振る舞いは、LOCALETYPE(*LOCALEUCS2) オプションまたは LOCALETYPE(*LOCALEUTF) オプションのいずれかをコンパイル・コマンドで指定した場合、現行ロケールの LC_UNI_CTYPE カテゴリーによって影響を受ける可能性があります。この関数は、コンパイル・コマンドに対して LOCALETYPE(*CLD) が指定されている場合には使用できません。詳細については、 549 ページの『CCSID およびロケールの理解』を参照してください。

ワイド文字関数: 詳細については、 553 ページの『ワイド文字』を参照してください。

説明

`iswctype()` 関数は、ワイド文字 *wc* がプロパティー *wc_prop* を持っているかどうかを判別します。*wc* の値が WEOF でもなく、マルチバイト文字に対応するワイド文字の値でもない場合は、その振る舞いは予期できません。 *wc_prop* の値が無効な場合 (つまり、前の `wctype()` 関数呼び出しで取得されたものでない、または *wc_prop* が後続の `setlocale()` 関数呼び出しによって無効になった場合)、その振る舞いは予期できません。

戻り値

`iswctype()` 関数は、ワイド文字 *wc* の値がプロパティー *wc_prop* を持っている場合、真を戻します。

以下のストリング alnum から xdigit は、標準文字クラスとして予約済みです。関数は、以下のとおりです。各関数につき、同等な isw*() 関数も示します。

```
iswctype(wc, wctype("alnum"));     /* is equivalent to */        iswalnum(wc);
iswctype(wc, wctype("alpha"));     /* is equivalent to */        iswalpha(wc);
iswctype(wc, wctype("cntrl"));     /* is equivalent to */        iswcntrl(wc);
iswctype(wc, wctype("digit"));     /* is equivalent to */        iswdigit(wc);
iswctype(wc, wctype("graph"));     /* is equivalent to */        iswgraph(wc);
iswctype(wc, wctype("lower"));     /* is equivalent to */        iswlower(wc);
iswctype(wc, wctype("print"));     /* is equivalent to */        iswprint(wc);
iswctype(wc, wctype("punct"));     /* is equivalent to */        iswpunct(wc);
iswctype(wc,wctype("space"));      /* is equivalent to */        iswspace(wc);
iswctype(wc, wctype("upper"));     /* is equivalent to */        iswupper(wc);
iswctype(wc, wctype("xdigit"));    /* is equivalent to */        iswxdigit(wc);
```

`iswctype()` の使用例

```
#include <stdio.h>
#include <wctype.h>

int main(void)
{
   int wc;

   for (wc=0; wc <= 0xFF; wc++) {
      printf("%3d", wc);
      printf(" %#4x ", wc);
      printf("%3s", iswctype(wc, wctype("alnum"))  ? "AN" : " ");
      printf("%2s", iswctype(wc, wctype("alpha"))  ? "A"  : " ");
      printf("%2s", iswctype(wc, wctype("cntrl"))  ? "C"  : " ");
      printf("%2s", iswctype(wc, wctype("digit"))  ? "D"  : " ");
      printf("%2s", iswctype(wc, wctype("graph"))  ? "G"  : " ");
      printf("%2s", iswctype(wc, wctype("lower"))  ? "L"  : " ");
      printf(" %c", iswctype(wc, wctype("print"))  ? wc   : ' ');
      printf("%3s", iswctype(wc, wctype("punct"))  ? "PU" : " ");
      printf("%2s", iswctype(wc, wctype("space"))  ? "S"  : " ");
      printf("%3s", iswctype(wc, wctype("print"))  ? "PR" : " ");
      printf("%2s", iswctype(wc, wctype("upper"))  ? "U"  : " ");
      printf("%2s", iswctype(wc, wctype("xdigit")) ? "X"  : " ");

      putchar('¥n');
   }
}
```

関連情報

- 517 ページの『wctype() — 文字特性種別のハンドルの取得』
- 180 ページの『iswalnum() から iswxdigit() — ワイド整数値のテスト』
- 20 ページの『<wctype.h>』

_itoa - 整数からストリングへの変換

フォーマット

```
#include <stdlib.h>
char *_itoa(int value, char *string, int radix);
```

注: _itoa 関数は C++ のみがサポート対象となっており、C はサポート対象外です。

言語レベル: Extension

スレッド・セーフ: はい。

説明

_itoa() は、指定した *value* の数字をヌル文字で終わる文字ストリングに変換し、その結果を *string* に保管します。 *radix* 引数は、*value* の基数を指定します。2 から 36 の範囲でなければなりません。*radix* が 10 と等しく、*value* が負の場合、保管されているストリングの先頭文字は負符号 (-) です。

注: *string* 用に予約済みのスペースは、戻されるストリングを保持するのに十分な大きさでなければなりません。この関数は、ヌル文字 (¥0) を含めて最大 33 バイトまで戻すことができます。

戻り値

_itoa はポインターを *string* に戻します。エラーの戻り値はありません。

string 引数が NULL、または *radix* が 2 から 36 の範囲外にある場合、errno は EINVAL に設定されます。

_itoa() の使用例

この例では、整数値 -255 を 10 進数、バイナリー、および 16 進数に変換し、その文字表現を配列 *buffer* に保管します。

```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
   char buffer[35];
   char *p;
   p = _itoa(-255, buffer, 10);
   printf("The result of _itoa(-255) with radix of 10 is %s\n", p);
   p = _itoa(-255, buffer, 2);
   printf("The result of _itoa(-255) with radix of 2\n    is %s\n", p);
   p = _itoa(-255, buffer, 16);
   printf("The result of _itoa(-255) with radix of 16 is %s\n", p);
   return 0;
}
```

The output should be:

```
      The result of _itoa(-255) with radix of 10 is -255
      The result of _itoa(-255) with radix of 2
          is 11111111111111111111111100000001
      The result of _itoa(-255) with radix of 16 is ffffff01
```

関連情報

- 157 ページの『_gcvt - 浮動小数点からストリングへの変換』
- 183 ページの『_itoa - 整数からストリングへの変換』
- 200 ページの『_ltoa - long 型整数からストリングへの変換』
- 438 ページの『_ultoa - 符号なし long 型整数からストリングへの変換』
- 18 ページの『<stdlib.h>』

labs() — llabs() — long 型および long long 型整数の絶対値の計算

フォーマット (labs())

```
#include <stdlib.h>
long int labs(long int n);
```

フォーマット (llabs())

```
#include <stdlib.h>
long long int llabs(long long int i);
```

言語レベル: ANSI

スレッド・セーフ: はい。

説明

labs() 関数は、long 型整数である引数 *n* の絶対値を生成します。引数が、使用できる long 型整数の最小値である LONG_MIN と等しい場合、その結果は予期できません。値 LONG_MIN は、<limits.h> インクルード・ファイルで定義されます。

llabs() 関数は、long long 型整数オペランドの絶対値を戻します。引数が、使用できる long 型整数の最小値である LONG_LONG_MIN と等しい場合、その結果は予期できません。値 LONG_LONG_MIN は、<limits.h> インクルード・ファイルで定義されます。

戻り値

labs() 関数は、*n* の絶対値を戻します。エラーの戻り値はありません。

llabs() 関数は、*i* の絶対値を戻します。エラーの戻り値はありません。

labs() の使用例

この例では、*y* を long 型整数値 -41567 の絶対値として計算します。

```
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
   long x, y;

   x = -41567L;
   y = labs(x);

   printf("The absolute value of %ld is %ld¥n", x, y);
}

/********************  Output should be similar to:  **************

The absolute value of -41567 is 41567
*/
```

関連情報

- 39 ページの『abs() — 整数の絶対値の計算』
- 94 ページの『fabs() — 浮動小数点絶対値の計算』
- 8 ページの『<limits.h>』

ldexp() — 2 のべき乗の乗算

フォーマット

```
#include <math.h>
double ldexp(double x, int exp);
```

言語レベル: ANSI

スレッド・セーフ: はい。

説明

ldexp() 関数は、$x * (2^{exp})$ の値を計算します。

戻り値

ldexp() 関数は、$x*(2^{exp})$ の値を戻します。オーバーフローが起きた場合、関数は、結果が大きい場合には +HUGE_VAL を戻し、結果が小さい場合には -HUGE_VAL を戻し、errno に ERANGE を設定します。

ldexp() の使用例

この例では、*y* を 5 のべき乗の乗算 1.5 x 2 ($1.5*2^5$) として計算します。

```
#include <math.h>
#include <stdio.h>

int main(void)
{
   double x, y;
   int p;

   x = 1.5;
   p = 5;
   y = ldexp(x,p);

   printf("%lf times 2 to the power of %d is %lf¥n", x, p, y);
}

/*******************  Output should be similar to:  **************

1.500000 times 2 to the power of 5 is 48.000000
*/
```

関連情報

- 93 ページの『exp() ― 指数関数の計算』
- 137 ページの『frexp() ― 浮動小数点値の分離』
- 232 ページの『modf() ― 浮動小数点値の分離』
- 9 ページの『<math.h>』

ldiv() ― lldiv() ― long 型整数および long long 型整数の除算の実行

フォーマット (ldiv())

```
#include <stdlib.h>
ldiv_t ldiv(long int numerator, long int denominator);
```

フォーマット (lldiv())

```
#include <stdlib.h>
lldiv_t lldiv(long long int numerator, long long int denominator);
```

言語レベル: ANSI

スレッド・セーフ: はい。ただし、関数バージョンのみがスレッド・セーフです。マクロ・バージョンはスレッド・セーフではありません。

説明

ldiv() 関数は、*numerator* を *denominator* で割った商および剰余を計算します。

戻り値

ldiv() 関数は、商 (long int quot) と剰余 (long int rem) の両方を含む ldiv_t 型の構造体を戻します。値を表せない場合は、未定義な戻り値が戻される結果となります。*denominator* が 0 の場合は例外が発生します。

`lldiv()` サブルーチンは、*numerator* パラメーターを *denominator* パラメーターで割った商および剰余を計算します。

`lldiv()` サブルーチンは、商と剰余の両方からなる lldiv_t 型の構造体を戻します。構造体は、以下のように定義されます。

```
struct lldiv_t
{
long long int quot; /* quotient */
long long int rem; /* remainder */
};
```

除算が不正確だと、結果である商の符号は代数符号になり、結果である商の絶対値は代数符号の絶対値未満で最も大きい long long 型整数になります。結果を表せない場合 (例えば、*denominator* が 0 の場合)、振る舞いは予期できません。

`ldiv()` の使用例

この例では、`ldiv()` を使用して、2 つの被除数と 2 つの除数からなる 1 組のセットの商と剰余を計算します。

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
   long int num[2] = {45,-45};
   long int den[2] = {7,-7};
   ldiv_t ans;   /* ldiv_t is a struct type containing two long ints:
                    'quot' stores quotient; 'rem' stores remainder */
   short i,j;

   printf("Results of long division:¥n");
   for (i = 0; i < 2; i++)
      for (j = 0; j < 2; j++)
      {
         ans = ldiv(num[i], den[j]);
         printf("Dividend: %6ld  Divisor: %6ld", num[i], den[j]);
         printf("  Quotient: %6ld  Remainder: %6ld¥n", ans.quot, ans.rem);
      }
}

/********************  Expected output:  **************************

Results of long division:
Dividend:  45  Divisor:   7  Quotient:   6  Remainder:   3
Dividend:  45  Divisor:  -7  Quotient:  -6  Remainder:   3
Dividend: -45  Divisor:   7  Quotient:  -6  Remainder:  -3
Dividend: -45  Divisor:  -7  Quotient:   6  Remainder:  -3
*/
```

関連情報

- 90 ページの『div() ― 商および剰余の計算』
- 18 ページの『<stdlib.h>』

localeconv() ― 環境からの情報の取得

フォーマット

```
#include <locale.h>
struct lconv *localeconv(void);
```

言語レベル: ANSI

スレッド・セーフ: はい。

ロケール依存: この関数の振る舞いは、現行ロケールの LC_NUMERIC カテゴリーおよび LC_MONETARY カテゴリーの影響を受ける可能性があります。詳細については、549 ページの『CCSID およびロケールの理解』を参照してください。

説明

`localeconv()` は、型 struct lconv を持つ構造体のコンポーネントに、現行ロケールに適切な値を設定します。構造体は、`localeconv()` に対する別の呼び出しによって、または `setlocale()` 関数を呼び出すことによって上書きされる場合があります。

構造体には、以下のエレメントが含まれます (デフォルトで表示されるのは C ロケール用です)。

エレメント	エレメントの目的	デフォルト
char *decimal_point	非通貨数量をフォーマット設定するのに使用される小数点文字。	"."
char *thousands_sep	フォーマット済みの非通貨数量で小数点文字の左側の桁グループを分離するのに使用される文字。	""
char *grouping	フォーマット済みの非通貨数量でそれぞれの桁グループのサイズを示すストリング。ストリングの各文字には、グループ内の桁数が指定されます。最初の文字は、10 進数区切り文字のすぐ左方にあるグループのサイズを表します。これに続く文字では、前のグループの左方で、後に続くグループを定義します。最後の文字が UCHAR_MAX ではない場合、最後の文字をサイズとして使用して、グループ化が反復されます。最後の文字が UCHAR_MAX の場合、すでにストリング内にあるグループに対してのみグループ化が行われます (反復なし)。グループ化の処理方法については、 190 ページの表 1を参照してください。	""
char *int_curr_symbol	現行ロケール用の国際通貨記号。先頭の 3 文字には、英字の国際通貨記号が含まれています。4 番目の文字 (通常はスペース) は、通貨数量から国際通貨記号を分離するのに使用される文字です。	""
char *currency_symbol	現行ロケールのローカル通貨記号。	""
char *mon_decimal_point	通貨数量のフォーマット設定に使用される小数点文字。	""
char *mon_thousands_sep	フォーマット済みの通貨数量における数字の区切り文字。	""

エレメント	エレメントの目的	デフォルト
char *mon_grouping	フォーマット済みの通貨数量でそれぞれの桁グループのサイズを示すストリング。ストリングの各文字には、グループ内の桁数が指定されます。最初の文字は、10 進数区切り文字のすぐ左方にあるグループのサイズを表します。これに続く文字では、前のグループの左方で、後に続くグループを定義します。最後の文字が UCHAR_MAX ではない場合、最後の文字をサイズとして使用して、グループ化が反復されます。最後の文字が UCHAR_MAX の場合、すでにストリング内にあるグループに対してのみグループ化が行われます (反復なし)。グループ化の処理方法については、 190 ページの表 1を参照してください。	""
char *positive_sign	通貨数量で使用される正符号を示すストリング。	""
char *negative_sign	通貨数量で使用される負符号を示すストリング。	""
char int_frac_digits	国際的にフォーマット済みの通貨数量に対して、その小数部の右側に表示される桁の数。	UCHAR_MAX
char frac_digits	通貨数量における小数部の右側の桁の数。	UCHAR_MAX
char p_cs_precedes	currency_symbol が、負でないフォーマット済みの通貨数量の値の前に追加されている場合は 1、追加されていない場合は 0。	UCHAR_MAX
char p_sep_by_space	currency_symbol が、負でないフォーマット済みの通貨数量の値からスペースで分離されている場合は 1、分離されていない場合は 0。	UCHAR_MAX
char n_cs_precedes	currency_symbol が負のフォーマット済みの通貨数量の値の前に追加されている場合は 1、追加されていない場合は 0。	UCHAR_MAX
char n_sep_by_space	currency_symbol が、負のフォーマット済みの通貨数量の値からスペースで分離されている場合は 1、分離されていない場合は 0。	UCHAR_MAX
char p_sign_posn	負でないフォーマット済み通貨数量の positive_sign の位置を示す値。	UCHAR_MAX
char n_sign_posn	負のフォーマット済み通貨数量の negative_sign の位置を示す値。	UCHAR_MAX

"" の値のストリングへのポインターは、値が C ロケールでは使用できない、または長さが 0 であることを示します。値が UCHAR_MAX の char 型は、値が現行ロケールで使用できないことを示します。

n_sign_posn および p_sign_posn エレメントは、次の値を持つことができます。

値 **意味**

0 数量および currency_symbol は、括弧で囲まれています。

1 数量と currency_symbol の前に符号が付きます。

2 数量と currency_symbol の後に符号が付きます。

3 currency_symbol の前に符号が付きます。

4 currency_symbol の後に符号が付きます。

グループ化の例

表 1. グループ化の例

ロケール・ソース	グループ化ストリング	番号	フォーマット済み番号
-1	0x00	123456789	123456789
3	0x0300	123456789	123,456,789
3;-1	0x03FF00	123456789	123456,789
3;2;1	0x03020100	123456789	1,2,3,4,56,789

通貨フォーマットの例

表 2. 通貨フォーマットの例

国別	正フォーマット	負フォーマット	国際フォーマット
イタリア	L.1.230	-L.1.230	ITL.1.230
オランダ	F 1.234,56	F -1.234,56	NLG 1.234,56
ノルウェー	kr1.234,56	kr1.234,56-	NOK1.234,56
スイス	SFRs.1,234.56	SFrx.1,234.56C	CHF 1,234.56

上記の表は、次の通貨フィールドを使用したロケールで生成されています。

表 3. 通貨フィールド

	イタリア	オランダ	ノルウェー	スイス
int_curr_symbol	″ITL.″	″NLG″	″NOK″	″CHF″
currency_symbol	″L.″	″F″	″kr″	″SFrs.″
mon_decimal_point	″″	″,″	″,″	″.″
mon_thousands_sep	″.″	″.″	″.″	″,″
mon_grouping	″¥3″	″¥3″	″¥3″	″¥3″
positive_sign	″″	″″	″″	″″
negative_sign	″-″	″-″	″-″	″C″
int_frac_digits	0	2	2	2
frac_digits	0	2	2	2
p_cs_precedes	1	1	1	1
p_sep_by_space	0	1	0	0
n_cs_preceds	1	1	1	1
n_sep_by_space	0	1	0	0
p_sep_posn	1	1	1	1
n_sign_posn	1	4	2	2

戻り値

`localeconv()` 関数は、構造体へのポインターを戻します。

*CLD ロケール・オブジェクトの使用例

この例では、お客様のロケールのデフォルトの小数点を出力し、次に LC_C_FRANCE ロケールの小数点を出力します。

```
#include <stdio.h>
#include <locale.h>

int main(void) {

   char * string;
   struct lconv * mylocale;
   mylocale = localeconv();

   /* Display default decimal point */

   printf("Default decimal point is a %s¥n", mylocale->decimal_point);

   if (NULL != (string = setlocale(LC_ALL, LC_C_FRANCE))) {
       mylocale = localeconv();

      /* A comma is set to be the decimal point when the locale is LC_C_FRANCE*/

      printf("France's decimal point is a %s¥n", mylocale->decimal_point);

   } else {

      printf("setlocale(LC_ALL, LC_C_FRANCE) returned <NULL>¥n");
   }
   return 0;
}
```

*LOCALE オブジェクトの使用例

```
/************************************************************************
This example prints out the default decimal point for
the C locale and then the decimal point for the French
locale using a *LOCALE object called
"QSYS.LIB/MYLIB.LIB/LC_FRANCE.LOCALE".

 Step 1: Create a French *LOCALE object by entering the command
   CRTLOCALE LOCALE('QSYS.LIB/MYLIB.LIB/LC_FRANCE.LOCALE') +
             SRCFILE('QSYS.LIB/QSYSLOCALE.LIB/QLOCALESRC.FILE/ +
                      FR_FR.MBR') CCSID(297)                            *
 Step 2: Compile the following C source, specifying
         LOCALETYPE(*LOCALE) on the compilation command.
 Step 3: Run the program.
************************************************************************/

#include <stdio.h>
#include <locale.h>
int main(void) {
  char * string;
  struct lconv * mylocale;
  mylocale = localeconv();

  /* Display default decimal point */
  printf("Default decimal point is a %s¥n", mylocale->decimal_point);
  if (NULL != (string = setlocale(LC_ALL,
      "QSYS.LIB/MYLIB.LIB/LC_FRANCE.LOCALE"))) {
     mylocale = localeconv();

     /* A comma is set to be the decimal point in the French locale */
     printf("France's decimal point is a %s¥n", mylocale->decimal_point);
  } else {
     printf("setlocale(LC_ALL, ¥"QSYS.LIB/MYLIB.LIB/LC_FRANCE.LOCALE¥") ¥
             returned <NULL>¥n");
  }
  return 0;
}
```

関連情報

- 354 ページの『setlocale() ― ロケールの設定』
- 8 ページの『<locale.h>』

localtime() ― 時間の変換

フォーマット

```
#include <time.h>
struct tm *localtime(const time_t *timeval);
```

言語レベル: ANSI

スレッド・セーフ: いいえ。代わりに `localtime_r()` を使用します。

ロケール依存: この関数の振る舞いは、現行ロケールの LC_TOD カテゴリーの影響を受ける可能性があります。

説明

`localtime()` 関数は、秒単位で、time 値を *tm* 構造体に変換します。

`localtime()` 関数は、*timeval* を協定世界時 (UTC) として仮定し、ジョブ・ロケール時間に変換します。この変換を行うため、`localtime()` はローカル時間帯および夏時間 (DST) の現行ロケール設定をチェック

します。これらの値が現行ロケールで未設定の場合、localtime() はローカル時間帯および夏時間 (DST) の設定を現行ジョブから取得します。変換が行われると、時間は型 *tm* の構造体内に戻されます。 DST がロケールで設定済みでも、時間帯情報が未設定の場合は、ロケール内の DST 情報は無視されます。

time 値は、通常、time() 関数を呼び出して取得します。

注:

1. gmtime() および localtime() 関数は、静的に割り振られた共通のバッファーを変換に使用します。これらの関数の 1 つを呼び出すごとに、以前の呼び出しの結果が破棄される可能性があります。 ctime_r()、gmtime_r()、および localtime_r() 関数は、静的に割り振られた共通のバッファーを使用しません。これらの関数は、再入可能性が望ましい場合に、asctime()、ctime()、gmtime() および localtime() 関数の代わりに使用可能です。
2. カレンダー時間とは、エポックである UTC (1970 年 1 月 1 日 00:00:00) から数えた秒数です。

戻り値

localtime() 関数は、構造体へのポインターを戻します。エラーの戻り値はありません。

localtime() の使用例

この例では、システム・クロックを照会して、現地時間を表示します。

```
#include <time.h>
#include <stdio.h>

int main(void)
{
   struct tm *newtime;
   time_t ltime;

   ltime = time(&ltime);
   newtime = localtime(&ltime);
   printf("The date and time is %s", asctime(newtime));}

/**************  If the local time is 3 p.m. February 15, 2008, **********
*************************  the output should be:  *********************

The date and time is Fri Feb 15 15:00:00 2008
*/
```

関連情報

- 41 ページの『asctime() — 時間から文字ストリングへの変換』
- 43 ページの『asctime_r() — 時間から文字ストリングへの変換 (再始動可能)』
- 75 ページの『ctime() — 時間から文字ストリングへの変換』
- 77 ページの『ctime64() — 時間から文字ストリングへの変換』
- 80 ページの『ctime64_r() — 時間から文字ストリングへの変換 (再始動可能)』
- 78 ページの『ctime_r() — 時間から文字ストリングへの変換 (再始動可能)』
- 167 ページの『gmtime() — 時間の変換』
- 169 ページの『gmtime64() — 時間の変換』
- 173 ページの『gmtime64_r() — 時間の変換 (再始動可能)』
- 171 ページの『gmtime_r() — 時間の変換 (再始動可能)』
- 195 ページの『localtime_r() — 時間の変換 (再始動可能)』

- 『localtime64() ― 時間の変換』
- 197 ページの『localtime64_r() ― 時間の変換 (再始動可能)』
- 228 ページの『mktime() ― 地方時の変換』
- 230 ページの『mktime64() ― 地方時の変換』
- 354 ページの『setlocale() ― ロケールの設定』
- 429 ページの『time() ― 現在時刻の判別』
- 430 ページの『time64() ― 現在時刻の判別』
- 19 ページの『<time.h>』

localtime64() ― 時間の変換

フォーマット

```
#include <time.h>
struct tm *localtime64(const time64_t *timeval);
```

言語レベル: ILE C Extension

スレッド・セーフ: いいえ。代わりに `localtime64_r()` を使用します。

ロケール依存: この関数の振る舞いは、現行ロケールの LC_TOD カテゴリーの影響を受ける可能性があります。

説明

`localtime64()` 関数は、秒単位で、time 値を *tm* 構造体に変換します。

`localtime64()` 関数は、*timeval* を協定世界時 (UTC) として仮定し、ジョブ・ロケール時間に変換します。この変換を行うため、`localtime64()` はローカル時間帯および夏時間 (DST) の現行ロケール設定をチェックします。これらの値が現行ロケールで未設定の場合、`localtime64()` はローカル時間帯および夏時間 (DST) の設定を現行ジョブから取得します。変換が行われると、時間は型 *tm* の構造体内に戻されます。DST がロケールで設定済みでも、時間帯情報が未設定の場合は、ロケール内の DST 情報は無視されます。

time 値は、通常、`time64()` 関数を呼び出して取得します。

注:

1. `gmtime64()` および `localtime64()` 関数は、静的に割り振られた共通のバッファーを変換に使用します。これらの関数の 1 つを呼び出すごとに、以前の呼び出しの結果を変更します。`asctime_r()`、`ctime64_r()`、`gmtime64_r()`、および `localtime64_r()` 関数は、静的に割り振られた共通のバッファーを使用しません。これらの関数は、スレッド・セーフティーが望ましい場合に、`asctime()`、`ctime64()`、`gmtime64()`、および `localtime64()` 関数の代わりに使用可能です。
2. カレンダー時間とは、エポックである UTC (1970 年 1 月 1 日 00:00:00) から数えた秒数です。
3. この関数でサポートされる日付と時刻の範囲は、01/01/0001 00:00:00 から 12/31/9999 23: 59: 59 です。

戻り値

`localtime64()` 関数は、結果である構造体へのポインターを戻します。指定された *timeval* が範囲外の場合は、NULL ポインターが戻され、errno は EOVERFLOW に設定されます。

localtime64() の使用例

この例では、システム・クロックを照会して、現地時間を表示します。

```
#include <stdio.h>
#include <time.h>

int main(void)
{
   struct tm *newtime;
   time64_t ltime;

   ltime = time64(&ltime);
   newtime = localtime64(&ltime);
   printf("The date and time is %s", asctime(newtime));
}

/**************  If the local time is 3 p.m. February 15, 2008, **********
*************************  the output should be: *********************

The date and time is Fri Feb 15 15:00:00 2008
*/
```

関連情報

- 41 ページの『asctime() — 時間から文字ストリングへの変換』
- 43 ページの『asctime_r() — 時間から文字ストリングへの変換 (再始動可能)』
- 75 ページの『ctime() — 時間から文字ストリングへの変換』
- 77 ページの『ctime64() — 時間から文字ストリングへの変換』
- 80 ページの『ctime64_r() — 時間から文字ストリングへの変換 (再始動可能)』
- 78 ページの『ctime_r() — 時間から文字ストリングへの変換 (再始動可能)』
- 167 ページの『gmtime() — 時間の変換』
- 169 ページの『gmtime64() — 時間の変換』
- 173 ページの『gmtime64_r() — 時間の変換 (再始動可能)』
- 171 ページの『gmtime_r() — 時間の変換 (再始動可能)』
- 192 ページの『localtime() — 時間の変換』
- 197 ページの『localtime64_r() — 時間の変換 (再始動可能)』
- 『localtime_r() — 時間の変換 (再始動可能)』
- 228 ページの『mktime() — 地方時の変換』
- 230 ページの『mktime64() — 地方時の変換』
- 354 ページの『setlocale() — ロケールの設定』
- 429 ページの『time() — 現在時刻の判別』
- 430 ページの『time64() — 現在時刻の判別』
- 19 ページの『<time.h>』

localtime_r() — 時間の変換 (再始動可能)

フォーマット

```
#include <time.h>
struct tm *localtime_r(const time_t *timeval, struct tm *result);
```

言語レベル: XPG4

スレッド・セーフ: はい

ロケール依存: この関数の振る舞いは、現行ロケールの LC_TOD カテゴリーの影響を受ける可能性があります。

説明

この関数は、`localtime()` の再始動可能バージョンです。戻される構造体 *result* が保管される場所を渡す点を除き、`localtime()` と同じです。

戻り値

`localtime_r()` は、構造体 result へのポインターを戻します。エラーの戻り値はありません。

`localtime_r()` の使用例

この例では、システム・クロックを照会して、現地時間を表示します。

```
#include <time.h>
#include <stdio.h>

int main(void)
{
   struct tm newtime;
   time_t ltime;
   char buf[50];

   ltime=time(&ltime);
   localtime_r(&ltime, &newtime);
   printf("The date and time is %s", asctime_r(&newtime, buf));
}

/**************  If the local time is 3 p.m. February 15, 2008,  **********
*************************  the output should be:  *********************

The date and time is Fri Feb 15 15:00:00 2008
*/
```

関連情報

- 41 ページの『asctime() ― 時間から文字ストリングへの変換』
- 43 ページの『asctime_r() ― 時間から文字ストリングへの変換 (再始動可能)』
- 75 ページの『ctime() ― 時間から文字ストリングへの変換』
- 78 ページの『ctime_r() ― 時間から文字ストリングへの変換 (再始動可能)』
- 167 ページの『gmtime() ― 時間の変換』
- 171 ページの『gmtime_r() ― 時間の変換 (再始動可能)』
- 192 ページの『localtime() ― 時間の変換』
- 228 ページの『mktime() ― 地方時の変換』
- 429 ページの『time() ― 現在時刻の判別』
- 19 ページの『<time.h>』

localtime64_r() — 時間の変換 (再始動可能)

フォーマット

```
#include <time.h>
struct tm *localtime64_r(const time64_t *timeval, struct tm *result);
```

言語レベル: ILE C Extension

スレッド・セーフ: はい。

ロケール依存: この関数の振る舞いは、現行ロケールの LC_TOD カテゴリーの影響を受ける可能性があります。

説明

この関数は、`localtime64()` の再始動可能バージョンです。場所内を通過して、戻される構造体 *result* を保管する点を除き、`localtime64()` と同じです。

注:

1. `gmtime64()` および `localtime64()` 関数は、静的に割り振られた共通のバッファーを変換に使用します。これらの関数の 1 つを呼び出すごとに、以前の呼び出しの結果を変更します。`asctime_r()`、`ctime64_r()`、`gmtime64_r()`、および `localtime64_r()` 関数は、戻りストリングの保持用に静的に割り振られた共通のバッファーを使用しません。これらの関数は、スレッド・セーフティーが望ましい場合に、`asctime()`、`ctime64()`、`gmtime64()`、および `localtime64()` 関数の代わりに使用可能です。
2. カレンダー時間とは、エポックである UTC (1970 年 1 月 1 日 00:00:00) から数えた秒数です。
3. この関数でサポートされる日付と時刻の範囲は、01/01/0001 00:00:00 から 12/31/9999 23: 59: 59 です。

戻り値

`localtime64_r()` 関数は、結果である構造体へのポインターを戻します。指定された *timeval* が範囲外の場合は、NULL ポインターが戻され、errno は EOVERFLOW に設定されます。

`localtime64_r()` の使用例

この例では、システム・クロックを照会して、現地時間を表示します。

```
#include <stdio.h>
#include <time.h>

int main(void)
{
   struct tm newtime;
   time64_t ltime;
   char buf[50];

   ltime = time64(&ltime);
   localtime64_r(&ltime, &newtime);
   printf("The date and time is %s¥n", asctime_r(&newtime, buf));
}

/**************  If the local time is 3 p.m. February 15, 2008, **********
*************************  the output should be: *********************

The date and time is Fri Feb 15 15:00:00 2008
*/
```

関連情報

- 41 ページの『asctime() ― 時間から文字ストリングへの変換』
- 43 ページの『asctime_r() ― 時間から文字ストリングへの変換 (再始動可能)』
- 77 ページの『ctime64() ― 時間から文字ストリングへの変換』
- 80 ページの『ctime64_r() ― 時間から文字ストリングへの変換 (再始動可能)』
- 169 ページの『gmtime64() ― 時間の変換』
- 173 ページの『gmtime64_r() ― 時間の変換 (再始動可能)』
- 194 ページの『localtime64() ― 時間の変換』
- 230 ページの『mktime64() ― 地方時の変換』
- 430 ページの『time64() ― 現在時刻の判別』
- 19 ページの『<time.h>』

log() ― 自然対数の計算

フォーマット

```
#include <math.h>
double log(double x);
```

言語レベル: ANSI

スレッド・セーフ: はい。

説明

`log()` 関数は、*x* の自然対数 (基数 e) を計算します。

戻り値

`log()` 関数は、計算値を戻します。 *x* が負の場合、`log()` は errno を EDOM に設定し、値 -HUGE_VAL を戻す場合があります。*x* がゼロの場合、`log()` は値 -HUGE_VAL を戻し、errno を ERANGE に設定する場合があります。

`log()` の使用例

この例では、1000.0 の対数を計算します。

```
#include <math.h>
#include <stdio.h>

int main(void)
{
   double x = 1000.0, y;

   y = log(x);

   printf("The natural logarithm of %lf is %lf¥n", x, y);
}

/********************  Output should be similar to:  **************

The natural logarithm of 1000.000000 is 6.907755
*/
```

関連情報

- 93 ページの『exp() ― 指数関数の計算』
- 『log10() ― 基数 10 の対数の計算』
- 237 ページの『pow() ― 累乗の計算』
- 9 ページの『<math.h>』

log10() ― 基数 10 の対数の計算

フォーマット

```
#include <math.h>
double log10(double x);
```

言語レベル: ANSI

スレッド・セーフ: はい。

説明

`log10()` 関数は、*x* の 10 を基数とする対数を計算します。

戻り値

`log10()` 関数は、計算値を戻します。 *x* が負の場合、`log10()` は errno を EDOM に設定し、値 -HUGE_VAL を戻す場合があります。*x* がゼロの場合、`log10()` は値 -HUGE_VAL を戻し、errno を ERANGE に設定する場合があります。

`log10()` の使用例

この例では、1000.0 の 10 を基数とする対数を計算します。

```
#include <math.h>
#include <stdio.h>

int main(void)
{
   double x = 1000.0, y;

   y = log10(x);

   printf("The base 10 logarithm of %lf is %lf¥n", x, y);
}

/********************  Output should be similar to:  **************

The base 10 logarithm of 1000.000000 is 3.000000
*/
```

関連情報

- 93 ページの『exp() ― 指数関数の計算』
- 198 ページの『log() ― 自然対数の計算』
- 237 ページの『pow() ― 累乗の計算』
- 9 ページの『<math.h>』

_ltoa - long 型整数からストリングへの変換

フォーマット

```
#include <stdlib.h>
char *_ltoa(long value, char *string, int radix);
```

注: _ltoa 関数は C++ のみがサポート対象となっており、C はサポート対象外です。

言語レベル: Extension

スレッド・セーフ: はい。

説明

`_ltoa` は、指定された long 型整数 *value* の数字をヌル文字で終わる文字ストリングに変換し、その結果を *string* に保管します。 *radix* 引数は、*value* の基数を指定します。2 から 36 の範囲でなければなりません。*radix* が 10 と等しく、*value* が負の場合、保管されているストリングの先頭文字は負符号 (-) です。

注: *string* 用に割り振られたスペースは、戻されるストリングを保持するために十分な大きさでなければなりません。この関数は、ヌル文字 (¥0) を含めて最大 33 バイトまで戻すことができます。

戻り値

`_ltoa` はポインターを *string* に戻します。エラーの戻り値はありません。

string 引数が NULL、または *radix* が 2 から 36 の範囲外にある場合、`errno` は EINVAL に設定されます。

`_ltoa()` の使用例

この例では、整数値 -255L を 10 進数、バイナリー、および 16 進数値に変換し、その文字表現を配列 *buffer* に保管します。

```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
   char buffer[35];
   char *p;
   p = _ltoa(-255L, buffer, 10);
   printf("The result of _ltoa(-255) with radix of 10 is %s\n", p);
   p = _itoa(-255L, buffer, 2);
   printf("The result of _ltoa(-255) with radix of 2\n    is %s\n", p);
   p = _itoa(-255L, buffer, 16);
   printf("The result of _ltoa(-255) with radix of 16 is %s\n", p);
   return 0;
}
```

The output should be:

```
The result of _ltoa(-255) with radix of 10 is -255
The result of _ltoa(-255) with radix of 2
    is 11111111111111111111111100000001
The result of _ltoa(-255) with radix of 16 is ffffff01
```

関連情報

- 52 ページの『atol() ― atoll() ― 文字ストリングの long 型整数または long long 型整数への変換』
- 157 ページの『_gcvt - 浮動小数点からストリングへの変換』
- 183 ページの『_itoa - 整数からストリングへの変換』
- 418 ページの『strtol() ― strtoll() ― 文字ストリングから long 型および long long 型整数への変換』
- 438 ページの『_ultoa - 符号なし long 型整数からストリングへの変換』
- 503 ページの『wcstol() ― wcstoll() ― ワイド文字ストリングから long 型および long long 型整数への変換』
- 18 ページの『<stdlib.h>』

longjmp() ― スタック環境の復元

フォーマット

```
#include <setjmp.h>
void longjmp(jmp_buf env, int value);
```

言語レベル: ANSI

スレッド・セーフ: はい。

説明

`longjmp()` 関数は、以前 `setjmp()` 関数によって *env* に保管されているスタック環境を復元します。`setjmp()` および `longjmp()` 関数は、非ローカル goto を実行する方法を提供します。これらは、シグナル・ハンドラーでよく使用されます。

`setjmp()` 関数の呼び出しにより、現行スタック環境が *env* に保管されます。次に `longjmp()` を呼び出すと、保管した環境が復元され、`setjmp()` 呼び出しに対応するプログラム内のポイントへ制御が戻されます。 `setjmp()` 呼び出しによって、指定 *value* が戻された時点から処理が再開されます。

制御を受け取る関数で使用可能なすべての変数 (register 変数を除く) には、`longjmp()` が呼び出されたときに設定されていた値が入ります。レジスター変数の値は、予測不可能です。`setjmp()` および `longjmp()` 関数の間の呼び出しで変更される、不揮発性 auto 変数も予測不可能です。

注: `setjmp()` を呼び出す関数が、対応する `longjmp()` 関数を呼び出す前に戻らないことを確認してください。 `setjmp()` を呼び出す関数が戻った後に `longjmp()` を呼び出すと、予測不可能なプログラムの振る舞いを引き起こします。

引数 *value* は非ゼロでなければなりません。ゼロ引数を *value* に指定すると、`longjmp()` により 1 に置換されます。

戻り値

`longjmp()` 関数は、普通の関数呼び出しや戻りメカニズムを使用しません。したがって、戻り値はありません。

`longjmp()` の使用例

この例では、下のステートメントで、スタック環境を保存します。

```
if(setjmp(mark) != 0) ...
```

システムは、最初に if ステートメントを実行する場合、環境を mark に保管し、条件を FALSE に設定します。これは、`setjmp()` 関数が環境を保管するとき 0 を戻すためです。プログラムで次のメッセージが出力されます。

```
setjmp has been called
```

関数 `p()` への以降の呼び出しで、`longjmp()` 関数が呼び出されます。制御は、*mark* 変数内に保存された環境を使用して、`setjmp()` 関数への呼び出しの直後に `main()` 関数内のポイントに渡されます。この場合、条件は TRUE ですが、これは戻り値がスタック上に置かれるよう、`longjmp()` 関数呼び出しの 2 番目のパラメーターで -1 が指定されているためです。次に、この例ではブロック中のステートメントが実行され、"`longjmp()` has been called" というメッセージが出力されます。その後、`recover()` 関数が実行され、プログラムが終了します。

```
#include <stdio.h>
#include <setjmp.h>
#include <stdlib.h>

jmp_buf mark;

void p(void);
void recover(void);

int main(void)
{
   if (setjmp(mark) != 0)
   {
      printf("longjmp has been called¥n");
      recover();
      exit(1);
      }
   printf("setjmp has been called¥n");
   printf("Calling function p()¥n");
   p();
   printf("This point should never be reached¥n");
}

void p(void)
{
   printf("Calling longjmp() from inside function p()¥n");
   longjmp(mark, -1);
   printf("This point should never be reached¥n");
}

void recover(void)
{
   printf("Performing function recover()¥n");
}
/*******************Output should be as follows: **********************
 setjmp has been called
 Calling function p()
 Calling longjmp() from inside function p()
 longjmp has been called
 Performing function recover()
**********************************************************************/
```

関連情報

- 352 ページの『setjmp() ― 環境の保存』
- 14 ページの『<setjmp.h>』

malloc() ― ストレージ・ブロックの予約

フォーマット

```
#include <stdlib.h>
void *malloc(size_t size);
```

言語レベル: ANSI

スレッド・セーフ: はい。

説明

`malloc()` 関数は、*size* バイトのストレージ・ブロックを予約します。 `calloc()` 関数とは異なり、`malloc()` はすべてのエレメントを 0 に初期化するわけではありません。非テラスペース `malloc()` の最大サイズは 16,711,568 バイトです。

注:

1. すべてのヒープ・ストレージは、呼び出しルーチンの活動化グループと関連付けられます。したがって、ストレージの割り振りと割り振り解除は同じ活動化グループ内で行ってください。 1 つの活動化グループ内でヒープ・ストレージを割り振ったり、異なる活動化グループからそのストレージを割り振り解除したりすることはできません。活動化グループについて詳しくは、「*ILE Concepts*」のマニュアルを参照してください。
2. C ソース・コードを変更せずに、単一レベル・ストア・ストレージの代わりにテラスペース・ストレージを使用する場合、コンパイラー・コマンドで TERASPACE(*YES *TSIFC) パラメーターを指定します。これにより、`malloc()` ライブラリー関数が `_C_TS_malloc()` (テラスペース・ストレージでの `malloc()` ライブラリー関数のカウンター・パート) にマップされます。 `_C_TS_malloc()` への各呼び出しにより割り振り可能なテラスペース・ストレージの最大量は、2GB - 224 バイト、つまり 2,147,483,424 バイトです。単一の要求で 2147483408 バイトより多く必要な場合は、`_C_TS_malloc64` (符号なし `long long` 型整数) を呼び出します。

   詳しくは、「*ILE Concepts*」を参照してください。
3. 活動化グループ内の MI プログラムで使用中のテラスペース・ストレージに関する現在の統計に対しては、`_C_TS_malloc_info` 関数を呼び出します。この関数は、全バイト数、割り振りバイト数とブロック数、割り振られていないバイト数とブロック数、要求バイト数、埋め込みバイト数、およびオーバーヘッド・バイト数などの情報を戻します。 `_C_TS_malloc()` 関数および `_C_TS_malloc64()` 関数で使用されるメモリー構造に関する詳細情報を得る場合は、`_C_TS_malloc_debug()` 関数を呼び出します。この関数が戻す情報を利用して、メモリーの破損問題を識別することができます。
4. 高速プール・メモリー・マネージャーが現行の活動化グループ内で使用可能にされている場合、ストレージは高速プール・メモリー・マネージャーを使用して検索されます。詳細については、 71 ページの『_C_Quickpool_Init() ー 高速プール・メモリー・マネージャーの初期化』を参照してください。

戻り値

`malloc()` 関数は、予約したスペースを指すポインターを戻します。戻り値が示すストレージ・スペースは、任意のタイプのオブジェクトのストレージ用に適切に位置合わせされます。十分なストレージが使用できない場合、あるいは *size* がゼロと指定してある場合には、戻り値は NULL になります。

`malloc()` の使用例

この例では、必要とされる配列項目の数を求めるプロンプトを出し、次にその項目がストレージで必要とするスペースを予約します。この例では、`malloc()` が正常の場合は、項目に値を割り当てて各項目を出力し、正常ではない場合は、エラーを出力します。

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
  long * array;    /* start of the array */
  long * index;    /* index variable     */
  int    i;        /* index variable     */
  int  num;        /* number of entries of the array */

  printf( "Enter the size of the array¥n" );
  scanf( "%i", &num );

  /* allocate num entries */
  if ( (index = array = (long *) malloc( num * sizeof( long ))) != NULL )
  {

    for ( i = 0; i < num; ++i )           /* put values in array    */
       *index++ = i;                      /* using pointer notation */

    for ( i = 0; i < num; ++i )           /* print the array out    */
      printf( "array[ %i ] = %i¥n", i, array[i] );
  }
  else { /* malloc error */
    perror( "Out of storage" );
    abort();
  }
}

/********************  Output should be similar to:  **************

Enter the size of the array
array[ 0 ] = 0
array[ 1 ] = 1
array[ 2 ] = 2
array[ 3 ] = 3
array[ 4 ] = 4
*/
```

関連情報

- 58 ページの『calloc() ― ストレージの予約と初期化』
- 69 ページの『_C_Quickpool_Debug() ― 高速プール・メモリー・マネージャー特性の変更』
- 71 ページの『_C_Quickpool_Init() ― 高速プール・メモリー・マネージャーの初期化』
- 73 ページの『_C_Quickpool_Report() ― 高速プール・メモリー・マネージャー・レポートの生成』
- 134 ページの『free() ― ストレージ・ブロックの解放』
- 276 ページの『realloc() ― 予約ストレージ・ブロック・サイズの変更』
- 564 ページの『ヒープ・メモリー』
- 18 ページの『<stdlib.h>』

mblen() ― マルチバイト文字の長さの計算

フォーマット

```
#include <stdlib.h>
int mblen(const char *string, size_t n);
```

言語レベル: ANSI

スレッド・セーフ: いいえ。代わりに mbrlen() を使用します。

ロケール依存: この関数の振る舞いは、現行ロケールの LC_CTYPE カテゴリーの影響を受ける可能性があります。この関数は、LOCALETYPE(*LOCALEUCS2) または LOCALETYPE(*LOCALEUTF) をコンパイル・コマンドで指定した場合、現行ロケールの LC_UNI_CTYPE カテゴリーによって影響を受ける可能性があります。詳細については、549 ページの『CCSID およびロケールの理解』を参照してください。

説明

`mblen()` 関数は、*string* が示すマルチバイト文字の長さ (バイト) を判別します。 *n* は、検査される最大バイト数を表します。

戻り値

ストリングが NULL の場合、`mblen()` 関数は以下を戻します。

- アクティブなロケールが混合バイトのストリングを許可する場合は非ゼロ。関数は、状態変数を初期化します。
- それ以外の場合は、ゼロ。

string が NULL 以外の場合、`mblen()` 関数は以下を戻します。

- *string* がヌル文字を指す場合は、ゼロ。
- マルチバイト文字を構成するバイト数。
- *string* が有効なマルチバイト文字を指さない場合は、-1。

注: `mblen()`、`mbtowc()`、および `wctomb()` 関数は、それぞれが静的に割り振られたストレージを使用するため、再始動できません。ただし、`mbrlen()`、`mbrtowc()`、および `wcrtomb()` は再始動可能です。

`mblen()` **の使用例**

この例では、`mblen()` および `mbtowc()` を使用して、マルチバイト文字を単一のワイド文字に変換します。

```
#include <stdio.h>
#include <stdlib.h>

int length, temp;
char string [6] = "w";
wchar_t arr[6];

int main(void)

{
   /* Initialize internal state variable */
   length = mblen(NULL, MB_CUR_MAX);

   /* Set string to point to a multibyte character  */
   length = mblen(string, MB_CUR_MAX);
   temp = mbtowc(arr,string,length);
   arr[1] = L'¥0';
   printf("wide character string: %ls¥n", arr);
}
```

関連情報

- 207 ページの『mbrlen() ― マルチバイト文字の長さの計算 (再始動可能)』
- 220 ページの『mbtowc() ― マルチバイト文字からワイド文字への変換』
- 216 ページの『mbstowcs() ― マルチバイト・ストリングからワイド文字ストリングへの変換』
- 392 ページの『strlen() ― ストリング長の判別』

- 482 ページの『wcslen() ― ワイド文字ストリング長の計算』
- 515 ページの『wctomb() ― ワイド文字からマルチバイト文字への変換』
- 18 ページの『<stdlib.h>』

mbrlen() ― マルチバイト文字の長さの計算 (再始動可能)

フォーマット

```
#include <wchar.h>
size_t mbrlen (const char *s, size_t n, mbstate_t *ps);
```

言語レベル: ANSI

スレッド・セーフ: はい (ps が NULL 以外の場合)。

ロケール依存: この関数の振る舞いは、現行ロケールの LC_CTYPE カテゴリーの影響を受ける可能性があります。この関数は、LOCALETYPE(*LOCALEUCS2) または LOCALETYPE(*LOCALEUTF) をコンパイル・コマンドで指定した場合も、現行ロケールの LC_UNI_CTYPE カテゴリーによって影響を受ける可能性があります。この関数は、コンパイル・コマンドに対して LOCALETYPE(*CLD) が指定されている場合には使用できません。詳細については、 549 ページの『CCSID およびロケールの理解』を参照してください。

説明

この関数は、`mblen()` の再始動可能バージョンです。

`mbrlen()` 関数は、マルチバイト文字の長さを判別します。

n は、検証するマルチバイト・ストリングのバイト数 (最大) です。

この関数は、対応する内部状態のマルチバイト文字関数とは以下の点で異なります。この関数には追加のパラメーターとして mbstate_t の型を指す *ps* ポインターがあり、このポインターは、関連するマルチバイト文字シーケンスの現在の変換状態を完全に表すことができるオブジェクトを指します。 *ps* が NULL ポインターである場合、`mbrlen()` は `mblen()` と同じように振る舞います。

`mbrlen()` は、`mblen()` の再始動可能バージョンです。つまり、シフト状態情報は引数の 1 つとして渡され (*ps* は初期シフトを表します)、終了時に更新されます。 `mbrlen()` を使用すると、シフト状態情報を保持している場合に、あるマルチバイト・ストリングから別のマルチバイト・ストリングに切り替えることができます。

戻り値

s が NULL ポインターであり、アクティブなロケールで混合バイトのストリングが許可されている場合は、`mbrlen()` 関数は非ゼロを戻します。 *s* が NULL ポインターの場合であり、アクティブなロケールで混合バイトのストリングが許可されていない場合は、ゼロを戻します。

s が NULL ポインターでない場合、`mbrlen()` 関数は、以下のいずれかを戻します。

0
: *s* が NULL ストリングの場合、*s* はヌル文字を指します。

正
: 続く *n* 以下のバイトで有効なマルチバイト文字が構成される場合。戻り値は、マルチバイト文字を構成するバイト数です。

(size_t)-1

s が有効なマルチバイト文字を指さない場合。

(size_t)-2

続く *n* 以下のバイト数によって、バイトが不完全な (ただし有効である可能性もある) マルチバイト文字を形成し、すべての *n* バイトが処理された場合。

mbrlen() の使用例

```
/* This program is compiled with LOCALETYPE(*LOCALE) and   */
/* SYSIFCOPT(*IFSIO)                                       */

#include <stdio.h>
#include <stdlib.h>
#include <locale.h>
#include <wchar.h>
#include <errno.h>

#define  LOCNAME     "qsys.lib/JA_JP.locale"
#define  LOCNAME_EN  "qsys.lib/EN_US.locale"

int main(void)
{
    int length, sl = 0;
    char  string[10];
    mbstate_t ps = 0;
    memset(string, '\0', 10);
    string[0] = 0xC1;
    string[1] = 0x0E;
    string[2] = 0x41;
    string[3] = 0x71;
    string[4] = 0x41;
    string[5] = 0x72;
    string[6] = 0x0F;
    string[7] = 0xC2;
    /* In this first example we will find the length of       */
    /* of a multibyte character when the CCSID of locale      */
    /* associated with LC_CTYPE is 37.                        */
    /* For single byte cases the state will always            */
    /* remain in the initial state  0                         */

    if (setlocale(LC_ALL, LOCNAME_EN) == NULL)
        printf("setlocale failed.\n");

    length = mbrlen(string, MB_CUR_MAX, &ps);

    /* In this case length is 1, which is always the case for */
    /* single byte CCSID  */

    printf("length = %d, state = %d\n\n", length, ps);
    printf("MB_CUR_MAX: %d\n\n", MB_CUR_MAX);

    /* Now let's try a multibyte example.  We first must set the */
    /* locale to a multibyte locale.  We choose a locale with     */
    /* CCSID 5026  */

    if (setlocale(LC_ALL, LOCNAME) == NULL)
        printf("setlocale failed.\n");

    length = mbrlen(string, MB_CUR_MAX, &ps);

    /* The first is single byte so length is 1 and             */
    /* the state is still the initial state 0                  */

    printf("length = %d, state = %d\n\n", length, ps);
    printf("MB_CUR_MAX: %d\n\n", MB_CUR_MAX);
```

```
    sl += length;

    length = mbrlen(&string[sl], MB_CUR_MAX, &ps);

    /* The next character is a mixed byte.  Length is 3 to     */
    /* account for the shiftout 0x0e.  State is                */
    /* changed to double byte state.                           */

    printf("length = %d, state = %d\n\n", length, ps);

    sl += length;


   length = mbrlen(&string[sl], MB_CUR_MAX, &ps);

   /* The next character is also a double byte character.     */
   /* The state is changed to initial state since this was    */
   /* the last double byte character.  Length is 3 to         */
   /* account for the ending 0x0f shiftin.                    */

   printf("length = %d, state = %d\n\n", length, ps);

   sl += length;

   length = mbrlen(&string[sl], MB_CUR_MAX, &ps);

   /* The next character is single byte so length is 1 and    */
   /* state remains in initial state.                         */

   printf("length = %d, state = %d\n\n", length, ps);


}
/*  The output should look like this:

length = 1, state = 0

MB_CUR_MAX: 1

length = 1, state = 0

MB_CUR_MAX: 4

length = 3, state = 2

length = 3, state = 0

length = 1, state = 0
                                   */
* * * End of File * * *
```

関連情報

- 205 ページの『mblen() ― マルチバイト文字の長さの計算』
- 220 ページの『mbtowc() ― マルチバイト文字からワイド文字への変換』
- 210 ページの『mbrtowc() ― マルチバイト文字からワイド文字への変換 (再始動可能)』
- 214 ページの『mbsrtowcs() ― マルチバイト・ストリングからワイド文字ストリングへの変換 (再始動可能)』
- 354 ページの『setlocale() ― ロケールの設定』
- 467 ページの『wcrtomb() ― ワイド文字からマルチバイト文字への変換 (再開可能)』

- 494 ページの『wcsrtombs() ― ワイド文字ストリングからマルチバイト・ストリングへの変換 (再開可能)』
- 8 ページの『<locale.h>』
- 20 ページの『<wchar.h>』

mbrtowc() ― マルチバイト文字からワイド文字への変換 (再始動可能)

フォーマット

```
#include <wchar.h>
size_t mbrtowc (wchar_t *pwc, const char *s, size_t n, mbstate_t *ps);
```

言語レベル: ANSI

スレッド・セーフ: はい (ps が NULL 以外の場合)。

ロケール依存: この関数の振る舞いは、現行ロケールの LC_CTYPE カテゴリーの影響を受ける可能性があります。この関数は、LOCALETYPE(*LOCALEUCS2) または LOCALETYPE(*LOCALEUTF) をコンパイル・コマンドで指定した場合も、現行ロケールの LC_UNI_CTYPE カテゴリーによって影響を受ける可能性があります。この関数は、コンパイル・コマンドに対して LOCALETYPE(*CLD) が指定されている場合には使用できません。詳細については、 549 ページの『CCSID およびロケールの理解』を参照してください。

ワイド文字関数: 詳細については、 553 ページの『ワイド文字』 を参照してください。

説明

この関数は、mbtowc() 関数の再始動可能バージョンです。

s が NULL ポインターの場合には、mbrtowc() 関数は初期シフト状態 (エンコードが状態依存でないか、または初期変換状態が記述されている場合はゼロ) を入力するのに必要なバイト数を判別します。この状態において、*pwc* パラメーターの値は無視され、結果として、記述されているシフト状態が初期変換状態となります。

s が NULL ポインターではない場合は、mbrtowc() 関数は *s* によってポイントされているマルチバイト文字 (および先行するシフト・シーケンス) 内のバイト数を判別して対応するマルチバイト文字の値を生成し、 *pwc* が NULL ポインターではない場合は、その値を *pwc* によってポイントされているオブジェクト内に保管します。対応するマルチバイト文字が NULL ワイド文字の場合には、結果の状態は初期変換状態にリセットされます。

この関数は、対応する内部状態のマルチバイト文字関数とは以下の点で異なります。この関数には追加のパラメーターとして *mbstate_t* の型を指す *ps* ポインターがあり、このポインターは、関連するマルチバイト文字シーケンスの現在の変換状態を完全に表すことができるオブジェクトを指します。 *ps* が NULL の場合、この関数は内部の静的変数を状態に使用します。

最大で、*n* バイトのマルチバイト・ストリングが検証されます。

戻り値

s が NULL ポインターの場合、mbrtowc() 関数は、初期シフト状態を入力するのに必要なバイト数を戻します。戻り値は MB_CUR_MAX マクロ未満でなければなりません。

変換エラーが発生した場合、errno は **ECONVERT** に設定される可能性があります。

s が NULL ポインターでない場合、`mbrtowc()` 関数は、以下のいずれかを戻します。

0 続く *n* 以下のバイト数によって、NULL ワイド文字に対応するマルチバイト文字が形成される場合。

正 続く *n* 以下のバイト数によって、有効なマルチバイト文字が形成される場合。戻り値は、マルチバイト文字を構成するバイト数です。

(size_t)-2
次の *n* バイトが不完全な (ただし有効である可能性もある) マルチバイト文字を形成し、すべての *n* バイトが処理された場合。*n* の値が MB_CUR_MAX マクロの値よりも小さい場合、これが起こるかどうかは指定されていません。

(size_t)-1
エンコード・エラーが発生した場合 (次の *n* または数バイトが完全で正確なマルチバイト文字を形成しないとき)。 EILSEQ マクロの値は errno に保管されるが、変換状態は未変更のままになります。

注: -2 の値が戻される場合、そのストリングには冗長シフトアウトおよびシフトイン文字、または一部の UTF-8 文字が含まれています。マルチバイト・ストリングの処理を継続するには、*n* の値でポインターを増分し、`mbrtowc()` をもう一度呼び出します。

`mbrtowc()` の使用例

```
/* This program is compiled with LOCALETYPE(*LOCALE) and   */
/* SYSIFCOPT(*IFSIO)                                        */

#include  <stdio.h>
#include  <stdlib.h>
#include  <locale.h>
#include  <wchar.h>
#include  <errno.h>

#define  LOCNAME     "/qsys.lib/JA_JP.locale"
#define  LOCNAME_EN  "/qsys.lib/EN_US.locale"

int main(void)
{
    int length, sl = 0;
    char  string[10];
    wchar_t buffer[10];
    mbstate_t ps = 0;
    memset(string, '\0', 10);
    string[0] = 0xC1;
    string[1] = 0x0E;
    string[2] = 0x41;
    string[3] = 0x71;
    string[4] = 0x41;
    string[5] = 0x72;
    string[6] = 0x0F;
    string[7] = 0xC2;
    /* In this first example we will convert                */
    /* a multibyte character when the CCSID of locale       */
    /* associated with LC_CTYPE is 37.                      */
    /* For single byte cases the state will always          */
    /* remain in the initial state  0                       */

    if (setlocale(LC_ALL, LOCNAME_EN) == NULL)
        printf("setlocale failed.\n");

    length = mbrtowc(buffer, string, MB_CUR_MAX, &ps);

    /* In this case length is 1, and C1 is converted 0x00C1   */
```

```
    printf("length = %d, state = %d\n\n", length, ps);
    printf("MB_CUR_MAX: %d\n\n", MB_CUR_MAX);

    /* Now lets try a multibyte example.  We first must set the */
    /* locale to a multibyte locale.  We choose a locale with     */
    /* CCSID 5026  */

    if (setlocale(LC_ALL, LOCNAME) == NULL)
        printf("setlocale failed.\n");

    length = mbrtowc(buffer, string, MB_CUR_MAX, &ps);

    /* The first is single byte so length is 1 and             */
    /* the state is still the initial state 0.  C1 is converted*/
    /* to 0x00C1     */

   printf("length = %d, state = %d\n\n", length, ps);
   printf("MB_CUR_MAX: %d\n\n", MB_CUR_MAX);

    sl += length;

    length = mbrtowc(&buffer[1], &string[sl], MB_CUR_MAX, &ps);

    /* The next character is a mixed byte.  Length is 3 to     */
    /* account for the shiftout 0x0e.  State is                */
    /* changed to double byte state.  0x4171 is copied into    */
    /* the buffer   */

   printf("length = %d, state = %d\n\n", length, ps);


    sl += length;

    length = mbrtowc(&buffer[2], &string[sl], MB_CUR_MAX, &ps);

    /* The next character is also a double byte character.     */
    /* The state is changed to initial state since this was   */
    /* the last double byte character.  Length is 3 to         */
    /* account for the ending 0x0f shiftin.  0x4172 is copied  */
    /* into the buffer.  */

    printf("length = %d, state = %d\n\n", length, ps);

    sl += length;

    length = mbrtowc(&buffer[3], &string[sl], MB_CUR_MAX, &ps);

    /* The next character is single byte so length is 1 and    */
    /* state remains in initial state.  0xC2 is converted to   */
    /* 0x00C2.   The buffer now has the value:                 */
    /* 0x00C14171417200C2                                      */

    printf("length = %d, state = %d\n\n", length, ps);

}
/*  The output should look like this:

length = 1, state = 0

MB_CUR_MAX: 1

length = 1, state = 0

MB_CUR_MAX: 4

length = 3, state = 2
```

```
length = 3, state = 0

length = 1, state = 0
                                    */
```

関連情報

- 205 ページの『mblen() ― マルチバイト文字の長さの計算』
- 207 ページの『mbrlen() ― マルチバイト文字の長さの計算 (再始動可能)』
- 214 ページの『mbsrtowcs() ― マルチバイト・ストリングからワイド文字ストリングへの変換 (再始動可能)』
- 354 ページの『setlocale() ― ロケールの設定』
- 467 ページの『wcrtomb() ― ワイド文字からマルチバイト文字への変換 (再開可能)』
- 494 ページの『wcsrtombs() ― ワイド文字ストリングからマルチバイト・ストリングへの変換 (再開可能)』
- 8 ページの『<locale.h>』
- 20 ページの『<wchar.h>』

mbsinit() ― 状態オブジェクトが初期状態であるかどうかのテスト

フォーマット

```
#include <wchar.h>
int mbsinit (const mbstate_t *ps);
```

言語レベル: ANSI

スレッド・セーフ: はい

ロケール依存: この関数の振る舞いは、現行ロケールの LC_CTYPE カテゴリーの影響を受ける可能性があります。この関数は、コンパイル・コマンドに対して LOCALETYPE(*CLD) が指定されている場合には使用できません。詳細については、 549 ページの『CCSID およびロケールの理解』を参照してください。

説明

ps が NULL ポインターでない場合、`mbsinit()` 関数は、*mbstate_t* オブジェクトを指すポインターが初期変換状態を記述しているかどうかを判別します。

戻り値

ps が NULL ポインターである場合、あるいはポインターで指されたオブジェクトが初期変換状態を記述する場合、`mbsinit()` は、ゼロ以外の値を戻します。それ以外の場合は、ゼロを戻します。

`mbsinit()` の使用例

この例では、変換状態が初期状態であるかどうかを表示して確認します。

```
#include <stdio.h>
#include <wchar.h>
#include <stdlib.h>

main()
{
   char     *string = "ABC";
   mbstate_t state = 0;
   wchar_t   wc;
   int   rc;

   rc = mbrtowc(&wc, string, MB_CUR_MAX, &state);
   if (mbsinit(&state))
     printf("In initial conversion state¥n");
}
```

関連情報

- 207 ページの『mbrlen() ― マルチバイト文字の長さの計算 (再始動可能)』
- 210 ページの『mbrtowc() ― マルチバイト文字からワイド文字への変換 (再始動可能)』
- 『mbsrtowcs() ― マルチバイト・ストリングからワイド文字ストリングへの変換 (再始動可能)』
- 354 ページの『setlocale() ― ロケールの設定』
- 467 ページの『wcrtomb() ― ワイド文字からマルチバイト文字への変換 (再開可能)』
- 494 ページの『wcsrtombs() ― ワイド文字ストリングからマルチバイト・ストリングへの変換 (再開可能)』
- 8 ページの『<locale.h>』
- 20 ページの『<wchar.h>』

mbsrtowcs() ― マルチバイト・ストリングからワイド文字ストリングへの変換 (再始動可能)

フォーマット

```
#include <wchar.h>
size_t mbsrtowcs (wchar_t *dst, const char **src, size_t len,
                  mbstate_t *ps);
```

言語レベル: ANSI

スレッド・セーフ: はい (ps が NULL 以外の場合)。

ロケール依存: この関数の振る舞いは、現行ロケールの LC_CTYPE カテゴリーの影響を受ける可能性があります。この関数は、LOCALETYPE(*LOCALEUCS2) または LOCALETYPE(*LOCALEUTF) をコンパイル・コマンドで指定した場合も、現行ロケールの LC_UNI_CTYPE カテゴリーによって影響を受ける可能性があります。この関数は、コンパイル・コマンドに対して LOCALETYPE(*CLD) が指定されている場合には使用できません。詳細については、 549 ページの『CCSID およびロケールの理解』を参照してください。

ワイド文字関数: 詳細については、 553 ページの『ワイド文字』 を参照してください。

説明

この関数は、`mbstowcs()` の再始動可能バージョンです。

mbsrtowcs() 関数は、*src* が間接的に指す配列から、*ps* が記述する変換状態で始まるマルチバイト文字のシーケンスを、対応するワイド文字に変換します。次に、*dst* で示される配列に、変換された文字を保管します。

変換は終了のヌル文字 (これを含む) まで続行し、このヌル文字も保管されます。変換は次の 2 つの場合、早期に停止します。すなわち、有効なマルチバイト文字を形成しないバイトのシーケンスに達したとき、または (*dst* が NULL ポインターでない場合に) *len* ワイド文字が、*dst* が指す配列に保管されたときです。それぞれの変換は、mbrtowc() 関数への呼び出しの場合と同様に行われます。

dst が NULL ポインターでない場合、*src* が指すポインター・オブジェクトには、NULL ポインターが割り当てられる (終了のヌル文字に達したために変換が停止した場合) か、変換された最後のマルチバイト文字の直後のアドレスが割り当てられます。終了のヌル文字に達したことにより変換が停止した場合は、初期変換状態が記述されます。

戻り値

入力ストリングが有効なマルチバイト文字で開始していない場合は、エンコード・エラーが発生し、mbsrtowcs() 関数がマクロ EILSEQ の値を errno に保管し、(size_t) -1 を戻しますが、変換状態は変更されません。その他の場合は、正常に変換されたマルチバイト文字の数を戻しますが、これは、*dst* が NULL ポインターでないときに変更された配列エレメント数と同じです。

変換エラーが発生した場合、errno は **ECONVERT** に設定される可能性があります。

mbsrtowcs() の使用例

```
#include <stdio.h>
#include <stdlib.h>
#include <wchar.h>
#include <locale.h>

#define SIZE 10

int main(void)
{
   char          mbs1[] = "abc";
   char          mbs2[] = "¥x81¥x41" "m" "¥x81¥x42";
   const char   *pmbs1 = mbs1;
   const char   *pmbs2 = mbs2;
   mbstate_t   ss1 = 0;
   mbstate_t   ss2 = 0;
   wchar_t     wcs1[SIZE], wcs2[SIZE];

   if (NULL == setlocale(LC_ALL, "/qsys.lib/locale.lib/ja_jp939.locale"))
 {
      printf("setlocale failed.¥n");
      exit(EXIT_FAILURE);
   }
   mbsrtowcs(wcs1, &pmbs1, SIZE, &ss1);
   mbsrtowcs(wcs2, &pmbs2, SIZE, &ss2);
   printf("The first wide character string is %ls.¥n", wcs1);
   printf("The second wide character string is %ls.¥n", wcs2);
   return 0;
}

   /*******************************************************
      The output should be similar to:

      The first wide character string is abc.
      The second wide character string is Am B.
    *******************************************************/
```

210 ページの『mbrtowc() ― マルチバイト文字からワイド文字への変換 (再始動可能)』の例も参照してください。

関連情報

- 205 ページの『mblen() ― マルチバイト文字の長さの計算』
- 207 ページの『mbrlen() ― マルチバイト文字の長さの計算 (再始動可能)』
- 210 ページの『mbrtowc() ― マルチバイト文字からワイド文字への変換 (再始動可能)』
- 『mbstowcs() ― マルチバイト・ストリングからワイド文字ストリングへの変換』
- 354 ページの『setlocale() ― ロケールの設定』
- 467 ページの『wcrtomb() ― ワイド文字からマルチバイト文字への変換 (再開可能)』
- 494 ページの『wcsrtombs() ― ワイド文字ストリングからマルチバイト・ストリングへの変換 (再開可能)』
- 8 ページの『<locale.h>』
- 20 ページの『<wchar.h>』

mbstowcs() ― マルチバイト・ストリングからワイド文字ストリングへの変換

フォーマット

```
#include <stdlib.h>
size_t mbstowcs(wchar_t *pwc, const char *string, size_t n);
```

言語レベル: ANSI

スレッド・セーフ: はい。

ロケール依存: この関数の振る舞いは、現行ロケールの LC_CTYPE カテゴリーの影響を受ける可能性があります。この関数は、LOCALETYPE(*LOCALEUCS2) または LOCALETYPE(*LOCALEUTF) をコンパイル・コマンドで指定した場合も、現行ロケールの LC_UNI_CTYPE カテゴリーによって影響を受ける可能性があります。詳細については、 549 ページの『CCSID およびロケールの理解』を参照してください。

ワイド文字関数: 詳細については、 553 ページの『ワイド文字』 を参照してください。

説明

mbstowcs() 関数は、*string* が示すマルチバイト文字のシーケンスの長さを判別します。次に、初期シフト状態で開始するマルチバイト文字ストリングをワイド文字ストリングに変換し、そのワイド文字を *pwc* が指すバッファーに保管します。最大で *n* 個のワイド文字が書き込まれます。

戻り値

mbstowcs() 関数は、生成されたワイド文字の数を戻します (終了 NULL ワイド文字を含まず)。無効なマルチバイト文字に遭遇した場合、関数は (size_t)-1 を戻します。

変換エラーが発生した場合、errno は **ECONVERT** に設定される可能性があります。

mbstowcs() の使用例

```
/* This program is compiled with LOCALETYPE(*LOCALEUCS2) and   */
/* SYSIFCOPT(*IFSIO)                                            */

#include   <stdio.h>
#include   <stdlib.h>
#include   <locale.h>
#include   <wchar.h>
#include   <errno.h>

#define  LOCNAME     "qsys.lib/JA_JP.locale"
#define  LOCNAME_EN  "qsys.lib/EN_US.locale"

int main(void)
{
    int length, sl = 0;
    char  string[10];
    char  string2[] = "ABC";
    wchar_t buffer[10];
    memset(string, '\0', 10);
    string[0] = 0xC1;
    string[1] = 0x0E;
    string[2] = 0x41;
    string[3] = 0x71;
    string[4] = 0x41;
    string[5] = 0x72;
    string[6] = 0x0F;
    string[7] = 0xC2;
    /* In this first example we will convert                 */
    /* a multibyte character when the CCSID of locale        */
    /* associated with LC_CTYPE is 37.                       */

    if (setlocale(LC_ALL, LOCNAME_EN) == NULL)
        printf("setlocale failed.\n");


    length = mbstowcs(buffer, string2, 10);

    /* In this case length ABC is converted to UNICODE ABC    */
    /* or 0x004100420043.  Length will be 3.                  */

    printf("length = %d\n\n", length);

    /* Now lets try a multibyte example.  We first must set the */
    /* locale to a multibyte locale.  We choose a locale with     */
    /* CCSID 5026  */

    if (setlocale(LC_ALL, LOCNAME) == NULL)
        printf("setlocale failed.\n");

    length = mbstowcs(buffer, string, 10);

    /* The buffer now has the value:                          */
    /* 0x004103A103A30042        length is 4                  */

    printf("length = %d\n\n", length);

}
/*  The output should look like this:

length = 3

length = 4
                                   */
```

```
/* This program is compiled with LOCALETYPE(*LOCALE) and       */
/* SYSIFCOPT(*IFSIO)                                           */

#include <stdio.h>
#include <stdlib.h>
#include <locale.h>
#include <wchar.h>
#include <errno.h>

#define  LOCNAME     "qsys.lib/JA_JP.locale"
#define  LOCNAME_EN  "qsys.lib/EN_US.locale"

int main(void)
{
    int length, sl = 0;
    char  string[10];
    char  string2[] = "ABC";
    wchar_t buffer[10];
    memset(string, '\0', 10);
    string[0] = 0xC1;
    string[1] = 0x0E;
    string[2] = 0x41;
    string[3] = 0x71;
    string[4] = 0x41;
    string[5] = 0x72;
    string[6] = 0x0F;
    string[7] = 0xC2;
    /* In this first example we will convert                  */
    /* a multibyte character when the CCSID of locale         */
    /* associated with LC_CTYPE is 37.                        */

    if (setlocale(LC_ALL, LOCNAME_EN) == NULL)
        printf("setlocale failed.\n");

    length = mbstowcs(buffer, string2, 10);

    /* In this case length ABC is converted to             */
    /* 0x00C100C200C3.  Length will be 3.                  */

    printf("length = %d\n\n", length);

    /* Now lets try a multibyte example.  We first must set the *
    /* locale to a multibyte locale.  We choose a locale with
    /* CCSID 5026  */

    if (setlocale(LC_ALL, LOCNAME) == NULL)
        printf("setlocale failed.\n");

    length = mbstowcs(buffer, string, 10);

    /* The buffer now has the value:                           */
    /* 0x00C14171417200C2        length is 4                   */

    printf("length = %d\n\n", length);

}

/*  The output should look like this:

length = 3

length = 4
                                   */
```

関連情報

- 205 ページの『mblen() ― マルチバイト文字の長さの計算』

- 『mbtowc() — マルチバイト文字からワイド文字への変換』
- 354 ページの『setlocale() — ロケールの設定』
- 482 ページの『wcslen() — ワイド文字ストリング長の計算』
- 505 ページの『wcstombs() — ワイド文字ストリングからマルチバイト・ストリングへの変換』
- 8 ページの『<locale.h>』
- 18 ページの『<stdlib.h>』
- 20 ページの『<wchar.h>』

mbtowc() — マルチバイト文字からワイド文字への変換

フォーマット

```
#include <stdlib.h>
int mbtowc(wchar_t *pwc, const char *string, size_t n);
```

言語レベル: ANSI

スレッド・セーフ: いいえ。代わりに mbrtowc() を使用します。

ロケール依存: この関数の振る舞いは、現行ロケールの LC_CTYPE カテゴリーの影響を受ける可能性があります。この関数は、LOCALETYPE(*LOCALEUCS2) または LOCALETYPE(*LOCALEUTF) をコンパイル・コマンドで指定した場合も、現行ロケールの LC_UNI_CTYPE カテゴリーによって影響を受ける可能性があります。詳細については、 549 ページの『CCSID およびロケールの理解』を参照してください。

ワイド文字関数: 詳細については、 553 ページの『ワイド文字』 を参照してください。

説明

mbtowc() 関数はまず、*string* が示すマルチバイト文字の長さを判別します。次に、そのマルチバイト文字を mbstowcs で記述されたワイド文字に変換します。最大で *n* バイトが検査されます。

戻り値

ストリングが NULL の場合、mbtowc() 関数は以下を戻します。
- アクティブなロケールが混合バイトの場合は、ゼロ以外。関数は、状態変数を初期化します。
- それ以外の場合は、0。

string が NULL 以外の場合、mbtowc() 関数は以下を戻します。
- ストリングがヌル文字を指す場合は、0。
- 変換されたマルチバイト文字を構成するバイト数。
- *string* が有効なマルチバイト文字を指さない場合は、-1。

変換エラーが発生した場合、errno は **ECONVERT** に設定される可能性があります。

mbtowc() の使用例

この例では、mblen() および mbtowc() 関数を使用して、マルチバイト文字を単一のワイド文字に変換します。

```
#include <stdio.h>
#include <stdlib.h>

#define LOCNAME "qsys.lib/mylib.lib/ja_jp959.locale"
/*Locale created from source JA_JP and CCSID 939 */

int length, temp;
char string [] = "¥x0e¥x41¥x71¥x0f";
wchar_t arr[6];

int main(void)
{
   /* initialize internal state variable */
   temp = mbtowc(arr, NULL, 0);

   setlocale (LC_ALL, LOCNAME);
   /* Set string to point to a multibyte character. */
   length = mblen(string, MB_CUR_MAX);
   temp = mbtowc(arr,string,length);
   arr[1] = L'¥0';
   printf("wide character string: %ls",arr);
}
```

関連情報

- 205 ページの『mblen() ― マルチバイト文字の長さの計算』
- 216 ページの『mbstowcs() ― マルチバイト・ストリングからワイド文字ストリングへの変換』
- 482 ページの『wcslen() ― ワイド文字ストリング長の計算』
- 515 ページの『wctomb() ― ワイド文字からマルチバイト文字への変換』
- 18 ページの『<stdlib.h>』

memchr() ― バッファーの検索

フォーマット

```
#include <string.h>
void *memchr(const void *buf, int c, size_t count);
```

言語レベル: ANSI

スレッド・セーフ: はい。

説明

memchr() 関数は、*buf* の最初の *count* バイトを検索して、符号なし文字に変換された *c* が最初に現れる位置を調べます。検索は、この関数が *c* を検出するか、*count* バイトを検査するまで続きます。

戻り値

memchr() 関数は、*buf* の *c* のロケーションへのポインターを戻します。 *c* が *buf* の先頭の *count* バイトの中にない場合は、NULL を戻します。

memchr() の使用例

この例では、指定したストリング内で、最初のオカレンス “x” を見つけます。見つかった場合は、その文字で始まるストリングが出力されます。

```
#include <stdio.h>
#include <string.h>

int main(int argc, char ** argv)
{
  char * result;

  if ( argc != 2 )
    printf( "Usage: %s string¥n", argv[0] );
  else
  {
    if ((result = (char *) memchr( argv[1], 'x', strlen(argv[1])) ) != NULL)
      printf( "The string starting with x is %s¥n", result );
    else
      printf( "The letter x cannot be found in the string¥n" );
  }
}

/********************  Output should be similar to:  **************

The string starting with x is xing
*/
```

関連情報

- 『memcmp() ― バッファーの比較』
- 223 ページの『memcpy() ― バイトのコピー』
- 226 ページの『memmove() ― バイトのコピー』
- 521 ページの『wmemchr() ―ワイド文字バッファーでのワイド文字の位置検出』
- 227 ページの『memset() ― 値へのバイトの設定』
- 375 ページの『strchr() ― 文字の検索』
- 19 ページの『<string.h>』

---

memcmp() ― バッファーの比較

フォーマット

```
#include <string.h>
int memcmp(const void *buf1, const void *buf2, size_t count);
```

言語レベル: ANSI

スレッド・セーフ: はい。

説明

memcmp() 関数は、*buf1* と *buf2* の先頭 *count* バイトを比較します。

戻り値

memcmp() 関数は、バッファー間の関係を示す次のような値を戻します。

値	意味
0 より小さい値	*buf1* は *buf2* より小さい
0	*buf1* は *buf2* と等しい
0 より大きい値	*buf1* は *buf2* より大きい

memcmp() の使用例

この例では、main() に渡される 1 番目と 2 番目の引数を比較して、どちらが大きいかを判別します (どちらかが大きい場合)。

```
#include <stdio.h>
#include <string.h>

int main(int argc, char ** argv)
{
  int  len;
  int  result;

  if ( argc != 3 )
  {
     printf( "Usage: %s string1 string2¥n", argv[0] );
  }
  else
  {
     /* Determine the length to be used for comparison */
     if (strlen( argv[1] ) < strlen( argv[2] ))
       len = strlen( argv[1] );
     else
       len = strlen( argv[2] );

     result = memcmp( argv[1], argv[2], len );

     printf( "When the first %i characters are compared,¥n", len );
     if ( result == 0 )
       printf( "¥"%s¥" is identical to ¥"%s¥"¥n", argv[1], argv[2] );
     else if ( result < 0 )
       printf( "¥"%s¥" is less than ¥"%s¥"¥n", argv[1], argv[2] );
     else
       printf( "¥"%s¥" is greater than ¥"%s¥"¥n", argv[1], argv[2] );
   }
}

/****************  If the program is passed the arguments  **************
*****************       firststring and secondstring,     ************
*****************       output should be:                 ************

When the first 11 characters are compared,
"firststring" is less than "secondstring"
**********************************************************************/
```

関連情報

- 221 ページの『memchr() ― バッファーの検索』
- 『memcpy() ― バイトのコピー』
- 522 ページの『wmemcmp() ―ワイド文字バッファーの比較』
- 226 ページの『memmove() ― バイトのコピー』
- 227 ページの『memset() ― 値へのバイトの設定』
- 376 ページの『strcmp() ― ストリングの比較』
- 19 ページの『<string.h>』

memcpy() ― バイトのコピー

フォーマット

```
#include <string.h>
void *memcpy(void *dest, const void *src, size_t count);
```

言語レベル: ANSI

スレッド・セーフ: はい。

説明

memcpy() 関数は、*count* バイトの *src* を *dest* にコピーします。オーバーラップしたオブジェクト間でコピーが行われる場合には、振る舞いは予期できません。オーバーラップしたオブジェクト間でのコピーは、memmove() 関数で可能です。

戻り値

memcpy() 関数は、*dest* へのポインターを戻します。

memcpy() の使用例

この例では、source のコンテンツを target へコピーします。

```
#include <string.h>
#include <stdio.h>

#define MAX_LEN 80

char source[ MAX_LEN ] = "This is the source string";
char target[ MAX_LEN ] = "This is the target string";

int main(void)
{
  printf( "Before memcpy, target is ¥"%s¥"¥n", target );
  memcpy( target, source, sizeof(source));
  printf( "After memcpy, target becomes ¥"%s¥"¥n", target );
}

/*********************  Expected output:  ************************

Before memcpy, target is "This is the target string"
After memcpy, target becomes "This is the source string"
*/
```

関連情報

- 221 ページの『memchr() ― バッファーの検索』
- 222 ページの『memcmp() ― バッファーの比較』
- 524 ページの『wmemcpy() ―ワイド文字バッファーのコピー』
- 226 ページの『memmove() ― バイトのコピー』
- 227 ページの『memset() ― 値へのバイトの設定』
- 380 ページの『strcpy() ― ストリングのコピー』
- 19 ページの『<string.h>』

memicmp() - バイトの比較

フォーマット

```
#include <string.h>    // also in <memory.h>
int memicmp(void *buf1, void *buf2, unsigned int cnt);
```

注: memicmp 関数は、C++ プログラムで使用可能です。 __cplusplus__strings__ マクロがプログラムで定義されている場合にのみ、C でも使用できます。

言語レベル: Extension

スレッド・セーフ: はい。

ロケール依存: この関数の振る舞いは、現行ロケールの LC_CTYPE カテゴリーの影響を受ける可能性があります。詳細については、 549 ページの『CCSID およびロケールの理解』を参照してください。

説明

memicmp 関数は、 *buf1* および *buf2* の先頭 *cnt* バイトを比較しますが、2 つのバッファー内の文字が大文字であるか小文字であるかは考慮に入れません。関数は、すべての大文字を小文字に変換してから比較を行います。

戻り値

memicmp の戻り値は、以下の結果を示します。

値	意味
0 より小さい値	*buf1* は *buf2* より小さい
0	*buf1* は *buf2* と等しい
0 より大きい値	*buf1* は *buf2* より大きい

memicmp() の使用例

この例では、それぞれが 29 文字のサブストリングを含む 2 つのストリング (大/小文字以外は同じ) をコピーします。次にこの例では、大/小文字に関係なく先頭 29 バイトを比較します。

```
#include <stdio.h>
#include <string.h>
char first[100],second[100];
int main(void)
{
    int result;
    strcpy(first, "Those Who Will Not Learn From History");
    strcpy(second, "THOSE WHO WILL NOT LEARN FROM their mistakes");
    printf("Comparing the first 29 characters of two strings.¥n");
    result = memicmp(first, second, 29);
    printf("The first 29 characters of String 1 are ");
    if (result < 0)
       printf("less than String 2.\n");
    else
       if (0 == result)
          printf("equal to String 2.\n");
       else
          printf("greater than String 2.\n");
    return 0;
}
```

The output should be:

```
Comparing the first 29 characters of two strings.
The first 29 characters of String 1 are equal to String 2
```

関連情報

- 221 ページの『memchr() ― バッファーの検索』
- 222 ページの『memcmp() ― バッファーの比較』
- 223 ページの『memcpy() ― バイトのコピー』
- 『memmove() ― バイトのコピー』
- 227 ページの『memset() ― 値へのバイトの設定』
- 376 ページの『strcmp() ― ストリングの比較』
- 378 ページの『strcmpi() - 大/小文字を区別しないストリングの比較』
- 390 ページの『stricmp() - 大/小文字を区別しないストリングの比較』
- 398 ページの『strnicmp - 大/小文字の区別をしないサブストリングの比較』
- 19 ページの『<string.h>』

memmove() ― バイトのコピー

フォーマット

```
#include <string.h>
void *memmove(void *dest, const void *src, size_t count);
```

言語レベル: ANSI

スレッド・セーフ: はい。

説明

memmove() 関数は、*count* バイトの *src* を *dest* にコピーします。この関数は、オーバーラップしたオブジェクト間で、最初に *src* が一時配列にコピーされた場合と同様にコピーすることができます。

戻り値

memmove() 関数は、*dest* へのポインターを戻します。

memmove() の使用例

この例では、単語「shiny」を位置 target + 2 から位置 target + 8 へコピーします。

```
#include <string.h>
#include <stdio.h>

#define SIZE    21

char target[SIZE] = "a shiny white sphere";

int main( void )
{
  char * p = target + 8;  /* p points at the starting character
                          of the word we want to replace */
  char * source = target + 2; /* start of "shiny" */

  printf( "Before memmove, target is ¥"%s¥"¥n", target );
  memmove( p, source, 5 );
  printf( "After memmove, target becomes ¥"%s¥"¥n", target );
}

/*********************  Expected output:  ************************

Before memmove, target is "a shiny white sphere"
After memmove, target becomes "a shiny shiny sphere"
*/
```

関連情報

- 221 ページの『memchr() — バッファーの検索』
- 222 ページの『memcmp() — バッファーの比較』
- 525 ページの『wmemmove() — ワイド文字バッファーのコピー』
- 223 ページの『memcpy() — バイトのコピー』
- 『memset() — 値へのバイトの設定』
- 380 ページの『strcpy() — ストリングのコピー』
- 19 ページの『<string.h>』

memset() — 値へのバイトの設定

フォーマット

```
#include <string.h>
void *memset(void *dest, int c, size_t count);
```

言語レベル: ANSI

スレッド・セーフ: はい。

説明

`memset()` 関数は、先頭 *count* バイトの *dest* を値 *c* に設定します。 *c* の値は、符号なし文字に変換されます。

戻り値

`memset()` 関数は、*dest* へのポインターを戻します。

`memset()` の使用例

この例では、10 バイトのバッファーを A に設定し、続く 10 バイトを B に設定します。

```
#include <string.h>
#include <stdio.h>

#define  BUF_SIZE  20

int main(void)
{
   char buffer[BUF_SIZE + 1];
   char *string;

   memset(buffer, 0, sizeof(buffer));
   string = (char *) memset(buffer,'A', 10);
   printf("¥nBuffer contents: %s¥n", string);
   memset(buffer+10, 'B', 10);
   printf("¥nBuffer contents: %s¥n", buffer);
}

/*******************  Output should be similar to:  ***************

Buffer contents: AAAAAAAAAA

Buffer contents: AAAAAAAAAABBBBBBBBBB
*/
```

関連情報

- 221 ページの『memchr() ― バッファーの検索』
- 222 ページの『memcmp() ― バッファーの比較』
- 223 ページの『memcpy() ― バイトのコピー』
- 226 ページの『memmove() ― バイトのコピー』
- 526 ページの『wmemset() ― 値に対するワイド文字バッファーの設定』
- 19 ページの『<string.h>』

mktime() ― 地方時の変換

フォーマット

```
#include <time.h>
time_t mktime(struct tm *time);
```

言語レベル: ANSI

スレッド・セーフ: はい。

ロケール依存: この関数の振る舞いは、現行ロケールの LC_TOD カテゴリーの影響を受ける可能性があります。

説明

`mktime()` 関数は、*time* が指す保管済みの *tm* 構造体 (ジョブ地方時として想定) を、別の time 関数の使用に適した time_t 構造体に変換します。変換後は、time_t 構造体は協定世界時 (UTC) であるとみなされます。この変換を行うため、`mktime()` はローカル時間帯および夏時間 (DST) の現行ロケール設定をチェックします。これらの値が現行ロケールで未設定の場合、`mktime()` はローカル時間帯および夏時間 (DST) の設定を現行ジョブから取得します。 DST がロケールで設定済みでも、時間帯情報が未設定の場合は、ロケール内の DST 情報は無視されます。 `mktime()` は次に現行の時間帯を使用して、UTC を判別します。

構造体のエレメントによっては、*time* が指す値が gmtime() 用に表示される範囲に制限されないものもあります。

mktime() に渡される tm_wday と tm_yday の値は無視され、戻りでこれらの正しい値が割り当てられます。

tm_isdst の値が正であるか 0 であるかにより、mktime() は、指定した時間で DST が有効になっているかいないかを最初に推定します。 *tm_isdst* が負の値の場合、 mktime() は、指定された時間で DST が有効になっているかどうかについて、判別を試みます。

戻り値

mktime() 関数は、型 time_t を持つ協定世界時 (UTC) を戻します。協定世界時を表示できない場合、値 (time_t)(-1) は戻ります。

mktime() の使用例

この例では、現在日付から 40 日と 16 時間後の曜日を出力します。

```
#include <stdio.h>
#include <time.h>

char *wday[] = { "Sunday", "Monday", "Tuesday", "Wednesday",
                 "Thursday", "Friday", "Saturday" };

int main(void)
{
  time_t t1, t3;
  struct tm *t2;

  t1 = time(NULL);
  t2 = localtime(&t1);
  t2 -> tm_mday += 40;
  t2 -> tm_hour += 16;
  t3 = mktime(t2);

  printf("40 days and 16 hours from now, it will be a %s ¥n",
          wday[t2 -> tm_wday]);
}

/*******************  Output should be similar to:  ***************

40 days and 16 hours from now, it will be a Sunday
*/
```

関連情報

- 41 ページの『asctime() ― 時間から文字ストリングへの変換』
- 43 ページの『asctime_r() ― 時間から文字ストリングへの変換 (再始動可能)』
- 75 ページの『ctime() ― 時間から文字ストリングへの変換』
- 77 ページの『ctime64() ― 時間から文字ストリングへの変換』
- 80 ページの『ctime64_r() ― 時間から文字ストリングへの変換 (再始動可能)』
- 78 ページの『ctime_r() ― 時間から文字ストリングへの変換 (再始動可能)』
- 167 ページの『gmtime() ― 時間の変換』
- 169 ページの『gmtime64() ― 時間の変換』
- 173 ページの『gmtime64_r() ― 時間の変換 (再始動可能)』

- 171 ページの『gmtime_r() ― 時間の変換 (再始動可能)』
- 192 ページの『localtime() ― 時間の変換』
- 194 ページの『localtime64() ― 時間の変換』
- 197 ページの『localtime64_r() ― 時間の変換 (再始動可能)』
- 195 ページの『localtime_r() ― 時間の変換 (再始動可能)』
- 『mktime64() ― 地方時の変換』
- 429 ページの『time() ― 現在時刻の判別』
- 430 ページの『time64() ― 現在時刻の判別』
- 19 ページの『<time.h>』

mktime64() ― 地方時の変換

フォーマット

```
#include <time.h>
time64_t mktime64(struct tm *time);
```

言語レベル: ILE C Extension

スレッド・セーフ: はい。

ロケール依存: この関数の振る舞いは、現行ロケールの LC_TOD カテゴリーの影響を受ける可能性があります。

説明

`mktime64()` 関数は、*time* が指す保管済みの *tm* 構造体 (ジョブ地方時として想定) を、別の time 関数の使用に適した `time64_t` 値に変換します。変換後は、`time64_t` 値は協定世界時 (UTC) であるとみなされます。この変換を行うため、`mktime64()` はローカル時間帯および夏時間 (DST) の現行ロケール設定をチェックします。これらの値が現行ロケールで未設定の場合、`mktime64()` はローカル時間帯および夏時間 (DST) の設定を現行ジョブから取得します。 DST がロケールで設定済みでも、時間帯情報が未設定の場合は、ロケール内の DST 情報は無視されます。 `mktime64()` 関数は次に、現行ジョブの時間帯情報を使用して UTC を判別します。

構造体のエレメントによっては、*time* が指す値が `gmtime64()` 用に表示される範囲に制限されないものもあります。

`mktime64()` に渡される tm_wday と tm_yday の値は無視され、戻りでこれらの正しい値が割り当てられます。

tm_isdst の値が正であるか 0 であるかにより、`mktime()` は、指定した時間で DST が有効になっているかいないかを最初に推定します。 *tm_isdst* が負の値の場合、 `mktime()` は、指定された時間で DST が有効になっているかどうかについて、判別を試みます。

注: この関数でサポートされる日付と時刻の範囲は、01/01/1970 00:00:00 から 12/31/9999 23:59:59 です。

戻り値

`mktime64()` 関数は、型 `time64_t` を持つ協定世界時 (UTC) を戻します。協定世界時を表示できない場合、または、指定された *time* が範囲外の場合、値 (time_t)(-1) は戻ります。指定された *time* が範囲外の場合、errno は EOVERFLOW に設定されます。

mktime64() の使用例

この例では、現在日付から 40 日と 16 時間後の曜日を出力します。

```
#include <stdio.h>
#include <time.h>

char *wday[] = { "Sunday", "Monday", "Tuesday", "Wednesday",
                 "Thursday", "Friday", "Saturday" };

int main(void)
{
  time64_t t1, t3;
  struct tm *t2;

  t1 = time64(NULL);
  t2 = localtime64(&t1);
  t2 -> tm_mday += 40;
  t2 -> tm_hour += 16;
  t3 = mktime64(t2);

  printf("40 days and 16 hours from now, it will be a %s ¥n",
         wday[t2 -> tm_wday]);
}

/*******************  Output should be similar to:  ***************

40 days and 16 hours from now, it will be a Sunday
*/
```

関連情報

- 41 ページの『asctime() ― 時間から文字ストリングへの変換』
- 43 ページの『asctime_r() ― 時間から文字ストリングへの変換 (再始動可能)』
- 75 ページの『ctime() ― 時間から文字ストリングへの変換』
- 77 ページの『ctime64() ― 時間から文字ストリングへの変換』
- 80 ページの『ctime64_r() ― 時間から文字ストリングへの変換 (再始動可能)』
- 78 ページの『ctime_r() ― 時間から文字ストリングへの変換 (再始動可能)』
- 167 ページの『gmtime() ― 時間の変換』
- 169 ページの『gmtime64() ― 時間の変換』
- 173 ページの『gmtime64_r() ― 時間の変換 (再始動可能)』
- 171 ページの『gmtime_r() ― 時間の変換 (再始動可能)』
- 192 ページの『localtime() ― 時間の変換』
- 194 ページの『localtime64() ― 時間の変換』
- 197 ページの『localtime64_r() ― 時間の変換 (再始動可能)』
- 195 ページの『localtime_r() ― 時間の変換 (再始動可能)』
- 228 ページの『mktime() ― 地方時の変換』
- 429 ページの『time() ― 現在時刻の判別』
- 430 ページの『time64() ― 現在時刻の判別』
- 19 ページの『<time.h>』

modf() ― 浮動小数点値の分離

フォーマット

```
#include <math.h>
double modf(double x, double *intptr);
```

言語レベル: ANSI

スレッド・セーフ: はい。

説明

modf() 関数は、浮動小数点値の *x* を小数と整数の部分に分けます。*x* の符号付き小数部分が戻ります。整数部分は、*intptr* が指す double 値として保管されます。小数部分と整数部分の両方には、*x* と同じ符号が指定されます。

戻り値

modf() 関数は、*x* の符号付き小数部分を戻します。

modf() の使用例

この例では、浮動小数点の数値 -14.876 を小数と整数のコンポーネントに分けます。

```
#include <math.h>
#include <stdio.h>

int main(void)
{
   double x, y, d;

   x = -14.876;
   y = modf(x, &d);

   printf("x = %lf¥n", x);
   printf("Integral part = %lf¥n", d);
   printf("Fractional part = %lf¥n", y);
}

/****************  Output should be similar to:  ******************

x = -14.876000
Integral part = -14.000000
Fractional part = -0.876000
*/
```

関連情報

- 113 ページの『fmod() ― 浮動小数点の剰余の計算』
- 137 ページの『frexp() ― 浮動小数点値の分離』
- 185 ページの『ldexp() ― 2 のべき乗の乗算』
- 9 ページの『<math.h>』

nextafter() ― nextafterl()― nexttoward() ― nexttowardl() ― 表現可能な次の浮動小数点値の計算

フォーマット

```
#include <math.h>
double nextafter(double x, double y);
long double nextafterl(long double x, long double y);
double nexttoward(double x, long double y);
long double nexttowardl(long double x, long double y);
```

言語レベル: ANSI

スレッド・セーフ: はい。

説明

nextafter()、nextafterl()、nexttoward()、および nexttowardl() 関数は、y の方向で x の後にくる次の表現可能な値を計算します。

戻り値

nextafter()、nextafterl()、nexttoward()、および nexttowardl() 関数は、y の方向で x の後にくる次の表現可能な値を計算します。x または y が NaN (非数字) の場合は NaN が戻され、errno が EDOM に設定されます。x が最大有限値で、結果が無限または表現可能ではない場合は、HUGE_VAL が戻され、errno に ERANGE が設定されます。

nextafter()、nextafterl()、nexttoward()、および nexttowardl()の使用例

この例では、浮動小数点の値を次のより大きな表現可能な値、および次のより小さな表現可能な値に変換します。変換した値は出力されます。

```
#include <stdio.h>
#include <math.h>
int main(void)
{
 double x, y;
 long double ld;

 x = nextafter(1.234567899, 10.0);
 printf("nextafter 1.234567899 is %#19.17g¥n" x);
 ld = nextafterl(1.234567899, -10.0);
 printf("nextafterl 1.234567899 is %#19.17g¥n" ld);

 x = nexttoward(1.234567899, 10.0);
 printf("nexttoward 1.234567899 is %#19.17g¥n" x);
 ld = nexttowardl(1.234567899, -10.0);
 printf("nexttowardl 1.234567899 is %#19.17g¥n" ld);

}

/***************** Output should be similar to: *****************
nextafter 1.234567899 is  1.2345678990000002
nextafterl 1.234567899 is 1.2345678989999997
nexttoward 1.234567899 is 1.2345678990000002
nexttowardl 1.234567899 is 1.2345678989999997

*/
```

関連情報

- 64 ページの『ceil() ― 整数の検索 >= 引数』
- 112 ページの『floor() ― 整数の検索 <=引数』
- 137 ページの『frexp() ― 浮動小数点値の分離』

- 232 ページの『modf() ― 浮動小数点値の分離』
- 9 ページの『<math.h>』

nl_langinfo() ― ロケール情報の検索

フォーマット

```
#include <langinfo.h>
#include <nl_types.h>
char *nl_langinfo(nl_item item);
```

言語レベル: XPG4

スレッド・セーフ: いいえ。

ロケール依存: この関数の振る舞いは、現行ロケールの LC_CTYPE、LC_MESSAGES、LC_MONETARY、LC_NUMERIC および LC_TIME カテゴリーの影響を受ける可能性があります。この関数は、コンパイル・コマンドに対して LOCALETYPE(*CLD) が指定されている場合には使用できません。詳細については、 549 ページの『CCSID およびロケールの理解』を参照してください。

説明

`nl_langinfo()` 関数は、現行ロケールから、*item* によって指定された要求情報が記述されているストリングを検索します。

現行ロケールからの次の情報の検索がサポートされています。

項目	説明
CODESET	文字形式のロケールの CCSID
D_T_FMT	フォーマット設定日時のストリング
D_FMT	日付形式ストリング
T_FMT	時刻形式ストリング
T_FMT_AMPM	a.m. または p.m. 時刻形式ストリング
AM_STR	A.M. 接尾辞
PM_STR	P.M. 接尾辞
DAY_1	週の初日の名前 (例えば、日曜日)
DAY_2	週の 2 番目の日の名前 (例えば、月曜日)
DAY_3	週の 3 番目の日の名前 (例えば、火曜日)
DAY_4	週の 4 番目の日の名前 (例えば、水曜日)
DAY_5	週の 5 番目の日の名前 (例えば、木曜日)
DAY_6	週の 6 番目の日の名前 (例えば、金曜日)
DAY_7	週の 7 番目の日の名前 (例えば、土曜日)
ABDAY_1	週の初日の省略名
ABDAY_2	週の 2 番目の日の省略名
ABDAY_3	週の 3 番目の日の省略名
ABDAY_4	週の 4 番目の日の省略名
ABDAY_5	週の 5 番目の日の省略名

ABDAY_6	週の 6 番目の日の省略名
ABDAY_7	週の 7 番目の日の省略名
MON_1	年の最初の月の名前
MON_2	年の 2 番目の月の名前
MON_3	年の 3 番目の月の名前
MON_4	年の 4 番目の月の名前
MON_5	年の 5 番目の月の名前
MON_6	年の 6 番目の月の名前
MON_7	年の 7 番目の月の名前
MON_8	年の 8 番目の月の名前
MON_9	年の 9 番目の月の名前
MON_10	年の 10 番目の月の名前
MON_11	年の 11 番目の月の名前
MON_12	年の 12 番目の月の名前
ABMON_1	年の最初の月の省略名
ABMON_2	年の 2 番目の月の省略名
ABMON_3	年の 3 番目の月の省略名
ABMON_4	年の 4 番目の月の省略名
ABMON_5	年の 5 番目の月の省略名
ABMON_6	年の 6 番目の月の省略名
ABMON_7	年の 7 番目の月の省略名
ABMON_8	年の 8 番目の月の省略名
ABMON_9	年の 9 番目の月の省略名
ABMON_10	年の 10 番目の月の省略名
ABMON_11	年の 11 番目の月の省略名
ABMON_12	年の 12 番目の月の省略名
ERA	年代記述セグメント
ERA_D_FMT	年代日付形式ストリング
ERA_D_T_FMT	年代日時形式ストリング
ERA_T_FMT	年代時刻形式ストリング
ALT_DIGITS	数字の代替シンボルのストリング
RADIXCHAR	基数文字
THOUSEP	1000 用の区切り文字
YESEXPR	肯定応答式
NOEXPR	否定応答式
YESSTR	はい/いいえクエリーの肯定応答
NOSTR	はい/いいえクエリーの否定応答
CRNCYSTR	通貨記号。記号が値の前に表れる場合は「-」、記号が値の後に表れる場合は「+」、または記号が基数文字に代わる場合は「.」。

戻り値

nl_langinfo() 関数は、アクティブな言語エリアまたは文化エリアに関する情報を含む、ヌル終了ストリングを指すポインターを戻します。アクティブな言語エリアまたは文化エリアは、最後の setlocale() 呼び出しによって判別されます。戻り値によって指示された配列は、その後に関数を呼び出すと変更されます。配列は、ユーザー・プログラムで変更しないでください。

項目が無効の場合には、関数は、空ストリングを指すポインターを戻します。

nl_langinfo() の使用例

この例では、nl_langinfo() 関数を使用して、コード・セットの名前を検索します。

```
#include <langinfo.h>
#include <locale.h>
#include <nl_types.h>
#include <stdio.h>

int main(void)
{
   printf("Current codeset is %s¥n", nl_langinfo(CODESET));
   return 0;
}

/**************************************************************************

  The output should be similar to:

  Current codeset is 37

**************************************************************************/
```

関連情報

- 187 ページの『localeconv() ― 環境からの情報の取得』
- 354 ページの『setlocale() ― ロケールの設定』
- 8 ページの『<langinfo.h>』
- 9 ページの『<nl_types.h>』

perror() ― エラー・メッセージの出力

フォーマット

```
#include <stdio.h>
void perror(const char *string);
```

言語レベル: ANSI

スレッド・セーフ: はい。

説明

perror() 関数は、エラー・メッセージを stderr に出力します。 *string* が NULL ではなく、ヌル文字を指示しない場合には、*string* によって指示されたストリングは、標準エラー・ストリームに出力され、その後ろにコロンとスペースが続きます。次に、errno の値と関連したメッセージが出力され、その後ろに改行文字が続きます。

正確な結果を得るには、エラーのあるライブラリー関数が戻った直後に perror() 関数を必ず呼び出してください。そうでないと、後続の呼び出しにより errno 値が変更されることがあります。

戻り値

戻り値はありません。

errno の値は、次のいずれかに設定されます。

値　　意味

EBADDATA
: メッセージ・データが無効です。

EBUSY
: レコードまたはファイルが使用中です。

ENOENT
: ファイルまたはライブラリーが見つかりません。

EPERM
: アクセス権限が不十分です。

ENOREC
: レコードが見つかりません。

EIOERROR
: リカバリー不能な入出力エラーが発生しました。

EIORECERR
: リカバリー可能な入出力エラーが発生しました。

perror() の使用例

この例では、ストリームを開こうと試みます。 fopen() が失敗した場合、この例ではメッセージが出力されてからプログラムが終了します。

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
   FILE *fh;

   if ((fh = fopen("mylib/myfile","r")) == NULL)
   {
      perror("Could not open data file");
      abort();
   }
}
```

関連情報

- 65 ページの『clearerr() ― エラー標識のリセット』
- 100 ページの『ferror() ― 読み取り/書き込みエラーのテスト』
- 384 ページの『strerror() ― ランタイム・エラー・メッセージを指すポインターの設定』
- 17 ページの『<stdio.h>』

pow() ― 累乗の計算

フォーマット

```
#include <math.h>
double pow(double x, double y);
```

言語レベル: ANSI

スレッド・セーフ: はい。

説明

pow() 関数は、*x* の *y* 乗の値を計算します。

戻り値

y が 0 の場合、pow() 関数は値 1 を戻します。*x* が 0 および *y* が負の場合、pow() 関数は errno を EDOM に設定して 0 を戻します。*x* および *y* の両方が 0 の場合、または *x* が負で *y* が整数以外の場合、pow() 関数は errno を EDOM に設定して 0 を戻します。errno 変数も、ERANGE に設定される可能性があります。結果がオーバーフローの場合、pow() 関数は、結果が大きい場合には +HUGE_VAL を戻し、結果が小さい場合には -HUGE_VAL を戻します。

pow() の使用例

この例では、2^3 の値を計算します。

```
#include <math.h>
#include <stdio.h>

int main(void)
{
   double x, y, z;

   x = 2.0;
   y = 3.0;
   z = pow(x,y);

   printf("%lf to the power of %lf is %lf¥n", x, y, z);
}

/*****************  Output should be similar to:  *****************

2.000000 to the power of 3.000000 is 8.000000
*/
```

関連情報

- 93 ページの『exp() ― 指数関数の計算』
- 198 ページの『log() ― 自然対数の計算』
- 199 ページの『log10() ― 基数 10 の対数の計算』
- 369 ページの『sqrt() ― 平方根の計算』
- 9 ページの『<math.h>』

printf() ― 定様式の文字の出力

フォーマット

```
#include <stdio.h>
int printf(const char *format-string, argument-list);
```

言語レベル: ANSI

スレッド・セーフ: はい。

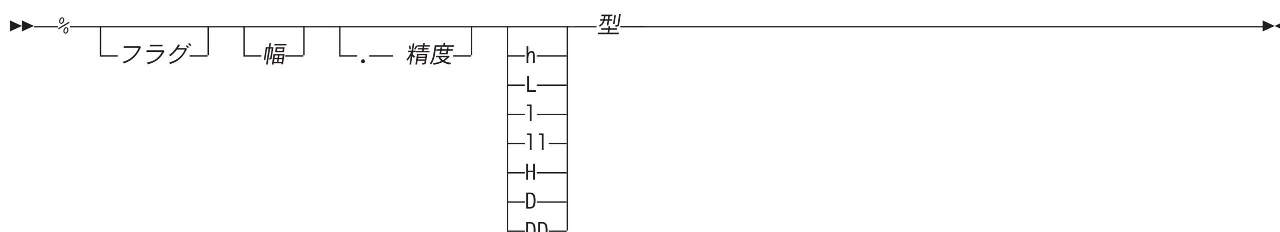
ロケール依存: この関数の振る舞いは、現行ロケールの LC_CTYPE カテゴリーおよび LC_NUMERIC カテゴリーの影響を受ける可能性があります。また、この振る舞いは、LOCALETYPE(*LOCALEUCS2) または LOCALETYPE(*LOCALEUTF) がコンパイル・コマンドに対して指定されている場合は、現行ロケールの LC_UNI_CTYPE カテゴリーの影響を受ける可能性もあります。詳細については、 549 ページの『CCSID およびロケールの理解』を参照してください。

説明

printf() 関数は、一連の文字および値をフォーマット設定し、標準出力ストリーム stdout に出力します。形式指定とは、パーセント記号 (%) で始まり、*format-string* に従って、*引数リスト*の出力フォーマットを判別するものです。 *format-string* はマルチバイト文字ストリングで、初期シフト状態で開始して終了します。

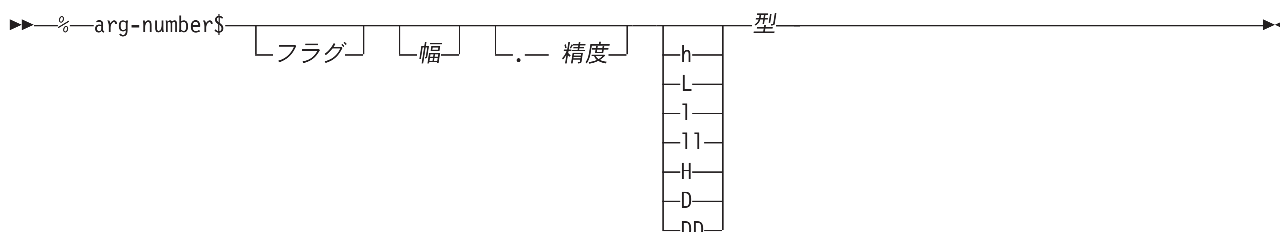
format-string は左から右へ読み取られます。最初の形式指定が見つかると、*format-string* 後の最初の引数の値が形式指定に従って、変換され出力されます。2 番目の形式指定は、*format-string* 後の 2 番目の引数を変換し出力させるというように、*format-string* の最後まで順に行われます。形式指定にあるよりも多くの引数が存在する場合は、余分の引数は評価され、無視されます。すべての形式指定に対して引数が足りない場合は、予期しない結果が引き起こされることになります。

形式指定の形式は、以下のとおりです。



変換の対象は、次の未使用の引数ではなく、引数リスト内の *format-string* の後で、「n 番目」の引数に適用するように指定できます。この場合、変換文字 % はシーケンス %n$ に置き換えられます。ここで n は範囲 [1,NL_ARGMAX] 内の 10 進の整数で、引数リスト内の引数の位置を与えます。この機能により、特定言語に適した順序で引数を選択する書式ストリングの定義が得られます。

代替の形式指定の形式は、以下のとおりです。



前のダイアグラムで説明した形式指定を使用して、引数リスト内の特定の項目を代わりに割り当てることができます。この形式指定と前の形式指定は、printf() への同じ呼び出しで混用できません。混用した場合は、予測不能な結果が発生します。

arg-number は正整数定数で、この場合、1 は引数リスト内の最初のエントリーを示します。 arg-number は、引数リスト内のエントリー数より多くできません。そうでないと、予期しない結果になります。 arg-number はまた、NL_ARGMAX より多くならないようにします。

%n$ 形式の変換指定を含む書式ストリングでは、引数リスト内の番号を持つ引数は、必要な回数だけ、書式ストリングから参照することができます。

%n$ 形式の変換指定を含む書式ストリングでは、フィールド幅または精度はシーケンス *m$ によって示される場合がありますが、ここで m は範囲 [1,NL_ARGMAX] 内の 10 進の整数で、以下の例のように、引数リスト内 (書式引数の後) におけるフィールド幅または精度を含む整数引数の位置を与えます。

```
printf("%1$d:%2$.*3$d:%4$.*3$d\n", hour, min, precision, sec);
```

format-string には、番号を持つ引数指定 (すなわち、%n$ および *m$)、または番号を持たない引数指定 (すなわち、% および *) を含めることができますが、通常、両方を含めることはできません。唯一の例外として、%% は %n$ 形成で混用することが可能です。番号を持つ引数指定と番号を持たない引数指定を format-string ストリングで混用した場合、結果は予期できません。番号を持つ引数指定が使用されると、「n 番目」の引数を指定するには、最初から (n-1) 番目の引数が書式ストリング内に指定される必要があります。

形式指定の各フィールドは、単一文字、または数字で、特殊な形式オプションを示します。*type* 文字 (最後のオプションの形式フィールドの後に現れる) は、関連する引数を文字、ストリング、数字、またはポインターとして解釈されるかどうかを判別します。最もシンプルな形式指定には、パーセント記号と *type* 文字のみが含まれます (例えば、%s)。

オプションの以下のフィールドは、フォーマット設定のその他の側面を制御します。

フィールド
説明

フラグ
符号、ブランク、10 進小数点、8 進数および 16 進数の接頭部の出力と出力の位置調整、および wchar_t 精度単位のセマンティクス。

width
出力される最小バイト数

精度
245 ページの表 4 を参照してください。

h、l、ll、L、H、D、DD
予期される引数のサイズは以下のとおりです。

h
d、i、o、u、x、X、および n 型の接頭部で、引数が short 型整数または符号なし short 型整数であることを指定する。

l
d、i、o、u、x、X、および n 型の接頭部で、引数が long int または unsigned long int であることを指定する。

ll
d、i、o、u、x、X、および n 型の接頭部で、引数が long long 型整数または符号なし long long 型整数であることを指定する。

L
e、E、f、F、g、または G 型の接頭部で、引数が long 型 double であることを示す。

H
e、E、f、F、g、または G 型の接頭部で、引数が _Decimal32 であることを示す。

D
e、E、f、F、g、または G 型の接頭部で、引数が _Decimal64 であることを示す。

DD e、E、f、F、g、または G 型の接頭部で、引数が _Decimal128 であることを示す。

形式指定の各フィールドについては、以下で詳細に説明します。パーセント記号 (%) の後に、形式フィールドとして意味を持たない文字が続いた場合は、文字は単に `stdout` にコピーされます。例えば、パーセント記号文字を出力するには、`%%` を使用します。

type 文字とその意味を、次の表で説明します。

文字	引数	出力フォーマット
d, i	整数	符号付き 10 進整数。
u	整数	符号なし 10 進整数。
o	整数	符号なし 8 進整数。
x	整数	符号なし 16 進整数。(abcdef を使用)
X	整数	符号なし 16 進整数。(ABCDEF を使用)
D(n,p)	パック 10 進数	形式は [-] *dddd.dddd* 。小数点の後の桁数は指定の精度と等しくなります。精度が欠落している場合のデフォルトは p。精度がゼロで `#` フラグが指定されていない場合、小数点文字は表示されません。 n および p が `*` の場合は引数リストの引数が値を提供します。 n および p は、引数リストでフォーマット設定されている値の前になければなりません。少なくとも 1 文字は小数点の前になければなりません。値は、適切な桁数に丸められます。
f	浮動小数点	形式 [-]*dddd.dddd* を持つ符号付き値。*dddd* は 1 つ以上の 10 進桁。小数点の前の桁数は、数字の大きさによって異なります。小数点の後の桁数は、要求した精度と等しくなります。[2]
F	浮動小数点	小文字英字の代わりに大文字英字が使用される点を除き、f 形式と同じです。[2]
e	浮動小数点	[-]*d.dddd* e[*sign*]*ddd* の形式を持つ、符号のついた値。ここで、*d* は、単一 10 進数字で、*dddd* は、1 つ以上の 10 進数字、*dddd* は 1 つ以上の 10 進数字、*ddd* は 2 つまたは 4 つの 10 進数字、そして、*sign* は + または -。[2]
E	浮動小数点	小文字英字の代わりに大文字英字が使用される点を除き、e 形式と同じです。[2]
g	浮動小数点	f または e 形式の符号付き値の出力。e 形式は、値の指数が -4 未満または *精度* より大きいときにだけ使用されます。末尾のゼロは切り捨てられ、小数点 1 つ以上の桁が続く場合にのみ、小数点が表れます。[2]
G	浮動小数点	小文字英字の代わりに大文字英字が使用される点を除き、g 形式と同じです。[2]
c	文字 (バイト)	単一文字。
s	ストリング	文字 (バイト) は、最初のヌル文字 (¥0) まで、または *精度*に達するまで出力されます。
n	整数へのポインター	現時点までに、正常に *stream* またはバッファーへ出力された文字 (バイト) の数。この値は、アドレスが引数として与えられる整数に保管されます。

文字	引数	出力フォーマット
p	ポインター	出力可能文字のシーケンスに変換されるポインター。以下のいずれかにすることができます。 • スペース・ポインター • システム・ポインター • 呼び出しポインター • プロシージャー・ポインター • オープン・ポインター • サスペンド・ポインター • データ・ポインター • ラベル・ポインター
lc または C	ワイド文字	(wchar_t) 文字は、wctomb() の呼び出しによって行われた場合と同様に、マルチバイト文字に変換され、この文字が出力されます。[1]
ls または S	ワイド文字	最初の (wchar_t) ヌル文字 (L¥0) まで、または、精度に達するまで、(wchar_t) 文字は、wcstombs() の呼び出しによって行われた場合と同様にマルチバイト文字に変換され、これらの文字は出力されます。引数がヌル・ストリングの場合は (null) が出力されます。[1]

注:

1. 詳細は、`wctomb()` 関数に関する文書または `wcstombs()` に関する文書を参照してください。 553 ページの『ワイド文字』で追加情報を見つけることもできます。
2. H、D、または DD 形式サイズ指定子が使用されていない場合、保障される出力は 15 桁の有効数字のみです。

次のリストで、i5/OS ポインターで出力される値の形式を示し、出力される値のコンポーネントの簡単な説明を提供します。

スペース・ポインター: SPP:Context:Object:Offset:AG

Context: コンテキストのタイプ、サブタイプ、および名前
Object: オブジェクトのタイプ、サブタイプ、および名前
Offset: スペース内のオフセット
AG: 活動化グループ ID

システム・ポインター: SYP:Context:Object:Auth:Index:AG

Context: コンテキストのタイプ、サブタイプ、および名前
Object: オブジェクトのタイプ、サブタイプ、および名前
Auth: 権限
Index: ポインターに関連付けられたインデックス
AG: 活動化グループ ID

呼び出しポインター: IVP:Index:AG

Index: ポインターに関連付けられたインデックス
AG: 活動化グループ ID

プロシージャー・ポインター:　PRP:Index:AG

Index:　ポインターに関連付けられたインデックス
AG:　活動化グループ ID

サスペンド・ポインター:　SUP:Index:AG

Index:　ポインターに関連付けられたインデックス
AG:　活動化グループ ID

データ・ポインター:　DTP:Index:AG

Index:　ポインターに関連付けられたインデックス
AG:　活動化グループ ID

ラベル・ポインター:　LBP:Index:AG

Index:　ポインターに関連付けられたインデックス
AG:　活動化グループ ID

NULL ポインター:　NULL

i5/OS オペレーティング・システム上におけるポインターの出力および走査には、次の制約事項が適用されます。

- ポインターが同じ活動化グループから出力され、走査されて戻された場合、走査されて戻されたポインターが出力されたポインターと等しいかどうか確認が行われます。
- `scanf()` ファミリー関数が別の活動化グループによって出力されたポインターを走査する場合、`scanf()` ファミリー関数はそのポインターを NULL に設定します。
- ポインターをテラスペース環境で出力する場合、ポインターの 16 進値のみが出力されます。%#p を使用した場合も、これらの結果は同じです。

i5/OS ポインターの使用法についての詳細は、「*IBM Rational Development Studio for i: ILE C/C++ Programmer's Guide*」を参照してください。

INFINITY または非数字 (NaN) の浮動小数点の値が e、f、または g 形式を使用してフォーマットされた場合、出力ストリングは `infinity` または `nan` です。INFINITY または非数字 (NaN) の浮動小数点の値が E、F、または G 形式を使用してフォーマットされた場合、出力ストリングは `INFINITY` または `NAN` です。

フラグ文字とその意味は以下のとおりです (複数のフラグを形式指定に入れることができます)。

フラグ	意味	デフォルト
-	フィールド幅内で結果を左そろえする。	右そろえ。
+	出力値が符号付き型の場合は、出力値の前に符号 (+ または -) が付けられる。	負の符号付き値にのみ符号が現れる (-)。
ブランク (' ')	出力値が符号付きで正の場合は、出力値の前にブランクが付けられる。+ フラグはブランク フラグを上書きする (両方のフラグが現れ、正の符号付き値が符号と一緒に出力される場合)。	ブランクなし。

フラグ	意味	デフォルト
#	o、x、または X 形式で使用される場合、# フラグは、ゼロ以外の出力値の先頭に 0、0x、または 0X をそれぞれ付ける。	接頭部なし。
	f、F、D(n,p)、e、または E 形式で使用される場合、# フラグは、常に、出力値に小数点を強制的に含める。	小数点は、数字がそのあとに続いている場合にだけ現れる。
	g または G 形式で使用される場合、# フラグは、常に、出力値に小数点を含むことを強制し、後続ゼロの切り捨てが起こらないようにする。	小数点は、数字がそのあとに続いている場合にだけ現れる。後続ゼロは切り捨てられる。
	ls または S 形式で使用される場合、# フラグは、文字のサイズに関係なく、精度が文字で測定されるようにする。例えば、1 バイト文字を出力する場合、精度が 4 だと、結果として 4 バイトで出力されます。 2 バイト文字を出力する場合、精度が 4 だと、結果として 8 バイトで出力されます。	精度は、出力される最大バイト数を示す。
	p 形式で使用された場合、# フラグはポインターを 16 進数字に変換します。テラスペース環境の場合を除き、これらの 16 進数字をポインター変換し直すことはできません。	出力可能文字のシーケンスに変換されるポインター。
0	d、i、D(n,p) o、u、x、X、e、E、f、F、g、または G 形式で使用されると、0 フラグによって、先行 0 で出力をフィールド幅まで埋め込む。 0 フラグは、精度が整数に指定された場合、または - フラグが指定された場合は、無視される。	スペースを埋め込む。 D(n,p) に埋め込むスペースはなし。

フラグは、c、lc、d、i、u、または s 型と一緒に使用してはなりません。

Width は、負以外の 10 進の整数で、出力される最小文字数を制御します。出力値の文字数 (バイト) が指定した *width* 未満の場合、最小幅になるまで (- フラグを指定したかどうかに応じて)、左側または右側にブランクが追加されます。

width では値の切り捨ては行われません。出力値の文字数 (バイト) が指定された *width* より大きいか、または *width* が指定されない場合は、すべての値の文字が (*精度* 指定に従って) 出力されます。

ls または S 型の場合、*width* はバイトで指定されます。出力値のバイト数が指定した幅未満の場合、最小幅になるまで (- フラグを指定したかどうかに応じて)、左側または右側に 1 バイトのブランクが追加されます。

width の指定には、アスタリスク (*) が可能です。その場合は、引数リストの引数が値を提供します。 *width* 引数は、引数リストでフォーマット設定されている値の前になければなりません。

精度 にはピリオドが前にくる、負以外の 10 進の整数で、出力される文字数、または小数点以下の桁数を指定します。 *width* の指定とは異なり、*精度* では出力値の切り捨てや、浮動小数点数値または パック 10 進数 値の丸めが起こり得ます。

精度 の指定には、アスタリスク (*) が可能です。その場合は、引数リストの引数が値を提供します。*精度* 引数は、引数リストでフォーマット設定されている値の前になければなりません。

精度 値の解釈と *精度* が省略された場合のデフォルトは、次の表に示されるように、*type* によって異なります。

表 4. 精度の値

タイプ	意味	デフォルト
i d u o x X	*精度* は出力される数字の最小桁数を指定する。引数内の数字の桁数が *精度* より少ない場合、出力値の左側をゼロで埋め込む。数字の桁数が *精度* を超えても、値は切り捨てられない。	*精度* が 0 であるか、すべて省略された場合、または、ピリオド (.) が後に数字を伴わずに指定された場合、*精度* は 1 に設定される。
f F D(n,p) e E	*精度* は小数点の後ろに出力される数字の桁数を指定する。最後の出力数字は丸められる。	f、F、e および E のデフォルトの *精度* は 6。 D(n,p) のデフォルトの *精度* は p。*精度* が 0 または後ろに数値を伴わずに表示されたピリオドの場合は、小数点は出力されない。
g G	*精度* は、有効数字出力の最大桁数を指定する。	すべての有効数字が出力される。デフォルトの *精度* は 6。
c	影響なし。	文字が出力される。
lc	影響なし。	wchar_t 文字は変換され、結果であるマルチバイト文字は出力される。
s	*精度* は出力される文字 (バイト) の最大数を指定する。余分な *精度* での文字 (バイト) は出力されない。	文字はヌル文字が検出されるまで出力される。
ls	*精度* は出力されるバイトの最大数を指定する。*精度* を超過するバイトは出力されない。しかしマルチバイトの整合性は常に維持される。	wchar_t 文字は変換され、結果であるマルチバイト文字は出力される。

戻り値

`printf()` 出力されるバイト数を戻します。 errno の値は、次のいずれかに設定されます。

値　　意味

EBADMODE
指定されたファイル・モードが無効です。

ECONVERT
変換エラーが発生しました。

EIOERROR
リカバリー不能な入出力エラーが発生しました。

EIORECERR
リカバリー可能な入出力エラーが発生しました。

EILSEQ
無効なマルチバイト文字シーケンスが検出されました。

EPUTANDGET
読み取り操作の後、正しくない書き込み操作が発生しました。

ESTDOUT
　　stdout をオープンできません。

注: printf() 関数の基数文字は、ロケールに依存します。基数文字とは、形式タイプ f、F、D(n,p)、e、E、g、および G のストリング・パラメーターの # フラグで使用される小数点のことです。

printf() の使用例

この例では、様々なフォーマットでデータを出力します。

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    char ch = 'h', *string = "computer";
    int count = 234, hex = 0x10, oct = 010, dec = 10;
    double fp = 251.7366;
    wchar_t wc = (wchar_t)0x0058;
    wchar_t ws[4];

    printf("1234567890123%n4567890123456789¥n¥n", &count);
    printf("Value of count should be 13; count = %d¥n¥n", count);
    printf("%10c%5c¥n", ch, ch);
    printf("%25s¥n%25.4s¥n¥n", string, string);
    printf("%f    %.2f    %e    %E¥n¥n", fp, fp, fp, fp);
    printf("%i    %i     %i¥n¥n", hex, oct, dec);
}
/*****************  Output should be similar to:  *****************

234   +234    000234     EA    ea     352

12345678901234567890123456789

Value of count should be 13; count = 13


  h    h
          computer
              comp

251.736600    251.74    2.517366e+02    2.517366E+02

16    8     10

*******************************************************************/
```

printf() の使用例

```
     #include <stdio.h>
     #include <stdlib.h>
     #include <locale.h>
     /* This program is compiled with LOCALETYPE(*LOCALEUCS2) and            */
     /* SYSIFCOPT(*IFSIO)                                                  */
     /* We will assume the locale setting is the same as the CCSID of the */
     /* job.  We will also assume any files involved have a CCSID of      */
     /* 65535 (no convert).  This way if printf goes to the screen or     */
     /* a file the output will be the same.                               */
    int main(void)
    {
        wchar_t wc = 0x0058;     /* UNICODE X */
        wchar_t ws[4];
        setlocale(LC_ALL,
```

```
        "/QSYS.LIB/EN_US.LOCALE"); /* a CCSID 37 locale */
       ws[0] = 0x0041;        /* UNICODE A   */
       ws[1] = (wchar_t)0x0042;        /* UNICODE B   */
       ws[2] = (wchar_t)0x0043;        /* UNICODE C   */
       ws[3] = (wchar_t)0x0000;
       /* The output displayed is CCSID 37  */
       printf("%lc   %ls\n\n",wc,ws);
       printf("%lc   %.2ls\n\n",wc,ws);

       /* Now let's try a mixed-byte CCSID example  */
       /* You would need a device that can handle mixed bytes to  */
       /* display this correctly.                         */

       setlocale(LC_ALL,
       "/QSYS.LIB/JA_JP.LOCALE");/* a CCSID 5026 locale */

       /* big A means an A that takes up 2 bytes on the screen   */
       /* It will look bigger then single byte A                 */
       ws[0] = (wchar_t)0xFF21;        /* UNICODE big A   */
       ws[1] = (wchar_t)0xFF22;        /* UNICODE big B   */
       ws[2] = (wchar_t)0xFF23;        /* UNICODE big C   */
       ws[3] = (wchar_t)0x0000;
       wc = 0xff11;                    /* UNICODE big 1   */

       printf("%lc   %ls\n\n",wc,ws);

       /* The output of this printf is not shown below and it   */
       /* will differ depending on the device you display it on,*/
       /* but if you looked at the string in hex it would look  */
       /* like this:  0E42F10F404040400E42C142C242C30F          */
       /* 0E is shift out, 0F is shift in, and 42F1 is the      */
       /* big 1 in CCSID   5026  */

       printf("%lc   %.4ls\n\n",wc,ws);

       /* The output of this printf is not shown below either. */
       /* The hex would look like:                             */
       /* 0E42F10F404040400E42C10F                             */
       /* Since the precision is in bytes we only get 4 bytes  */
       /* of the string.         */

       printf("%lc   %#.2ls\n\n",wc,ws);

       /* The output of this printf is not shown below either. */
       /* The hex would look like:                             */
       /* 0E42F10F404040400E42C142C20F                         */
       /* The # means precision is in characters reguardless   */
       /* of size.  So we get 2 characters of the string.      */
   }
/*****************  Output should be similar to:  *****************


X     ABC

X     AB

*******************************************************************/
```

printf() の使用例

```
#include <stdio.h>
#include <stdlib.h>
#include <locale.h>
/* This program is compile LOCALETYPE(*LOCALE)  and               */
/* SYSIFCOPT(*IFSIO)    */
int main(void)
{
```

```
    wchar_t wc = (wchar_t)0x00C4;     /*  D */
    wchar_t ws[4];
    ws[0] = (wchar_t)0x00C1;        /* A   */
    ws[1] = (wchar_t)0x00C2;        /* B   */
    ws[2] = (wchar_t)0x00C3;        /* C   */
    ws[3] = (wchar_t)0x0000;
    /* The output displayed is CCSID 37  */
    printf("%lc   %ls\n\n",wc,ws);

    /* Now let's try a mixed-byte CCSID example  */
    /* You would need a device that can handle mixed bytes to  */
    /* display this correctly.                         */

    setlocale(LC_ALL,
    "/QSYS.lib/JA_JP.LOCALE"); /* a CCSID 5026 locale */

    /* big A means an A that takes up 2 bytes on the screen   */
    /* It will look bigger than single byte A                 */

    ws[0] = (wchar_t)0x42C1;        /* big A   */
    ws[1] = (wchar_t)0x42C2;        /* big B   */
    ws[2] = (wchar_t)0x42C3;        /* big C   */
    ws[3] = (wchar_t)0x0000;
    wc = 0x42F1;                    /* big 1   */

    printf("%lc   %ls¥n¥n",wc,ws);

    /* The output of this printf is not shown below and it   */
    /* will differ depending on the device you display it on,*/
    /* but if you looked at the string in hex it would look  */
    /* like this:  0E42F10F404040400E42C142C242C30F          */
    /* 0E is shift out, 0F is shift in, and 42F1 is the      */
    /* big 1 in CCSID   5026  */

    printf("%lc   %.4ls\n\n",wc,ws);

    /* The output of this printf is not shown below either. */
    /* The hex would look like:                             */
    /* 0E42F10F404040400E42C10F                             */
    /* Since the precision is in bytes we only get 4 bytes  */
    /* of the string.         */

    printf("%lc   %#.2ls\n\n",wc,ws);

    /* The output of this printf is not shown below either. */
    /* The hex would look like:                             */
    /* 0E42F10F404040400E42C142C20F                         */
    /* The # means precision is in characters regardless   */
    /* of size.  So we get 2 characters of the string.      */
}
/*****************  Output should be similar to:  *****************


D     ABC


*******************************************************************/
```

関連情報

- 122 ページの『fprintf() — フォーマット済みデータのストリームへの書き込み』
- 138 ページの『fscanf() — フォーマット済みデータの読み取り』
- 344 ページの『scanf() — データの読み取り』
- 368 ページの『sprintf() — フォーマット設定データのバッファーへの出力』
- 371 ページの『sscanf() — データの読み取り』

- 444 ページの『vfprintf() — ストリームへの引数データの出力』
- 452 ページの『vprintf() — 引数データの出力』
- 456 ページの『vsprintf() — 引数データのバッファーへの出力』
- 527 ページの『wprintf() — データのワイド文字としてのフォーマット設定と出力』
- 17 ページの『<stdio.h>』

putc() - putchar() — 文字の書き込み

フォーマット

```
#include <stdio.h>
int putc(int c, FILE *stream);
int putchar(int c);
```

言語レベル: ANSI

スレッド・セーフ: いいえ。`#undef putc` または `#undef putchar` は、`putc` または `putchar` 関数のマクロ・バージョンの代わりに、これらの関数の呼び出しを許可します。関数はスレッド・セーフです。

説明

`putc()` 関数は、*c* を unsigned char に変換してから、現在位置で *c* を出力 *stream* に書き込みます。`putchar()` は、`putc(`*c*`, stdout)` と同等です。

`putc()` 関数はマクロとして定義可能なため、引数を複数回にわたって評価することができます。

`putc()` および `putchar()` 関数は、type=record でオープンされたファイルではサポートされません。

戻り値

`putc()` および `putchar()` 関数は、書き込まれた文字を戻します。 EOF の戻り値はエラーを示します。

errno の値は、次のいずれかに設定されます。

値　　意味

ECONVERT
: 変換エラーが発生しました。

EPUTANDGET
: 読み取り操作の後、正しくない書き込み操作が発生しました。

EIOERROR
: リカバリー不能な入出力エラーが発生しました。

EIORECERR
: リカバリー可能な入出力エラーが発生しました。

`putc()` の使用例

この例では、バッファーの内容をデータ・ストリームに書き込みます。この例では、for 文の本体は NULL ですが、これは、この例ではテスト式内で書き込み操作を実行するためです。

```
#include <stdio.h>
#include <string.h>

#define  LENGTH 80

int main(void)
{
   FILE *stream = stdout;
   int i, ch;
   char buffer[LENGTH + 1] = "Hello world";

   /* This could be replaced by using the fwrite routine */
   for ( i = 0;
        (i < strlen(buffer)) && ((ch = putc(buffer[i], stream)) !=     EOF);
         ++i);
}

/********************  Expected output:  *************************

Hello world
*/
```

関連情報

- 124 ページの『fputc() ― 文字の書き込み』
- 152 ページの『fwrite() ― 項目の書き込み』
- 158 ページの『getc() - getchar() ― 文字の読み取り』
- 251 ページの『puts() ― ストリングの書き込み』
- 252 ページの『putwc() ― ワイド文字の書き込み』
- 254 ページの『putwchar() ― ワイド文字の stdout への書き込み』
- 17 ページの『<stdio.h>』

putenv() ― 環境変数の変更/追加

フォーマット

```
#include <stdlib.h>
int putenv(const char *varname);
```

言語レベル: XPG4

スレッド・セーフ: はい

ジョブ CCSID インターフェース: この関数に送信される文字データは、すべてジョブの CCSID 内にあると想定されます。この関数によって戻された文字データは、すべてジョブの CCSID 内にあります。詳しくは、 549 ページの『CCSID およびロケールの理解』を参照してください。

説明

`putenv()` 関数は、既存の変数を変更するか、または新しい変数を作成することにより、環境変数の値を設定します。*varname* パラメーターは形式 var=x のストリングを指しますが、ここで x は、環境変数 `var` の新規の値です。

name には、ブランクまたは等号 (=) を使用できません。例えば、次のような場合、

```
PATH NAME=/my_lib/joe_user
```

は無効ですが、これは、PATH と NAME の間にブランクがあるためです。同様に、

```
PATH=NAME=/my_lib/joe_user
```

は無効ですが、これは、PATH と NAME の間に等号があるためです。システムは、最初の等号の後に続くすべての文字を環境変数の値として解釈します。

戻り値

putenv() 関数は、正常に実行された場合には 0 を戻します。失敗した場合、putenv() は -1 を戻し、エラーを示す errno が設定されます。

putenv() の使用例

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
   char *pathvar;

   if (-1 == putenv("PATH=/:/home/userid")) {
      printf("putenv failed ¥n");
      return EXIT_FAILURE;
   }
   /* getting and printing the current environment path     */

   pathvar = getenv("PATH");
   printf("The current path is: %s¥n", pathvar);
   return 0;
}

/************************************************************
   The output should be:

   The current path is: /:/home/userid
```

関連情報

- 160 ページの『getenv() — 環境変数の検索』
- 18 ページの『<stdlib.h>』

puts() — ストリングの書き込み

フォーマット

```
#include <stdio.h>
int puts(const char *string);
```

言語レベル: ANSI

スレッド・セーフ: はい。

説明

puts() 関数は、指定された *string* を標準出力ストリーム stdout に書き込み、改行文字を出力に付加します。終了のヌル文字は書き込まれません。

戻り値

puts() 関数は、エラーが起きた場合、EOF を戻します。戻り値が負ではない場合、エラーが起きていないことを示します。

errno の値は、次のいずれかに設定されます。

値　　意味

ECONVERT
　　変換エラーが発生しました。

EPUTANDGET
　　読み取り操作の後、正しくない書き込み操作が発生しました。

EIOERROR
　　リカバリー不能な入出力エラーが発生しました。

EIORECERR
　　リカバリー可能な入出力エラーが発生しました。

puts() の使用例

この例では、Hello World を stdout に書き込みます。

```
#include <stdio.h>

int main(void)
{
  if ( puts("Hello World") == EOF )
    printf( "Error in puts¥n" );
}

/************************  Expected output:  *********************

Hello World
*/
```

関連情報

- 127 ページの『fputs() ― ストリングの書き込み』
- 130 ページの『fputws() ― ワイド文字ストリングの書き込み』
- 162 ページの『gets() ― 行の読み取り』
- 249 ページの『putc() - putchar() ― 文字の書き込み』
- 『putwc() ― ワイド文字の書き込み』
- 17 ページの『<stdio.h>』

putwc() ― ワイド文字の書き込み

フォーマット

```
#include <stdio.h>
#include <wchar.h>
wint_t putwc(wint_t wc, FILE *stream);
```

言語レベル: ANSI

スレッド・セーフ: はい。

ロケール依存: この関数の振る舞いは、現行ロケールの LC_CTYPE カテゴリーの影響を受ける可能性があります。この振る舞いは、LOCALETYPE(*LOCALEUCS2) または LOCALETYPE(*LOCALEUTF) をコンパイル・コマンドで指定した場合も、現行ロケールの LC_UNI_CTYPE カテゴリーによって影響を受ける可能性があります。この関数は、コンパイル・コマンドに対して LOCALETYPE(*CLD) が指定されている場合には使用できません。詳細については、 549 ページの『CCSID およびロケールの理解』を参照してください。

統合ファイル・システム・インターフェース: この関数は、コンパイル・コマンドに対して SYSIFCOPT(*NOIFSIO) が指定されている場合には使用できません。

ワイド文字関数: 詳細については、 553 ページの『ワイド文字』を参照してください。

説明

`putwc()` 関数は、ワイド文字 *wc* を現在位置でストリームに書き込みます。また、ストリーム用のファイル位置標識を適切に先に進めます。 `putwc()` 関数は、`putwc()` がプラットフォームによってはマクロとして実装されている点を除き、`fputwc()` 関数と同等です。したがって、移植性の観点から、`putwc()` に対する stream 引数は、副次作用のある式にはしないでください。

非ワイド文字関数を `putwc()` 関数と共に同じストリーム上で使用すると、予期しない振る舞いが生じる結果となります。 `putwc()` 関数の呼び出し後、EOF に到達していない限り、ストリームに対する書き込み関数を呼び出す前にバッファーをフラッシュするか、またはストリーム・ポインターの位置を変更します。ストリームに対する書き込み操作の後、`putwc()` 関数を呼び出す前にバッファーをフラッシュするか、ストリーム・ポインターの位置を変更してください。

戻り値

`putwc()` 関数は、書き込まれたワイド文字を戻します。書き込みエラーが発生すると、ストリームのエラー標識が設定され、WEOF が戻ります。ワイド文字のマルチバイト文字への変換中にエンコード・エラーが発生すると、`putwc()` 関数は errno に EILSEQ を設定し、WEOF を戻します。

`putwc()` の errno 値について詳しくは、124 ページの『fputc() ― 文字の書き込み』を参照してください。

`putwc()` の使用例

次の例では、`putwc()` 関数を使用して *wcs* 内のワイド文字をマルチバイト文字に変換してから、ファイル **putwc.out** に書き込みます。

```
#include <stdio.h>
#include <stdlib.h>
#include <wchar.h>
#include <errno.h>

int main(void)
{
   FILE    *stream;
   wchar_t *wcs = L"A character string.";
   int     i;

   if (NULL == (stream = fopen("putwc.out", "w"))) {
      printf("Unable to open: ¥"putwc.out¥".¥n");
      exit(1);
   }

   for (i = 0; wcs[i] != L'¥0'; i++) {
      errno = 0;
      if (WEOF == putwc(wcs[i], stream)) {
         printf("Unable to putwc() the wide character.¥n"
                "wcs[%d] = 0x%lx¥n", i, wcs[i]);
         if (EILSEQ == errno)
            printf("An invalid wide character was encountered.¥n");
         exit(1);
      }
   }
   fclose(stream);
   return 0;

   /****************************************************************
      The output file putwc.out should contain :

      A character string.
   ****************************************************************/
}
```

関連情報

- 124 ページの『fputc() ― 文字の書き込み』
- 128 ページの『fputwc() ― 文字の書き込み』
- 130 ページの『fputws() ― ワイド文字ストリングの書き込み』
- 163 ページの『getwc() ― ストリームからのワイド文字の読み取り』
- 249 ページの『putc() - putchar() ― 文字の書き込み』
- 『putwchar() ― ワイド文字の stdout への書き込み』
- 17 ページの『<stdio.h>』
- 20 ページの『<wchar.h>』

---

putwchar() ― ワイド文字の stdout への書き込み

フォーマット

```
#include <wchar.h>
wint_t putwchar(wint_t wc);
```

言語レベル: ANSI

スレッド・セーフ: はい。

ロケール依存: この関数の振る舞いは、現行ロケールの LC_CTYPE カテゴリーの影響を受ける可能性があります。この振る舞いは、LOCALETYPE(*LOCALEUCS2) または LOCALETYPE(*LOCALEUTF) をコンパイル・コマンドで指定した場合も、現行ロケールの LC_UNI_CTYPE カテゴリーによって影響を受ける可能性があります。この関数は、コンパイル・コマンドに対して LOCALETYPE(*CLD) が指定されている場合には使用できません。詳細については、 549 ページの『CCSID およびロケールの理解』を参照してください。

統合ファイル・システム・インターフェース: この関数は、コンパイル・コマンドに対して SYSIFCOPT(*NOIFSIO) が指定されている場合には使用できません。

ワイド文字関数: 詳細については、 553 ページの『ワイド文字』を参照してください。

説明

`putwchar()` 関数は、ワイド文字 *wc* をマルチバイト文字に変換してから、`stdout` に書き込みます。`putwchar()` 関数への呼び出しは、`putwc(wc, stdout)` と同等です。

非ワイド文字関数を `putwchar()` 関数と共に同じストリーム上で使用すると、予期しない振る舞いが生じる結果となります。 `putwchar()` 関数の呼び出し後、EOF に到達していない限り、ストリームに対する書き込み関数を呼び出す前にバッファーをフラッシュするか、またはストリーム・ポインターの位置を変更します。ストリームに対する書き込み操作の後、`putwchar()` 関数を呼び出す前にバッファーをフラッシュするか、ストリーム・ポインターの位置を変更してください。

戻り値

`putwchar()` 関数は、書き込まれたワイド文字を戻します。書き込みエラーが発生すると、`putwchar()` 関数はストリームのエラー標識を設定し、WEOF を戻します。ワイド文字のマルチバイト文字への変換中にエンコード・エラーが発生すると、`putwchar()` 関数は errno に EILSEQ を設定し、WEOF を戻します。

`putwc()` の errno 値について詳しくは、124 ページの『fputc() — 文字の書き込み』を参照してください。

`putwchar()` の使用例

この例では、`putwchar()` 関数を使用して *wcs* にストリングを書き込みます。

```
#include <stdio.h>
#include <wchar.h>
#include <errno.h>
#include <stdlib.h>

int main(void)
{
   wchar_t *wcs = L"A character string.";
   int     i;

   for (i = 0; wcs[i] != L'¥0'; i++) {
      errno = 0;
      if (WEOF == putwchar(wcs[i])) {
         printf("Unable to putwchar() the wide character.¥n");
         printf("wcs[%d] = 0x%lx¥n", i, wcs[i]);
         if (EILSEQ == errno)
            printf("An invalid wide character was encountered.¥n");
         exit(EXIT_FAILURE);
      }
   }
   return 0;

   /****************************************************************
      The output should be similar to :

      A character string.
   ****************************************************************/
}
```

関連情報

- 124 ページの『fputc() ― 文字の書き込み』
- 128 ページの『fputwc() ― 文字の書き込み』
- 130 ページの『fputws() ― ワイド文字ストリングの書き込み』
- 165 ページの『getwchar() ― STDIN からのワイド文字の取得』
- 249 ページの『putc() - putchar() ― 文字の書き込み』
- 252 ページの『putwc() ― ワイド文字の書き込み』
- 20 ページの『<wchar.h>』

qsort() ― 配列のソート

フォーマット

```
#include <stdlib.h>
void qsort(void *base, size_t num, size_t width,
           int(*compare)(const void *key, const void *element));
```

言語レベル: ANSI

スレッド・セーフ: はい。

説明

`qsort()` 関数は、*num* 個のエレメントの配列を、それぞれのサイズ *width* (バイト数) でソートします。*base* ポインターは、ソートする配列に対するポインターです。`qsort()` 関数は、ソートされたエレメントでこの配列を上書きします。

compare 引数は、ユーザーが指定する必要のある、関数を指すポインターです。この関数には *key* 引数、配列 *element* の順序でポインターを指定します。 `qsort()` 関数は、この関数を検索の途中で 1 回以上呼

び出します。この関数は、*key* と *element* を比較し、以下いずれかの値を戻します。

値	意味
0 より小さい値	*key* は *element* より小さい
0	*key* は *element* と等しい
0 より大きい値	*key* は *element* より大きい

値 **意味**

0 より小さい値
key は *element* より小さい

0 *key* は *element* と等しい

0 より大きい値
key は *element* より大きい

ソートされた配列エレメントは、*比較*関数によって定義されたように、昇順で保管されます。 *compare* において、「より大きい」と「より小さい」の意味を逆にすることによって、逆順でソートできます。比較により 2 つのエレメントが等しい場合、エレメントの順序は指定されません。

戻り値

戻り値はありません。

`qsort()` **の使用例**

この例では、提供されている比較関数 `compare()` を使用して、引数 (argv) を昇順の字句シーケンスでソートします。

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

               /* Declaration of compare() as a function */
int compare(const void *, const void *);

int main (int argc, char *argv[ ])
{
   int i;
   argv++;
   argc--;
   qsort((char *)argv, argc, sizeof(char *), compare);
   for (i = 0; i < argc; ++i)
      printf("%s¥n", argv[i]);
   return 0;
}

int compare (const void *arg1, const void *arg2)
{
               /* Compare all of both strings */
   return(strcmp(*(char **)arg1, *(char **)arg2));
}

/***********  If the program is passed the arguments:  ************
******** 'Does' 'this' 'really' 'sort' 'the' 'arguments' 'correctly?'****
****************  then the expected output is: *******************

arguments
correctly?
really
sort
the
this
Does
*/
```

関連情報

- 54 ページの『bsearch() ― 配列の検索』
- 18 ページの『<stdlib.h>』

QXXCHGDA() ― データ域の変更

フォーマット

```
#include <xxdtaa.h>

void QXXCHGDA(_DTAA_NAME_T dtaname, short int offset, short int len,
              char *dtaptr);
```

言語レベル: ILE C Extension

スレッド・セーフ: はい。

ジョブ CCSID インターフェース: この関数に送信される文字データは、すべてジョブの CCSID 内にあると想定されます。この関数によって戻された文字データは、すべてジョブの CCSID 内にあります。詳しくは、 549 ページの『CCSID およびロケールの理解』を参照してください。

説明

QXXCHGDA() 関数を使用すると、*dtaname* によって指定されたデータ域を変更できます。このデータ域は、*offset* の位置で始まり、ユーザー・バッファー内のデータは、長さ *len* の *dtaptr* によって指示されます。構造体 *dtaname* には、データ域とそのデータ域が含まれるライブラリーの名前が含まれています。データ域名に指定できる値は、以下のとおりです。

***LDA** ローカル・データ域の内容を変更することを指定します。ライブラリー名 *dtaa_lib* はブランクでなければなりません。

***GDA** グループ・データ域の内容を変更することを指定します。ライブラリー名 *dtaa_lib* はブランクでなければなりません。

data-area-name
データ域作成 (CRTDTAARA) CL コマンドを使用して作成されたデータ域の内容を変更することを指定します。ライブラリー名 *dtaa_lib* は、*LIBL、*CURLIB、またはデータ域 (*data-area-name*) が置かれているライブラリーの名前のいずれかでなければなりません。データ域は、変更中はロックされます。

QXXCHGDA は、文字データの変更にのみ使用できます。

QXXCHGDA() の使用例

```
#include <stdio.h>
#include <xxdtaa.h>

#define START   1
#define LENGTH  8

int main(void)
{
  char newdata[LENGTH] = "new data";

  /* The local data area will be changed              */
  _DTAA_NAME_T dtaname = {"*LDA      ", "          "};

  /* Use function to change the local data area.      */
  QXXCHGDA(dtaname,START,LENGTH,newdata);
  /* The first 8 characters in the local data area    */
  /* are: new data                                    */
}
```

関連情報

- 264 ページの『QXXRTVDA() ― データ域の検索』

QXXDTOP() ― double からパック 10 進数への変換

フォーマット

```
#include <xxcvt.h>
void QXXDTOP(unsigned char *pptr, int digits, int fraction,
             double value);
```

言語レベル: ILE C Extension

スレッド・セーフ: はい。

説明

QXXDTOP 関数は、*value* で指定された *double* 値をパック 10 進数に変換します。この場合、*digits* は合計桁数で、*fraction* は小数部の桁数です。結果は、*pptr* が指す配列に保管されます。

QXXDTOP() の使用例

```
#include <xxcvt.h>
#include <stdio.h>

int main(void)
{
  unsigned char pptr[10];
  int digits = 8, fraction = 6;
  double value = 3.141593;

  QXXDTOP(pptr, digits, fraction, value);
}
```

関連情報

- 『QXXDTOZ() ― double からゾーン 10 進数への変換』
- 261 ページの『QXXITOP() ― 整数からパック 10 進数への変換』
- 262 ページの『QXXITOZ() ― 整数からゾーン 10 進数への変換』
- 262 ページの『QXXPTOD() ― パック 10 進数から double への変換』
- 263 ページの『QXXPTOI() ― パック 10 進数から整数への変換』
- 265 ページの『QXXZTOD() ― ゾーン 10 進数から double への変換』
- 266 ページの『QXXZTOI() ― ゾーン 10 進数から整数への変換』

QXXDTOZ() ― double からゾーン 10 進数への変換

フォーマット

```
#include <xxcvt.h>
void QXXDTOZ(unsigned char *zptr, int digits, int fraction,
             double value);
```

言語レベル: ILE C Extension

スレッド・セーフ: はい。

説明

QXXDTOZ 関数は、*value* で指定された *double* 値をゾーン 10 進数に変換します。この場合、*digits* は合計桁数で、*fraction* は小数部の桁数です。結果は、*zptr* が指す配列に保管されます。

QXXDTOZ() の使用例

```
#include <xxcvt.h>
#include <stdio.h>

int main(void)
{
  unsigned char zptr[10];
  int digits = 8, fraction = 6;
  double value = 3.141593;

  QXXDTOZ(zptr, digits, fraction, value);
}                          /* Zoned value is : 03141593 */
```

関連情報

- 259 ページの『QXXDTOP() ― double からパック 10 進数への変換』
- 『QXXITOP() ― 整数からパック 10 進数への変換』
- 262 ページの『QXXITOZ() ― 整数からゾーン 10 進数への変換』
- 262 ページの『QXXPTOD() ― パック 10 進数から double への変換』
- 263 ページの『QXXPTOI() ― パック 10 進数から整数への変換』
- 265 ページの『QXXZTOD() ― ゾーン 10 進数から double への変換』
- 266 ページの『QXXZTOI() ― ゾーン 10 進数から整数への変換』

QXXITOP() ― 整数からパック 10 進数への変換

フォーマット

```
#include <xxcvt.h>
void QXXITOP(unsigned char *pptr, int digits, int fraction,
             int value);
```

言語レベル: ILE C Extension

スレッド・セーフ: はい。

説明

QXXITOP 関数は、*value* で指定された整数をパック 10 進数に変換します。この場合、*digits* は合計桁数で、*fraction* は小数部の桁数です。結果は、*pptr* が指す配列に保管されます。

QXXITOP() の使用例

```
#include <xxcvt.h>
#include <stdio.h>

int main(void)
{
  unsigned char pptr[10];
  int digits = 3, fraction = 0;
  int value = 116;

  QXXITOP(pptr, digits, fraction, value);
}
```

関連情報

- 259 ページの『QXXDTOP() ― double からパック 10 進数への変換』
- 260 ページの『QXXDTOZ() ― double からゾーン 10 進数への変換』

- 『QXXITOZ() ― 整数からゾーン 10 進数への変換』
- 『QXXPTOD() ― パック 10 進数から double への変換』
- 263 ページの『QXXPTOI() ― パック 10 進数から整数への変換』
- 265 ページの『QXXZTOD() ― ゾーン 10 進数から double への変換』
- 266 ページの『QXXZTOI() ― ゾーン 10 進数から整数への変換』

QXXITOZ() ― 整数からゾーン 10 進数への変換

フォーマット

```
#include <xxcvt.h>
void QXXITOZ(unsigned char *zptr, int digits, int fraction, int value);
```

言語レベル: ILE C Extension

スレッド・セーフ: はい。

説明

QXXITOZ 関数は、*value* で指定された整数をゾーン 10 進数に変換します。この場合、*digits* は合計桁数で、*fraction* は小数部の桁数です。結果は、*zptr* が指す配列に保管されます。

QXXITOZ() の使用例

```
#include <xxcvt.h>
#include <stdio.h>

int main(void)
{
  unsigned char zptr[10];
  int digits = 9, fraction = 0;
  int value = 111115;

  QXXITOZ(zptr, digits, fraction, value);
                        /* Zoned value is : 000111115 */
}
```

関連情報

- 259 ページの『QXXDTOP() ― double からパック 10 進数への変換』
- 260 ページの『QXXDTOZ() ― double からゾーン 10 進数への変換』
- 261 ページの『QXXITOP() ― 整数からパック 10 進数への変換』
- 『QXXPTOD() ― パック 10 進数から double への変換』
- 263 ページの『QXXPTOI() ― パック 10 進数から整数への変換』
- 265 ページの『QXXZTOD() ― ゾーン 10 進数から double への変換』
- 266 ページの『QXXZTOI() ― ゾーン 10 進数から整数への変換』

QXXPTOD() ― パック 10 進数から double への変換

フォーマット

```
#include <xxcvt.h>
double QXXPTOD(unsigned char *pptr, int digits, int fraction);
```

言語レベル: ILE C Extension

スレッド・セーフ: はい。

説明

QXXPTOD 関数は、パック 10 進数を *double* に変換します。

QXXPTOD() の使用例

```
#include <xxcvt.h>
#include <stdio.h>

int main(void)
{
  unsigned char pptr[10];
  int digits = 8, fraction = 6;
  double value = 6.123456, result;
          /* First convert an integer to a packed decimal,*/
  QXXDTOP(pptr, digits, fraction, value);
          /* then convert it back to a double.            */
  result = QXXPTOD(pptr, digits, fraction);
          /* result = 6.123456                            */
}
```

関連情報

- 259 ページの『QXXDTOP() ― double からパック 10 進数への変換』
- 260 ページの『QXXDTOZ() ― double からゾーン 10 進数への変換』
- 261 ページの『QXXITOP() ― 整数からパック 10 進数への変換』
- 262 ページの『QXXITOZ() ― 整数からゾーン 10 進数への変換』
- 『QXXPTOI() ― パック 10 進数から整数への変換』
- 265 ページの『QXXZTOD() ― ゾーン 10 進数から double への変換』
- 266 ページの『QXXZTOI() ― ゾーン 10 進数から整数への変換』

QXXPTOI() ― パック 10 進数から整数への変換

フォーマット

```
#include <xxcvt.h>
int QXXPTOI(unsigned char *pptr, int digits, int fraction);
```

言語レベル: ILE C Extension

スレッド・セーフ: はい。

説明

QXXPTOI 関数は、パック 10 進数を整数 に変換します。

QXXPTOI() の使用例

```
#include <xxcvt.h>
#include <stdio.h>

int main(void)
{
  unsigned char pptr[10];
  int digits = 3, fraction = 0, value = 104, result;
          /* First convert an integer to a packed decimal,*/
  QXXITOP(pptr, digits, fraction, value);
          /* then convert it back to an integer.          */
  result = QXXPTOI(pptr, digits, fraction);
          /* result = 104                                 */
}
```

関連情報

- 259 ページの『QXXDTOP() — double からパック 10 進数への変換』
- 260 ページの『QXXDTOZ() — double からゾーン 10 進数への変換』
- 261 ページの『QXXITOP() — 整数からパック 10 進数への変換』
- 262 ページの『QXXITOZ() — 整数からゾーン 10 進数への変換』
- 262 ページの『QXXPTOD() — パック 10 進数から double への変換』
- 265 ページの『QXXZTOD() — ゾーン 10 進数から double への変換』
- 266 ページの『QXXZTOI() — ゾーン 10 進数から整数への変換』

QXXRTVDA() — データ域の検索

フォーマット

```
#include <xxdtaa.h>

void QXXRTVDA(_DTAA_NAME_T dtaname, short int offset,
              short int len, char *dtaptr);
```

言語レベル: ILE C Extension

スレッド・セーフ: はい。

ジョブ CCSID インターフェース: この関数に送信される文字データは、すべてジョブの CCSID 内にあると想定されます。この関数によって戻された文字データは、すべてジョブの CCSID 内にあります。詳しくは、 549 ページの『CCSID およびロケールの理解』を参照してください。

説明

次の typedef 定義は、<xxdtaa.h> ヘッダー・ファイル内にあります。文字配列は非ヌル終了ストリングであるため、ブランクで埋める必要があります。

```
typedef struct _DTAA_NAME_T {
   char dtaa_name[10]; /* name of data area */
   char dtaa_lib[10];  /* library that contains data area */
}_DTAA_NAME_T;
```

QXXRTVDA() 関数は、*dtaname* で指定されたデータ域のコピーを検索します。このデータ域は、*offset* の位置で始まり、長さは *len* です。構造体 *dtaname* には、データ域とそのデータ域が含まれるライブラリーの名前が含まれています。データ域名に指定できる値は、以下のとおりです。

***LDA** ローカル・データ域の内容を検索します。ライブラリー名 *dtaa_lib* はブランクでなければなりません。

***GDA** グループ・データ域の内容を検索します。ライブラリー名 *dtaa_lib* はブランクでなければなりません。

***PDA** プログラム初期化パラメーター (PIP) データ域の内容を検索することを指定します。 PIP データ域は、開始前ジョブごとに作成され、長さが 2000 文字までの文字域です。 PIP データ域は、リクエスターを獲得するまで検索できません。ライブラリー名 *dtaa_lib* はブランクでなければなりません。

data-area-name
データ域作成 (CRTDTAARA) CL コマンドを使用して作成されたデータ域の内容を検索することを指定します。ライブラリー名 *dtaa_lib* は、*LIBL、*CURLIB、またはデータ域 (*data-area-name*) が置かれているライブラリーの名前のいずれかでなければなりません。データ域は、データの検索中はロックされます。

パラメーター *dtaptr* は、検索したデータ域のコピーを受け取るストレージを指すポインターです。 QXXRTVDA を使用して検索できるのは、文字データのみです。

QXXRTVDA() の使用例

```
#include <stdio.h>
#include <xxdtaa.h>

#define DATA_AREA_LENGTH  30
#define START             6
#define LENGTH            7

int main(void)
{
  char uda_area[DATA_AREA_LENGTH];

 /* Retrieve data from user-defined data area currently in MYLIB */
  _DTAA_NAME_T dtaname = {"USRDDA    ", "MYLIB     "};

 /* Use the function to retrieve some data into uda_area.        */
  QXXRTVDA(dtaname,START,LENGTH,uda_area);

 /* Print the contents of the retrieved subset.                  */
  printf("uda_area contains %7.7s¥n",uda_area);

}
```

関連情報

- 258 ページの『QXXCHGDA() — データ域の変更』

QXXZTOD() — ゾーン 10 進数から double への変換

フォーマット

```
#include <xxcvt.h>
double QXXZTOD(unsigned char *zptr, int digits, int fraction);
```

言語レベル: ILE C Extension

スレッド・セーフ: はい。

説明

QXXZTOD 関数は、*zptr* が指すゾーン 10 進数を *double* に変換します (*digits* は合計桁数で、 *fraction* は小数部の桁数)。その結果、*double* 値が戻されます。

QXXZTOD() の使用例

```
#include <xxcvt.h>
#include <stdio.h>

int main(void)
{
  unsigned char zptr[] = "06123456";
  int digits = 8, fraction = 6;
  double result;

  result = QXXZTOD(zptr, digits, fraction);
                /* result = 6.123456 */
}
```

関連情報

- 259 ページの『QXXDTOP() ― double からパック 10 進数への変換』
- 260 ページの『QXXDTOZ() ― double からゾーン 10 進数への変換』
- 261 ページの『QXXITOP() ― 整数からパック 10 進数への変換』
- 262 ページの『QXXITOZ() ― 整数からゾーン 10 進数への変換』
- 262 ページの『QXXPTOD() ― パック 10 進数から double への変換』
- 263 ページの『QXXPTOI() ― パック 10 進数から整数への変換』
- 『QXXZTOI() ― ゾーン 10 進数から整数への変換』

QXXZTOI() ― ゾーン 10 進数から整数への変換

フォーマット

```
#include <xxcvt.h>
int QXXZTOI(unsigned char *zptr, int digits, int fraction);
```

言語レベル: ILE C Extension

スレッド・セーフ: はい。

説明

QXXZTOI 関数は、*zptr* が指すゾーン 10 進数を整数 に変換します (*digits* は合計桁数で、 *fraction* は小数部の桁数)。その結果、整数が戻されます。

QXXZTOI() の使用例

```
#include <xxcvt.h>
#include <stdio.h>

int main(void)
{
  unsigned char zptr[] = "000111115";
  int digits = 9, fraction = 0, result;

  result = QXXZTOI(zptr, digits, fraction);
                      /* result = 111115 */
}
```

関連情報

- 259 ページの『QXXDTOP() — double からパック 10 進数への変換』
- 260 ページの『QXXDTOZ() — double からゾーン 10 進数への変換』
- 261 ページの『QXXITOP() — 整数からパック 10 進数への変換』
- 262 ページの『QXXITOZ() — 整数からゾーン 10 進数への変換』
- 262 ページの『QXXPTOD() — パック 10 進数から double への変換』
- 263 ページの『QXXPTOI() — パック 10 進数から整数への変換』
- 265 ページの『QXXZTOD() — ゾーン 10 進数から double への変換』

raise() — シグナルの送信

フォーマット

```
#include <signal.h>
int raise(int sig);
```

言語レベル: ANSI

スレッド・セーフ: はい。

説明

`raise()` 関数は、実行中のプログラムにシグナル *sig* を送ります。コンパイル・コマンドで SYSIFCOPT(*ASYNCSIGNAL) を使用してコンパイルされた場合、この関数は非同期シグナルを使用します。この関数の非同期バージョンは、プロセスまたはスレッドにシグナルをスローします。

戻り値

`raise()` 関数は、正常に実行された場合には 0、失敗した場合にはゼロ以外の値を戻します。

raise() の使用例

この例では、シグナル SIGUSR1 に対してシグナル・ハンドラー sig_hand を設定します。シグナル・ハンドラーは、シグナル SIGUSR1 が出されるたびに呼び出されますが、シグナルは、最初に出されてから 9 回までは無視されます。 10 回目に出されたシグナルで、プログラムはエラー・コード 10 で終了します。シグナル・ハンドラーは、呼び出すたびに再設定する必要があることに注意してください。

```
#include <signal.h>
#include <stdio.h>

void sig_hand(int);  /* declaration of sig_hand() as a function */

int main(void)
{
```

```
   signal(SIGUSR1, sig_hand); /* set up handler for SIGUSR1 */

   raise(SIGUSR1);   /* signal SIGUSR1 is raised */
                     /* sig_hand() is called     */
}

void sig_hand(int sig)
{
   static int count = 0;  /* initialized only once */

   count++;
   if (count == 10)  /* ignore the first 9 occurrences of this signal */
      exit(10);
   else
      signal(SIGUSR1, sig_hand);  /* set up the handler again */
}
/* This is a program fragment and not a complete program */
```

関連情報

- 362 ページの『signal() ― 割り込みシグナルの処理』
- 536 ページの『シグナル処理アクションの定義』
- 15 ページの『<signal.h>』
- i5/OS Information Center のトピック『API』内のシグナル API。
- i5/OS Information Center のトピック『API』内の POSIX スレッド API。

rand()、rand_r() ― 乱数の生成

フォーマット

```
#include <stdlib.h>
int rand(void);
int rand_r(unsigned int *seed);
```

言語レベル: ANSI

スレッド・セーフ: いいえ。 rand() はスレッド・セーフではありませんが、rand_r() はスレッド・セーフです。

説明

rand() 関数は、0 から RAND_MAX (<stdlib.h> で定義されるマクロ) までの範囲の疑似乱数整数を生成します。 rand() を呼び出す前に srand() 関数を使用して、乱数発生ルーチンの開始点を設定します。最初に srand() 関数を呼び出さない場合、デフォルトの seed は 1 です。

注: rand_r() 関数は、rand() の再始動可能なバージョンです。

戻り値

rand() 関数は疑似乱数を戻します。

rand() の使用例

この例では、生成された最初の 10 の乱数を出力します。

```
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
   int x;

   for (x = 1; x <= 10; x++)
      printf("iteration %d, rand=%d¥n", x, rand());
}

/*********************  Output should be similar to:  ************

iteration 1, rand=16838
iteration 2, rand=5758
iteration 3, rand=10113
iteration 4, rand=17515
iteration 5, rand=31051
iteration 6, rand=5627
iteration 7, rand=23010
iteration 8, rand=7419
iteration 9, rand=16212
iteration 10, rand=4086
*/
```

関連情報

- 370 ページの『srand() — rand() 関数の seed の設定』
- 18 ページの『<stdlib.h>』

_Racquire() — プログラム装置の獲得

フォーマット

```
#include <recio.h>
int _Racquire(_RFILE *fp, char *dev);
```

言語レベル: ILE C Extension

スレッド・セーフ: いいえ。

ジョブ CCSID インターフェース: この関数に送信される文字データは、すべてジョブの CCSID 内にあると想定されます。この関数によって戻された文字データは、すべてジョブの CCSID 内にあります。詳しくは、 549 ページの『CCSID およびロケールの理解』を参照してください。

説明

`_Racquire()` 関数は、*dev* パラメーターで指定されたプログラム装置を獲得し、それを *fp* で指定されたファイルに関連付けます。 *dev* パラメーターは、ヌル終了 C ストリングです。プログラム装置名は、大文字で指定してください。プログラム装置は、ファイルに定義する必要があります。

この関数は、ディスプレイおよび ICF ファイルの場合に有効です。

戻り値

`_Racquire()` 関数は、正常終了した場合には 1、正常終了しなかった場合には 0 を戻します。 errno の値は、EIOERROR (リカバリー不能な入出力エラーの発生) または EIORECERR (リカバリー可能な入出力エラーの発生) に設定されます。

errno の設定については、 531 ページの表 12 および 535 ページの表 14 を参照してください。

_Racquire() の使用例

```
#include <stdio.h>
#include <recio.h>
#include <string.h>
#include <stdlib.h>

int main(void)
{
    _RFILE     *fp;
    _RIOFB_T   *rfb;

    /* Open the device file.                                      */
    if (( fp = _Ropen ( "MYLIB/T1677RD2", "ar+" )) == NULL )
    {
        printf ( "Could not open file¥n" );
        exit ( 1 );
    }

    _Racquire ( fp,"DEVICE1" );    /* Acquire another program device. */
                                   /* Replace with actual device name.*/

    _Rformat ( fp,"FORMAT1" );     /* Set the record format for the   */
                                   /* display file.                   */

    rfb = _Rwrite ( fp, "", 0 );   /* Set up the display.             */

    /* Do some processing...                                      */


    _Rclose ( fp );
}
```

関連情報

- 328 ページの『_Rrelease() ― プログラム装置の解除』

_Rclose() ― ファイルのクローズ

フォーマット

```
#include <recio.h>

int _Rclose(_RFILE *fp);
```

言語レベル: ILE C Extension

スレッド・セーフ: はい。

説明

_Rclose() 関数は、*fp* で指定されたファイルをクローズします。このファイルをクローズする前に、ファイルに関連したすべてのバッファーがフラッシュされ、そのファイル用に予約されたすべてのバッファーが解放されます。ファイルは、例外が発生した場合でもクローズされます。 _Rclose() 関数は、すべてのタイプのファイルに適用されます。

| **注:** _RFILE ポインターが指すストレージは、_Rclose() 関数によって解放されます。_Rclose() 関数を使用した後は、_RFILE ポインターを使用しようとする試みはすべて無効になります。

戻り値

_Rclose() 関数は、ファイルが正常にクローズされた場合にはゼロを戻し、クローズ操作が失敗したり、ファイルがすでにクローズされている場合には EOF を戻します。ファイルは、例外が発生した場合でもクローズされ、ゼロが戻されます。

errno の値は、次のいずれかに設定されます。

値	意味
ENOTOPEN	ファイルはオープンされていません。
EIOERROR	リカバリー不能な入出力エラーが発生しました。
EIORECERR	リカバリー可能な入出力エラーが発生しました。

errno の設定については、 531 ページの表 12 および 535 ページの表 14 を参照してください。

_Rclose() の使用例

```
#include <stdio.h>
#include <stdlib.h>
#include <recio.h>

int main(void)
{
    _RFILE      *fp;

    /* Open the file for processing in arrival sequence.             */
    if (( fp = _Ropen ( "MYLIB/T1677RD1", "rr+, arrseq=Y" )) == NULL )
    {
        printf ( "Open failed¥n" );
        exit ( 1 );
    }
    else
    /* Do some processing */;


    _Rclose ( fp );
}
```

関連情報

- 301 ページの『_Ropen() ― レコード・ファイルをオープンして入出力操作を行う』

_Rcommit() ― 現行レコードのコミット

フォーマット

```
#include <recio.h>
int _Rcommit(char *cmtid);
```

言語レベル: ILE C Extension

スレッド・セーフ: いいえ。

ジョブ CCSID インターフェース: この関数に送信される文字データは、すべてジョブの CCSID 内にあると想定されます。この関数によって戻された文字データは、すべてジョブの CCSID 内にあります。詳しくは、 549 ページの『CCSID およびロケールの理解』を参照してください。

説明

_Rcommit() 関数は、それを呼び出すジョブの現行トランザクションを完了し、新規コミットメント境界を設定します。最後のコミットメント境界以降に加えられたすべての変更は、永続となります。ジョブ内のコミットメント制御の下でオープンしているファイルまたはリソースが影響を受けます。

cmtid パラメーターは、ヌル終了 C ストリングで、コミットメント境界に関連した一連の変更を識別するために使用されます。この長さは 4000 バイト未満でなければなりません。

_Rcommit() 関数は、データベースと DDM ファイルに適用されます。

戻り値

_Rcommit() 関数は、正常終了した場合には 1、正常終了しなかった場合にはゼロを戻します。 errno の値は、EIOERROR (リカバリー不能な入出力エラーの発生) または EIORECERR (リカバリー可能な入出力エラーの発生) に設定されます。

errno の設定については、 531 ページの表 12 および 535 ページの表 14 を参照してください。

_Rcommit() の使用例

```
#include <stdio.h>
#include <recio.h>
#include <stdlib.h>
#include <string.h>

int main(void)
{
  char       buf[40];
  int        rc = 1;
  _RFILE     *purf;
  _RFILE     *dailyf;

  /* Open purchase display file and daily transaction file          */
  if ( ( purf = _Ropen ( "MYLIB/T1677RD3", "ar+,indicators=y" )) == NULL )
  {
      printf ( "Display file did not open.¥n" );
      exit ( 1 );
  }

  if ( ( dailyf = _Ropen ( "MYLIB/T1677RDA", "wr,commit=y") ) == NULL )
  {
      printf ( "Daily transaction file did not open.¥n" );
      exit ( 2 );
  }

  /* Select purchase record format */
  _Rformat ( purf, "PURCHASE" );

  /* Invite user to enter a purchase transaction.                   */
  /* The _Rwrite function writes the purchase display.              */
  _Rwrite ( purf, "", 0 );
  _Readn ( purf, buf, sizeof(buf), __DFT );

  /* Update daily transaction file                                  */
  rc = (( _Rwrite ( dailyf, buf, sizeof(buf) ))->num_bytes );

  /* If the databases were updated, then commit the transaction.    */
  /* Otherwise, rollback the transaction and indicate to the        */
  /* user that an error has occurred and end the application.       */
  if ( rc )
    {
        _Rcommit ( "Transaction complete" );
    }
  else
    {
        _Rrollbck ( );
        _Rformat ( purf, "ERROR" );
    }

  _Rclose ( purf );
  _Rclose ( dailyf );
}
```

関連情報

- 331 ページの『_Rrollbck() ― コミットメント制御の変更のロールバック』

_Rdelete() ― レコードの削除

フォーマット

```
#include <recio.h>
_RIOFB_T *_Rdelete(_RFILE *fp);
```

言語レベル: ILE C Extension

スレッド・セーフ: はい。

説明

`_Rdelete()` 関数は、 *fp* で指定されたファイル内で更新のために現在ロックされているレコードを削除します。削除操作の後、レコードはロックされません。ファイルは更新のためにオープンしていなければなりません。

レコードは、読み取りまたは位置指定オプションで __NO_LOCK が指定されない場合、その読み取りまたは位置指定によって、更新のためにロックされます。レコードをロックした位置指定操作で __NO_POSITION オプションが指定された場合、削除されたレコードは、ファイルが現在位置指定されているレコードではない場合があります。

この関数は、データベースと DDM ファイルの場合に有効です。

戻り値

`_Rdelete()` 関数は、*fp* に関連した _RIOFB_T 構造体へのポインターを戻します。操作が正常終了した場合、num_bytes フィールドには 1 が含まれます。操作が正常終了しなかった場合、num_bytes フィールドにはゼロが含まれます。

errno の値は、次のいずれかに設定されます。

値　　意味

ENOTDLT
ファイルは削除操作用にオープンされていません。

EIOERROR
リカバリー不能な入出力エラーが発生しました。

EIORECERR
リカバリー可能な入出力エラーが発生しました。

errno の設定については、 531 ページの表 12 および 535 ページの表 14 を参照してください。

`_Rdelete()` の使用例

```
#include <stdio.h>
#include <stdlib.h>
#include <recio.h>

int main(void)
{
    _RFILE      *fp;
    _XXOPFB_T   *opfb;

    /* Open the file for processing in arrival sequence.              */
    if (( fp = _Ropen ( "MYLIB/T1677RD1", "rr+, arrseq=Y" )) == NULL )
    {
        printf ( "Open failed¥n" );
        exit ( 1 );
    }

    /* Get the library and file names of the file opened.             */
    opfb = _Ropnfbk ( fp );
    printf ( "Library: %10.10s¥nFile:    %10.10s¥n",
              opfb->library_name,
              opfb->file_name);

    /* Get the first record.                                           */
    _Rreadf ( fp, NULL, 20, __DFT );
    printf ( "First record:  %10.10s¥n", *(fp->in_buf) );

    /* Delete the first record.                                        */
    _Rdelete ( fp );


    _Rclose ( fp );
}
```

関連情報

- 329 ページの『_Rrlslck() ― レコード・ロックの解除』

_Rdevatr() ― 装置属性の取得

フォーマット

```
#include <recio.h>
#include <xxfdbk.h>
_XXDEV_ATR_T *_Rdevatr(_RFILE *fp, char *dev);
```

言語レベル: ILE C Extension

スレッド・セーフ: いいえ。

ジョブ CCSID インターフェース: この関数に送信される文字データは、すべてジョブの CCSID 内にあると想定されます。この関数によって戻された文字データは、すべてジョブの CCSID 内にあります。詳しくは、 549 ページの『CCSID およびロケールの理解』を参照してください。

説明

`_Rdevatr()` 関数は、 *dev* が指すデバイスと、*fp* で指定されたファイルが指す装置の装置属性フィードバック域のコピーへのポインターを戻します。

dev パラメーターは、ヌル終了 C ストリングです。装置名は、大文字で指定してください。

`_Rdevatr()` 関数は、ディスプレイ・ファイルと ICF ファイルの場合に有効です。

戻り値

_Rdevatr() 関数は、エラーが発生した場合には NULL を戻します。

errno の設定については、 531 ページの表 12 および 535 ページの表 14 を参照してください。

_Rdevatr() の使用例

```
#include <stdio.h>
#include <recio.h>
#include <string.h>
#include <stdlib.h>

int main(int argc, char ** argv)
{
    _RFILE   *fp; /* File pointer                                     */
    _RIOFB_T *rfb; /*Pointer to the file's feedback structure         */
    _XXIOFB_T *iofb; /* Pointer to the file's feedback area           */
    _XXDEV_ATR_T *dv_atr; /* Pointer to a copy of the file's device   */
                           /* attributes feedback area                */

    /* Open the device file.                                          */
    if (( fp = _Ropen ( "MYLIB/T1677RD2", "ar+" )) == NULL )
    {
        printf ( "Could not open file¥n" );
        exit ( 1 );
    }

    dv_atr = _Rdevatr (fp, argv[1]);
    if (dv_atr == NULL)
       printf("Error occurred getting device attributes for %s.¥n",
              argv[1]);

    _Rclose ( fp );
}
```

関連情報

- 269 ページの『_Racquire() ― プログラム装置の獲得』
- 328 ページの『_Rrelease() ― プログラム装置の解除』

realloc() ― 予約ストレージ・ブロック・サイズの変更

フォーマット

```
#include <stdlib.h>
void *realloc(void *ptr, size_t size);
```

言語レベル: ANSI

スレッド・セーフ: はい。

説明

realloc() 関数は、以前に予約されたストレージ・ブロックのサイズを変更します。 *ptr* 引数は、ブロックの先頭を指しています。*size* 引数は、ブロックの新規サイズ (バイト単位) を指定しています。ブロックの内容は、新旧サイズの短い方に達するまでは変更されません。

ptr が NULL の場合には、realloc() は *size* バイトのストレージのブロックを予約します。これによって、必ずしも各エレメントのすべてのビットが初期値 0 にされるわけではありません。

size が 0 で、*ptr* が NULL でない場合、 `realloc()` は *ptr* に割り当てられていたストレージを解放し、NULL を戻します。

注:

1. すべてのヒープ・ストレージは、呼び出しルーチンの活動化グループと関連付けられます。したがって、ストレージの割り振りと割り振り解除は同じ活動化グループ内で行ってください。 1 つの活動化グループ内でヒープ・ストレージを割り振ったり、異なる活動化グループからそのストレージを割り振り解除したりすることはできません。活動化グループについての詳細は、「*ILE 概念*」のマニュアルを参照してください。
2. 高速プール・メモリー・マネージャーが現行の活動化グループ内で使用可能にされている場合、ストレージは高速プール・メモリー・マネージャーを使用して検索されます。詳細については、 71 ページの『_C_Quickpool_Init() ― 高速プール・メモリー・マネージャーの初期化』を参照してください。

戻り値

`realloc()` 関数は、再度割り振られたストレージ・ブロックを指すポインターを戻します。ブロックの保管場所は、`realloc()` 関数で移動できます。したがって、`realloc()` 関数の *ptr* 引数は、必ずしも戻り値と同じである必要はありません。

size が 0 の場合、`realloc()` 関数は NULL を戻します。ストレージの大きさが十分でなく、指定されたサイズにブロックを拡張できない場合には、元のブロックは変更されず、`realloc()` 関数は NULL を戻します。

戻り値が指しているストレージは、不特定型のオブジェクトのストレージに位置合わせされます。

C ソース・コードを変更せずに、単一レベル・ストア・ストレージの代わりに**テラスペース**・ストレージを使用する場合、コンパイラー・コマンドで TERASPACE(*YES *TSIFC) パラメーターを指定します。これにより、`realloc()` ライブラリー関数が `_C_TS_realloc()` (テラスペース・ストレージでの `realloc()` ライブラリー関数のカウンター・パート) にマップされます。`_C_TS_realloc()` への各呼び出しにより割り振り可能なテラスペース・ストレージの最大量は、2GB - 240、または 214743408 バイトです。テラスペース・ストレージの追加情報については、「*ILE Concepts*」のマニュアルを参照してください。

`realloc()` の使用例

この例では、プロンプトで指示されたサイズの array に対してストレージを割り振り、 `realloc()` を使用して、その配列の新規サイズを保持するブロックを再度割り振ります。配列の内容は、各割り振りの後で出力されます。

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
  long * array;    /* start of the array */
  long * ptr;      /* pointer to array   */
  int    i;        /* index variable     */
  int  num1, num2; /* number of entries of the array */
  void print_array( long *ptr_array, int size);
  printf( "Enter the size of the array¥n" );
  scanf( "%i", &num1);
  /* allocate num1 entries using malloc() */
  if ( (array = (long *) malloc( num1 * sizeof( long ))) != NULL )
  {
     for ( ptr = array, i = 0; i < num1 ; ++i ) /* assign values */
        *ptr++ = i;
     print_array( array, num1 );
     printf("¥n");
  }
  else { /*  malloc error  */
    perror( "Out of storage" );
    abort();
  }
  /* Change the size of the array ... */
  printf( "Enter the size of the new array¥n" );
  scanf( "%i", &num2);
  if ( (array = (long *) realloc( array, num2* sizeof( long ))) != NULL )
  {
     for ( ptr = array + num1, i = num1; i <= num2; ++i )
        *ptr++ = i + 2000;  /* assign values to new elements */
     print_array( array, num2 );
  }
  else { /* realloc error */
    perror( "Out of storage" );
    abort();
  }
}

void print_array( long  * ptr_array, int size )
{
  int i;
  long * index = ptr_array;
  printf("The array of size %d is:¥n", size);
  for ( i = 0; i < size; ++i )           /* print the array out    */
    printf( "  array[ %i ] = %li¥n", i, ptr_array[i] );
}

/**** If the initial value entered is 2 and the second value entered
      is 4, then the expected output is:
Enter the size of the array
The array of size 2 is:
  array[ 0 ] = 0
  array[ 1 ] = 1
Enter the size of the new array
The array of size 4 is:
  array[ 0 ] = 0
  array[ 1 ] = 1
  array[ 2 ] = 2002
  array[ 3 ] = 2003                                                */
```

関連情報

- 58 ページの『calloc() ― ストレージの予約と初期化』
- 69 ページの『_C_Quickpool_Debug() ― 高速プール・メモリー・マネージャー特性の変更』
- 71 ページの『_C_Quickpool_Init() ― 高速プール・メモリー・マネージャーの初期化』

- 73 ページの『_C_Quickpool_Report() ― 高速プール・メモリー・マネージャー・レポートの生成』
- 564 ページの『ヒープ・メモリー』
- 134 ページの『free() ― ストレージ・ブロックの解放』
- 203 ページの『malloc() ― ストレージ・ブロックの予約』
- 18 ページの『<stdlib.h>』

regcomp() ― 正規表現のコンパイル

フォーマット

```
#include <regex.h>
int regcomp(regex_t *preg, const char *pattern, int cflags);
```

言語レベル: XPG4

スレッド・セーフ: はい。

ロケール依存: この関数の振る舞いは、現行ロケールの LC_CTYPE カテゴリーおよび LC_COLLATE カテゴリーの影響を受ける可能性があります。この関数は、コンパイル・コマンドに対して LOCALETYPE(*CLD) が指定されている場合には使用できません。詳細については、 549 ページの『CCSID およびロケールの理解』を参照してください。

説明

`regcomp()` 関数は、*pattern* が指すソースの正規表現を実行可能なバージョンにコンパイルし、それを *preg* が指すロケーションに保管します。その後で、`regexec()` 関数を使用して、正規表現をその他のストリングと比較することができます。

cflags フラグは、次のコンパイル・プロセスの属性を定義します。

cflag	ストリングの説明
REG_ALT_NL	• LOCALETYPE(*LOCALE) が指定されている場合は、統合ファイル・システムの改行文字が正規表現と一致します。 • LOCALETYPE(*LOCALEUTF) が指定されている場合は、データベースの改行文字が一致します。 REG_ALT_NL フラグが設定されていない場合、LOCALETYPE(*LOCALE) のデフォルトがデータベースの改行に一致し、 LOCALETYPE(*LOCALEUTF) のデフォルトが統合ファイル・システムの改行に一致します。 **注:** UTF-8 および UTF-32 の場合、統合ファイル・システムの改行文字とデータベースの改行文字は同じです。
REG_EXTENDED	拡張正規表現をサポートします。
REG_NEWLINE	改行文字を特別な行末文字として扱います。次に、] および $ パターンと一致する行の境界を設定し、明示的に ¥n を使用するストリング内でのみ一致します。 (このフラグを省略した場合、改行文字はその他の文字と同様に扱われます。)
REG_ICASE	照合で大/小文字を区別しません。

cflag	ストリングの説明
REG_NOSUB	*pattern* に指定された副次式の数を無視します。ストリングをコンパイル済みパターンと比較する場合 (`regexec()` を使用) は、ストリングがパターン全体に一致しなければなりません。次に、`regexec()` 関数は、一致が見つかったかどうかのみを示す値を戻します。この値は、ストリング内のどのポイントから一致が開始するか、あるいは一致するストリングが何であるかは示しません。

正規表現は、コンテキストから独立した構文で、幅広い文字セットと文字セットの配列を表します。これは、現在のロケールによって、さまざまに解釈されます。関数 `regcomp()`、`regerror()`、`regexec()`、および `regfree()` は、 UNIX® awk、ed、grep、および egrep の各コマンドと同様な方法で正規表現を使用します。

戻り値

`regcomp()` 関数は、正常終了した場合は 0 を戻します。それ以外の場合はエラー・コードを戻します (このエラー・コードは `regerror()` 関数への呼び出しで使用できます)。また、*preg* の内容は予期できません。

`regcomp()` の使用例

```
#include <regex.h>
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
   regex_t    preg;
   char       *string = "a very simple simple simple string";
   char       *pattern = "¥¥(sim[a-z]le¥¥) ¥¥1";
   int        rc;
   size_t     nmatch = 2;
   regmatch_t pmatch[2];

   if (0 != (rc = regcomp(&preg, pattern, 0))) {
      printf("regcomp() failed, returning nonzero (%d)¥n", rc);
      exit(EXIT_FAILURE);
   }

   if (0 != (rc = regexec(&preg, string, nmatch, pmatch, 0))) {
      printf("Failed to match '%s' with '%s',returning %d.¥n",
      string, pattern, rc);
   }
   else {
      printf("With the whole expression, "
             "a matched substring ¥"%.*s¥" is found at position %d to %d.¥n",
             pmatch[0].rm_eo - pmatch[0].rm_so, &string[pmatch[0].rm_so],
             pmatch[0].rm_so, pmatch[0].rm_eo - 1);
      printf("With the sub-expression, "
             "a matched substring ¥"%.*s¥" is found at position %d to %d.¥n",
             pmatch[1].rm_eo - pmatch[1].rm_so, string[pmatch[1].rm_so],
             pmatch[1].rm_so, pmatch[1].rm_eo - 1);
   }
   regfree(&preg);
   return 0;

   /****************************************************************************
      The output should be similar to :

      With the whole expression, a matched substring "simple simple" is found
      at position 7 to 19.
      With the sub-expression, a matched substring "simple" is found
      at position 7 to 12.
   ****************************************************************************/
}
```

関連情報

- 『regerror() ― 正規表現のエラー・メッセージの戻し』
- 283 ページの『regexec() ― コンパイル済み正規表現の実行』
- 285 ページの『regfree() ― 正規表現のメモリーの解放』
- 13 ページの『<regex.h>』

regerror() ― 正規表現のエラー・メッセージの戻し

フォーマット

```
#include <regex.h>
size_t regerror(int errcode,  const regex_t *preg,
               char *errbuf, size_t errbuf_size);
```

言語レベル: XPG4

スレッド・セーフ: はい。

ロケール依存: この関数の振る舞いは、現行ロケールの LC_CTYPE カテゴリーおよび LC_COLLATE カテゴリーの影響を受ける可能性があります。この関数は、コンパイル・コマンドに対して LOCALETYPE(*CLD) が指定されている場合には使用できません。詳細については、 549 ページの『CCSID およびロケールの理解』を参照してください。

説明

`regerror()` 関数は、正規表現 *preg* のエラー・コード *errcode* の説明を検索します。 *errcode* の説明は、 *errbuf* に割り当てられます。 *errbuf_size* 値は、保管できる最大メッセージ・サイズ (errbuf のサイズ) を指定します。 *errcode* の説明ストリングは、以下のとおりです。

エラー・コード	**ストリングの説明**
REG_NOMATCH	`regexec()` は一致を見つけられませんでした。
REG_BADPAT	正規表現が無効です。
REG_ECOLLATE	無効な照合エレメントを参照しました。
REG_ECTYPE	無効な文字クラス・タイプを参照しました。
REG_EESCAPE	正規表現の最後の文字が **¥** です。
REG_ESUBREG	`¥digit` 内の数値が無効、またはエラーです。
REG_EBRACK	**[]** が不揃いです。
REG_EPAREN	**¥(¥)** または **()** が不揃いです。
REG_EBRACE	**¥{ ¥}** が不揃いです。
REG_BADBR	**¥{** と **¥}** の間の式が無効です。
REG_ERANGE	範囲式内のエンドポイントが無効です。
REG_ESPACE	メモリー不足です。
REG_BADRPT	**?**、*****、または **+** の前に有効な正規表現がありません。
REG_ECHAR	マルチバイト文字が無効です。
REG_EBOL	**^** アンカーが正規表現の先頭にありません。
REG_EEOL	**$** アンカーが正規表現の末尾にありません。
REG_ECOMP	`regcomp()` 呼び出し中に不明なエラーが発生しました。
REG_EEXEC	`regexec()` 呼び出し中に不明なエラーが発生しました。

戻り値

`regerror()` は、エラー条件を説明するストリングを保持するために必要なバッファーのサイズを戻します。 `errno` の値は **ECONVERT** (変換エラー) に設定される可能性があります。

`regerror()` の使用例

この例では、無効な正規表現をコンパイルし、`regerror()` 関数を使用してエラー・メッセージを出力します。

```
#include <regex.htm>
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
   regex_t preg;
   char    *pattern = "a[missing.bracket";
   int     rc;
   char    buffer[100];

   if (0 != (rc = regcomp(&preg, pattern, REG_EXTENDED))) {
      regerror(rc, &preg, buffer, 100);
      printf("regcomp() failed with '%s'¥n", buffer);
      exit(EXIT_FAILURE);
   }
   return 0;

/**********************************************************
      The output should be similar to:

      regcomp() failed with '[] imbalance.'
   **********************************************************/
}
```

関連情報

- 279 ページの『regcomp() ― 正規表現のコンパイル』
- 『regexec() ― コンパイル済み正規表現の実行』
- 285 ページの『regfree() ― 正規表現のメモリーの解放』
- 13 ページの『<regex.h>』

regexec() ― コンパイル済み正規表現の実行

フォーマット

```
#include <regex.h>
int regexec(const regex_t *preg, const char *string,
            size_t nmatch, regmatch_t *pmatch, int eflags);
```

言語レベル: XPG4

スレッド・セーフ: はい。

ロケール依存: この関数の振る舞いは、現行ロケールの LC_CTYPE カテゴリーおよび LC_COLLATE カテゴリーの影響を受ける可能性があります。この関数は、コンパイル・コマンドに対して LOCALETYPE(*CLD) が指定されている場合には使用できません。詳細については、 549 ページの『CCSID およびロケールの理解』を参照してください。

説明

`regexec()` 関数は、ヌル終了の *string* とコンパイル済み正規表現 *preg* を比較して、両者の間に一致を見つけます。

nmatch 値は、 `regexec()` 関数が *preg* 内の副次式とのマッチングを試みる必要のある *string* 内のサブストリングの値です。 *pmatch* に指定する配列には、少なくとも *nmatch* 個のエレメントがなければなりません。

regexec() 関数によって、配列 *pmatch* のエレメントが *string* のサブストリングのオフセットで埋められますが、これは、*preg* を作成するために regcomp() 関数に対して指定された元のパターンの括弧で囲まれた副次式に対応します。配列のゼロ番目のエレメントは、パターン全体に対応しています。 *nmatch* より多い副次式がある場合は、最初の *nmatch - 1* 個だけが保管されます。 *nmatch* が 0 である場合、または regcomp() 関数によって *preg* の作成時に REG_NOSUB フラグが設定された場合、 regexec() 関数は *pmatch* 引数を無視します。

eflag *flag* は、regexec() 関数のカスタマイズ可能な振る舞いを定義します。

errflag	**ストリングの説明**
REG_NOTBOL	*string* の先頭文字が行の始まりではないことを示します。
REG_NOTEOL	*string* の先頭文字が行の終わりではないことを示します。

基本正規表現、または拡張正規表現とのマッチングの場合には、元のパターンの指定された括弧で囲まれた副次式のいずれかが、 *string* の複数の異なるサブストリングと一致する候補となる可能性があります。次の規則に従って、*pmatch* で報告されるサブストリングが判別されます。

1. 正規表現内の副次式 *i* が別の副次式に含まれていない場合に、それが複数回一致する候補となったときは、その最後の一致が *pmatch[i]* 内のバイト・オフセットで区切られます。
2. 副次式 *i* が別の副次式に含まれていない場合に、それが他の方法で正常に実行された一致の候補とならなかったときは、 *pmatch[i]* 内のバイト・オフセットは -1 になります。以下の条件のいずれかに当てはまる場合、副次式は、一致の候補とはなりません。
   - 基本正規表現内の副次式の直後に ***** または **¥{ ¥}** が表示される。
   - 拡張正規表現内の副次式の直後に *****、**?**、または **{ }** が表示され、副次式が一致しなかった (一致回数 0 回)。
   - この副次式または別の副次式を選択するために、拡張正規表現内に **|** が使用されており、別の副次式が一致している。
3. 副次式 *i* が別の副次式 *j* に含まれており、 *i* が *j* 内にある他のいずれの副次式にも含まれていない場合に、副次式 *j* の一致が *pmatch[j]* で報告されているときは、 *pmatch[i]* で報告されている副次式 *i* の一致または不一致は、前述の 1. と 2. のようになります。ただし、これは、ストリング全体ではなく、*pmatch[j]* で報告されているサブストリング内の場合です。
4. 副次式 *i* が副次式 *j* に含まれている場合に、*pmatch[j]* 内のバイト・オフセットが -1 であるときは、*pmatch[i]* 内のオフセットも -1 になります。
5. 副次式 *i* がゼロ長ストリングに一致した場合、 *pmatch[i]* 内のバイト・オフセットは、ともにゼロ長ストリングの直後に続く文字またはヌル終止符のバイト・オフセットになります。

regcomp() 関数によって *preg* の作成時に REG_NOSUB フラグが設定された場合、 *pmatch* の内容は未指定です。 *preg* の作成時に REG_NEWLINE フラグが設定された場合、ストリングには改行文字が許可されます。

戻り値

一致が見つかった場合、regexec() 関数は 0 を戻します。一致が見つからなかった場合、regexec() 関数は REG_NOMATCH を戻します。それ以外の場合には、エラーを示すゼロ以外の値を戻します。ゼロ以外の戻り値は、regerror() 関数への呼び出しに使用できます。

regexec() の使用例

```
#include <regex.h>
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
   regex_t    preg;
   char       *string = "a very simple simple simple string";
   char       *pattern = "¥¥(sim[a-z]le¥¥) ¥¥1";
   int        rc;
   size_t     nmatch = 2;
   regmatch_t pmatch[2];

   if (0 != (rc = regcomp(&preg, pattern, 0))) {
      printf("regcomp() failed, returning nonzero (%d)¥n", rc);
      exit(EXIT_FAILURE);
   }

   if (0 != (rc = regexec(&preg, string, nmatch, pmatch, 0))) {
      printf("Failed to match '%s' with '%s',returning %d.¥n",
      string, pattern, rc);
   }
   else {
      printf("With the whole expression, "
             "a matched substring ¥"%.*s¥" is found at position %d to %d.¥n",
             pmatch[0].rm_eo - pmatch[0].rm_so, &string[pmatch[0].rm_so],
             pmatch[0].rm_so, pmatch[0].rm_eo - 1);
      printf("With the sub-expression, "
             "a matched substring ¥"%.*s¥" is found at position %d to %d.¥n",
             pmatch[1].rm_eo - pmatch[1].rm_so, &string[pmatch[1].rm_so],
             pmatch[1].rm_so, pmatch[1].rm_eo - 1);
   }
   regfree(&preg);
   return 0;

/****************************************************************************
      The output should be similar to :

      With the whole expression, a matched substring "simple simple" is found
      at position 7 to 19.
      With the sub-expression, a matched substring "simple" is found
      at position 7 to 12.
   ****************************************************************************/
}
```

関連情報

- 279 ページの『regcomp() — 正規表現のコンパイル』
- 281 ページの『regerror() — 正規表現のエラー・メッセージの戻し』
- 『regfree() — 正規表現のメモリーの解放』
- 13 ページの『<regex.h>』

regfree() — 正規表現のメモリーの解放

フォーマット

```
#include <regex.h>
void regfree(regex_t *preg);
```

言語レベル: XPG4

スレッド・セーフ: はい。

ロケール依存: この関数の振る舞いは、現行ロケールの LC_CTYPE カテゴリーおよび LC_COLLATE カテゴリーの影響を受ける可能性があります。この関数は、コンパイル・コマンドに対して LOCALETYPE(*CLD) が指定されている場合には使用できません。詳細については、 549 ページの『CCSID およびロケールの理解』を参照してください。

説明

`regfree()` 関数は、正規表現 *preg* を実装するために `regcomp()` 関数によって割り振られたメモリーを解放します。 `regfree()` 関数への呼び出しの後、 *preg* によって定義された式は、以後コンパイル済みの正規表現または拡張式ではなくなります。

戻り値

戻り値はありません。

`regfree()` の使用例

この例では、拡張正規表現をコンパイルします。

```
#include <regex.h>
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
   regex_t preg;
   char    *pattern = ".*(simple).*";
   int     rc;

   if (0 != (rc = regcomp(&preg, pattern, REG_EXTENDED))) {
      printf("regcomp() failed, returning nonzero (%d)¥n", rc);
      exit(EXIT_FAILURE);
   }

   regfree(&preg);
   printf("regcomp() is successful.¥n");
   return 0;

/************************************************************
      The output should be similar to:

      regcomp() is successful.
   ************************************************************/
}
```

関連情報

- 279 ページの『regcomp() ― 正規表現のコンパイル』
- 281 ページの『regerror() ― 正規表現のエラー・メッセージの戻し』
- 283 ページの『regexec() ― コンパイル済み正規表現の実行』
- 13 ページの『<regex.h>』

remove() ― ファイルの削除

フォーマット

```
#include <stdio.h>
int remove(const char *filename);
```

言語レベル: ANSI

スレッド・セーフ: はい。

説明

`remove()` 関数は、*filename* で指定されたファイルを削除します。ファイル名にメンバー名が含まれている場合は、そのメンバーが除去されるか、ファイルが削除されます。

注: 存在しないファイルやオープンしているファイルは除去できません。

戻り値

`remove()` 関数は、ファイルを正常に削除した場合には 0 を戻します。ゼロ以外の戻り値はエラーを示します。

`errno` の値は **ECONVERT** (変換エラー) に設定される可能性があります。

`remove()` の使用例

この例をファイル名で呼び出すと、プログラムはそのファイルを除去しようとします。プログラムは、エラーが発生した場合にメッセージを表示します。

```
#include <stdio.h>

int main(int argc, char ** argv)
{
  if ( argc != 2 )
    printf( "Usage: %s fn¥n", argv[0] );
  else
    if ( remove( argv[1] ) != 0 )
      perror( "Could not remove file" );
}
```

関連情報

- 114 ページの『fopen() ー ファイルのオープン』
- 『rename() ー ファイルの名前変更』
- 17 ページの『<stdio.h>』

rename() ー ファイルの名前変更

フォーマット

```
#include <stdio.h>
int rename(const char *oldname, const char *newname);
```

言語レベル: ANSI

スレッド・セーフ: はい。

説明

rename() 関数は、*oldname* で指定されたファイルの名前を、 *newname* で指定された名前に変更します。 *oldname* ポインターは、既存のファイルの名前を指している必要があります。 *newname* ポインターは、既存のファイル名を指定する必要はありません。ファイルは、既存のファイルの名前に名前変更することはできません。オープン・ファイルを名前変更することもできません。

新規名に使用できるファイル・フォーマットは、以前の名前のフォーマットによって異なります。次の表に、ファイルの旧名を指定するために使用できる有効なファイル・フォーマットと、対応する新規名の有効なファイル・フォーマットを示します。

新規名と旧名の両方のフォーマットが lib/file(member) である場合、ファイルは変更できません。ファイル名が変更された場合、名前変更は機能しません。例えば、次の名前は無効です。 lib/file1(member1) lib/file2(member1)。

旧名	新規名
lib/file(member)	lib/file(member)、lib/file、file、file(member)
lib/file	lib/file、file
ファイル	lib/file、file
file(member)	lib/file(member)、lib/file、file、file(member)

戻り値

rename() 関数は、正常に実行された場合には 0 を戻します。エラーの場合は、ゼロ以外の値を戻します。

errno の値は **ECONVERT** (変換エラー) に設定される可能性があります。

rename() の使用例

この例では、2 つのファイル名を入力として使用し、 rename() を使用して、ファイル名を最初の名前から 2 番目の名前に変更します。

```
#include <stdio.h>

int main(int argc, char ** argv )
{
  if ( argc != 3 )
    printf( "Usage: %s old_fn new_fn¥n", argv[0] );
  else if ( rename( argv[1], argv[2] ) != 0 )
    perror ( "Could not rename file" );
}
```

関連情報

- 114 ページの『fopen() ― ファイルのオープン』
- 286 ページの『remove() ― ファイルの削除』
- 17 ページの『<stdio.h>』

rewind() ― 現在のファイル位置の調整

フォーマット

```
#include <stdio.h>
void rewind(FILE *stream);
```

言語レベル: ANSI

スレッド・セーフ: はい。

説明

rewind() 関数は、*stream* に関連したファイル・ポインターの位置をファイルの先頭に変更します。
rewind() 関数への呼び出しは、以下の場合と同じです。

```
(void)fseek(stream, 0L, SEEK_SET);
```

ただし、rewind() 関数は *stream* のエラー標識もクリアします。

rewind() 関数は、type=record を指定してオープンしたファイルについてはサポートされません。

戻り値

戻り値はありません。

errno の値は、次のいずれかに設定されます。

値　　意味

EBADF
ファイル・ポインター、または記述子が有効ではありません。

ENODEV
誤ったデバイスに対して、操作が試行されました。

EIOERROR
リカバリー不能な入出力エラーが発生しました。

EIORECERR
リカバリー可能な入出力エラーが発生しました。

rewind() の使用例

この例では、まず入力と出力用のファイル myfile をオープンします。この例では、ファイルに整数を書き込み、 rewind() を使用してファイル・ポインターの位置をファイルの先頭に変更してから、データを読み取ります。

```
#include <stdio.h>

FILE *stream;

int data1, data2, data3, data4;
int main(void)
{
   data1 = 1; data2 = -37;

      /* Place data in the file */
   stream = fopen("mylib/myfile", "w+");
   fprintf(stream, "%d %d¥n", data1, data2);

      /* Now read the data file */
   rewind(stream);
   fscanf(stream, "%d", &data3);
   fscanf(stream, "%d", &data4);
   printf("The values read back in are: %d and %d¥n",
       data3, data4);
}

/********************  Output should be similar to:  **************

The values read back in are: 1 and -37
*/
```

関連情報

- 104 ページの『fgetpos() ― ファイル位置の取得』
- 140 ページの『fseek() ― fseeko() ― ファイル位置の位置変更』
- 142 ページの『fsetpos() ― ファイル位置の設定』
- 144 ページの『ftell() ― ftello() ― 現在位置の取得』
- 17 ページの『<stdio.h>』

_Rfeod() ― データの終わりの強制

フォーマット

```
#include <recio.h>

int _Rfeod(_RFILE *fp);
```

言語レベル: ILE C Extension

スレッド・セーフ: はい。

説明

`_Rfeod()` 関数は、*fp* で指定されたファイルに関連した装置またはメンバーのデータの終わりの条件を強制します。システムがバッファリング中の未解決の更新、削除または書き込みは、不揮発性ストレージに強制されます。データベース・ファイルが入力用にオープンしている場合、未解決のロックは解除されます。

ファイルが複数メンバー処理用にオープンしておらず、現在のメンバーがファイル内の最後のメンバーでない場合、 `_Rfeod()` 関数は、ファイルを *END に位置指定します。複数メンバー処理が有効で、現在のメンバーがファイル内の最後のメンバーでない場合、 `_Rfeod()` は、ファイルの次のメンバーをオープンし、それを *START に位置指定します。

`_Rfeod()` 関数は、すべてのタイプのファイルの場合に有効です。

戻り値

_Rfeod() 関数は、複数メンバー処理が実行され、次のメンバーがオープンされた場合に 1 を戻します。ファイルが *END に位置指定された場合には、EOF が戻されます。指定の操作が正常に実行されなかった場合は、ゼロが戻されます。 errno の値は、EIOERROR (リカバリー不能なエラーの発生) または EIORECERR (リカバリー可能な入出力エラーの発生) に設定されます。 errno の設定については、531 ページの表 12 および 535 ページの表 14 を参照してください。

_Rfeod() の使用例

```
#include <stdio.h>
#include <stdlib.h>
#include <recio.h>

int main(void)
{
    _RFILE   *in;
    char     new_purchase[21] = "PEAR      1002022244";

    /* Open the file for processing in keyed sequence.             */

    if ( (in = _Ropen("MYLIB/T1677RD4", "rr+, arrseq=N")) == NULL )
    {
        printf("Open failed¥n");
        exit(1);
    };

    /* Update the first record in the keyed sequence.              */

    _Rlocate(in, NULL, 0, __FIRST);
    _Rupdate(in, new_purchase, 20);

    /* Force the end of data.                                       */

    _Rfeod(in);
```

関連情報

- 269 ページの『_Racquire() — プログラム装置の獲得』
- 『_Rfeov() — ファイルの終わりの強制』

_Rfeov() — ファイルの終わりの強制

フォーマット

```
#include <recio.h>

int _Rfeov(_RFILE *fp);
```

言語レベル: ILE C Extension

スレッド・セーフ: はい。

説明

_Rfeov() 関数は、*fp* で指定されたファイルに関連したテープ・ファイルのボリュームの終わりの状態を強制します。 _Rfeov() 関数は、ファイルをファイルの次のボリュームに位置指定します。ファイルが出力用にオープンしている場合は、出力バッファーがフラッシュされます。

_Rfeov() 関数は、テープ・ファイルの場合に有効です。

戻り値

_Rfeov() 関数は、ファイルがあるボリュームから次のボリュームに移動された場合、1 を戻します。ファイルの最後のボリュームの処理中に呼び出された場合は EOF を戻します。操作が正常に実行されなかった場合はゼロを戻します。 errno の値は、EIOERROR (リカバリー不能なエラーの発生) または EIORECERR (リカバリー可能な入出力エラーの発生) に設定されます。 errno の設定については、 531 ページの表 12 および 535 ページの表 14 を参照してください。

_Rfeov() の使用例

```
#include <stdio.h>
#include <recio.h>
#include <stdlib.h>

int main(void)
{
    _RFILE *tape;
    _RFILE *fp;
    char   buf[92];
    int    i, feov2;

    /* Open source physical file containing C source.           */

    if (( fp = _Ropen ( "QCSRC(T1677SRC)", "rr blkrcd=y" )) == NULL )
    {
        printf ( "could not open C source file¥n" );
        exit ( 1 );
    }

    /* Open tape file to receive C source statements            */

    if (( tape = _Ropen ( "T1677TPF", "wr lrecl=92 blkrcd=y" )) == NULL )
    {
        printf ( "could not open tape file¥n" );
        exit ( 2 );
    }

    /* Read the C source statements, find their sizes           */
    /* and add them to the tape file.                           */

    while (( _Rreadn ( fp, buf, sizeof(buf), __DFT )) -> num_bytes != EOF
)
    {
        for ( i = sizeof(buf) - 1 ; buf[i] == ' ' && i > 12;       --i );
        i = (i == 12) ? 80 : (1-12);
        memmove( buf, buf+12, i );
        _Rwrite ( tape, buf, i );
    }
    feov2 = _Rfeov (fp);

    _Rclose ( fp );
    _Rclose ( tape );
}
```

関連情報

- 269 ページの『_Racquire() ― プログラム装置の獲得』
- 290 ページの『_Rfeod() ― データの終わりの強制』

_Rformat() ― レコード・フォーマット名の設定

フォーマット

```
#include <recio.h>

void  _Rformat(_RFILE *fp, char *fmt);
```

言語レベル: ILE C Extension

スレッド・セーフ: はい。

ジョブ CCSID インターフェース: この関数に送信される文字データは、すべてジョブの CCSID 内にあると想定されます。この関数によって戻された文字データは、すべてジョブの CCSID 内にあります。詳細については、 549 ページの『CCSID およびロケールの理解』を参照してください。

説明

`_Rformat()` 関数は、 *fp* で指定されたファイルのレコード・フォーマットを *fmt* に設定します。

fmt パラメーターは、ヌル終了 C ストリングです。 *fmt* パラメーターは大文字で指定してください。

`_Rformat()` 関数は、複数フォーマットの論理データベース、DDM ファイル、ディスプレイ、ICF およびプリンター・ファイルの場合に有効です。

戻り値

`_Rformat()` 関数はボイドを戻します。 errno の設定については、 531 ページの表 12 および 535 ページの表 14 を参照してください。

`_Rformat()` の使用例

この例では、`_Rformat()` の使用法を示します。

```
#include <stdio.h>
#include <recio.h>
#include <stdlib.h>
#include <string.h>

int main(void)
{
  char       buf[40];
  int        rc = 1;
  _RFILE     *purf;
  _RFILE     *dailyf;

  /* Open purchase display file and daily transaction file          */
  if ( ( purf = _Ropen ( "MYLIB/T1677RD3", "ar+,indicators=y" )) == NULL )
  {
      printf ( "Display file did not open.¥n" );
      exit ( 1 );
  }

  if ( ( dailyf = _Ropen ( "MYLIB/T1677RDA", "wr,commit=y") ) == NULL )
  {
      printf ( "Daily transaction file did not open.¥n" );
      exit ( 2 );
  }

  /* Select purchase record format */
  _Rformat ( purf, "PURCHASE" );

  /* Invite user to enter a purchase transaction.                   */
  /* The _Rwrite function writes the purchase display.              */
  _Rwrite ( purf, "", 0 );
  _Rreadn ( purf, buf, sizeof(buf), __DFT );

  /* Update daily transaction file                                  */
  rc = (( _Rwrite ( dailyf, buf, sizeof(buf) ))->num_bytes );

  /* If the databases were updated, then commit the transaction.    */
  /* Otherwise, rollback the transaction and indicate to the        */
  /* user that an error has occurred and end the application.       */
  if ( rc )
    {
        _Rcommit ( "Transaction complete" );
    }
  else
    {
        _Rrollbck ( );
        _Rformat ( purf, "ERROR" );
    }

  _Rclose ( purf );
  _Rclose ( dailyf );
}
```

関連情報

- 301 ページの『_Ropen() ― レコード・ファイルをオープンして入出力操作を行う』

_Rindara() ― 分離標識域の設定

フォーマット

```
#include <recio.h>

void _Rindara(_RFILE *fp, char *indic_buf);
```

言語レベル: ILE C Extension

スレッド・セーフ: いいえ。

ジョブ CCSID インターフェース: この関数に送信される文字データは、すべてジョブの CCSID 内にあると想定されます。この関数によって戻された文字データは、すべてジョブの CCSID 内にあります。詳細については、 549 ページの『CCSID およびロケールの理解』を参照してください。

説明

`_Rindara()` 関数は、*indic_buf* を、 *fp* で指定されたファイルで使用する分離標識域として登録します。ファイルは、`_Ropen()` 関数にキーワード indicators=Y を指定してオープンする必要があります。ファイルの DDS は、分離標識域を使用することを指定することも必要です。通常は、各バイト内を「0」(文字) にして、分離標識域を明示的に初期化することが最善の方法です。

`_Rindara()` 関数は、ディスプレイ、ICF、およびプリンター・ファイルの場合に有効です。

戻り値

`_Rindara()` 関数はボイドを戻します。 errno の設定については、 531 ページの表 12 および 535 ページの表 14 を参照してください。

`_Rindara()` の使用例

```
#include <stdio.h>
#include <recio.h>
#include <stdlib.h>
#include <string.h>
#define  PF03     2
#define  IND_OFF '0'
#define  IND_ON  '1'

int main(void)
{
  char       buf[40];
  int        rc = 1;
  _SYSindara ind_area;
  _RFILE     *purf;
  _RFILE     *dailyf;
  /* Open purchase display file and daily transaction file          */
  if ( ( purf = _Ropen ( "MYLIB/T1677RD3", "ar+,indicators=y" )) == NULL )
  {
      printf ( "Display file did not open.¥n" );
      exit ( 1 );
  }
  if ( ( dailyf = _Ropen ( "MYLIB/T1677RDA", "wr,commit=y") ) == NULL )
  {
      printf ( "Daily transaction file did not open.¥n" );
      exit ( 2 );
  }
  /* Associate separate indicator area with purchase file           */
  _Rindara ( purf, ind_area );
  /* Select purchase record format */
  _Rformat ( purf, "PURCHASE" );
  /* Invite user to enter a purchase transaction.                   */
  /* The _Rwrite function writes the purchase display.              */
  _Rwrite ( purf, "", 0 );
  _Rreadn ( purf, buf, sizeof(buf), __DFT );
  /* While user is entering transactions, update daily and          */
  /* monthly transaction files.                                     */
  while ( rc && ind_area[PF03] == IND_OFF )
  {
    rc = (( _Rwrite ( dailyf, buf, sizeof(buf) ))->num_bytes );
  /* If the databases were updated, then commit transaction         */
  /* otherwise, rollback the transaction and indicate to the        */
  /* user that an error has occurred and end the application.       */
    if ( rc )
    {
        _Rcommit ( "Transaction complete" );
    }
    else
    {
        _Rrollbck ( );
        _Rformat ( purf, "ERROR" );
    }
    _Rwrite ( purf, "", 0 );
    _Rreadn ( purf, buf, sizeof(buf), __DFT );
  }

  _Rclose ( purf );
  _Rclose ( dailyf );
}
```

関連情報

- 301 ページの『_Ropen() ― レコード・ファイルをオープンして入出力操作を行う』

_Riofbk() ― 入出力フィードバック情報の取得

フォーマット

```
#include <recio.h>
#include <xxfdbk.h>
_XXIOFB_T *_Riofbk(_RFILE *fp);
```

言語レベル: ILE C Extension

スレッド・セーフ: はい。

説明

_Riofbk() 関数は、*fp* で指定されたファイルの入出力フィードバック域のコピーを指すポインターを戻します。

_Riofbk() 関数は、すべてのタイプのファイルの場合に有効です。

戻り値

_Riofbk() 関数は、エラーが発生した場合に NULL を戻します。 errno の設定については、 531 ページの表 12 および 535 ページの表 14 を参照してください。

_Riofbk() の使用例

```
#include <stdio.h>
#include <recio.h>
#include <string.h>
#include <stdlib.h>
typedef struct {
    char name[20];
    char address[25];
} format1 ;
typedef struct {
    char name[8];
    char password[10];
} format2 ;
typedef union {
    format1 fmt1;
    format2 fmt2;
} formats ;

int main(void)
{
    _RFILE   *fp; /* File pointer                                      */
    _RIOFB_T *rfb; /*Pointer to the file's feedback structure          */
    _XXIOFB_T *iofb; /* Pointer to the file's feedback area            */
    formats  buf, in_buf, out_buf; /* Buffers to hold data             */
 /* Open the device file.                                          */
    if (( fp = _Ropen ( "MYLIB/T1677RD2", "ar+" )) == NULL )
    {
        printf ( "Could not open file¥n" );
        exit ( 1 );
    }
        _Racquire ( fp,"DEVICE1" );    /* Acquire another device. Replace
*/
                                   /* with actual device name.       */
    _Rformat ( fp,"FORMAT1" );     /* Set the record format for the   */
                                   /* display file.                   */
    rfb = _Rwrite ( fp, "", 0 );   /* Set up the display.             */
    _Rpgmdev ( fp,"DEVICE2" );  /* Change the default program device. */
                                /* Replace with actual device name.   */
    _Rformat ( fp,"FORMAT2" );   /* Set the record format for the     */
                                 /* display file.                     */
    rfb = _Rwrite ( fp, "", 0 );   /* Set up the display.             */
    rfb = _Rwriterd ( fp, &buf, sizeof(buf) );
    rfb = _Rwrread ( fp, &in_buf, sizeof(in_buf), &out_buf,
```

```
                    sizeof(out_buf ));
    _Rreadindv ( fp, &buf, sizeof(buf), __DFT );
                                 /* Read from the first device that  */
                                 /* enters data - device becomes     */
                                 /* default program device.          */
 /* Determine which terminal responded first.                     */
    iofb = _Riofbk ( fp );
    if ( !strncmp ( "FORMAT1  ", iofb -> rec_format, 10 ))
    {
        _Rrelease ( fp, "DEVICE1" );
    }
    else
    {
        _Rrelease(fp, "DEVICE2" );
    }
 /* Continue processing.                                          */
    printf ( "Data displayed is %45.45s¥n", &buf);
    _Rclose ( fp );
}
```

関連情報

- 305 ページの『_Ropnfbk() ― オープン・フィードバック情報の取得』

_Rlocate() ― レコードの位置指定

フォーマット

```
#include <recio.h>

_RIOFB_T *_Rlocate(_RFILE *fp, void *key, int klen_rrn, int opts);
```

言語レベル: ILE C Extension

スレッド・セーフ: はい。ただし、スレッド間でファイル・ポインターが渡された場合、入出力フィードバック域はそれらのスレッド間で共有されます。

ジョブ CCSID インターフェース: この関数に送信される文字データは、すべてジョブの CCSID 内にあると想定されます。この関数によって戻された文字データは、すべてジョブの CCSID 内にあります。詳細については、 549 ページの『CCSID およびロケールの理解』を参照してください。

説明

`_Rlocate()` 関数は、*fp* に関連し、 *key*、*klen_rrn*、*opts* の各パラメーターで指定されたファイルにある、レコードの位置指定を行います。 `_Rlocate()` 関数は、__NO_LOCK が指定されない場合、 *key*、*klen_rrn* および *opts* パラメーターで指定されたレコードをロックします。

`_Rlocate()` 関数は、`_Ropen()` 関数でオープンされたデータベースと DDM ファイルの場合に有効です。 `_Rlocate()` 関数の有効なパラメーターは、以下のとおりです。

key
: 位置決めに使用するキー・フィールドを含むストリングを指します。

klen_rrn
: キーによる位置決めの場合はキーの長さ、相対レコード番号による位置決めの場合は相対レコード番号の長さを指定します。

opts
: 位置指定操作に使用する位置決めオプションを指定します。指定できるマクロは、以下のとおりです。

__DFT デフォルトは __KEY_EQ に設定され、ファイルが更新用にオープンしている場合には、更新のためにレコードをロックします。

__END
ファイル内の最後のレコードの直後に位置指定します。この位置に関連したレコードはありません。

__END_FRC
ファイル内の最後のレコードの直後に位置指定します。バッファーに入れられた変更は、すべて永続します。この位置に関連したレコードはありません。

__FIRST
現在 *fp* で使用されているアクセス・パス内の最初のレコードに位置指定します。 *key* パラメーターは無視されます。

__KEY_EQ
指定されたキーを持つ最初のレコードに位置指定します。

__KEY_GE
指定されたキー以上のキーを持つ最初のレコードに位置指定します。

__KEY_GT
指定されたキーより大きいキーを持つ最初のレコードに位置指定します。

__KEY_LE
指定されたキー以下のキーを持つ最初のレコードに位置指定します。

__KEY_LT
指定されたキーより小さいキーを持つ最初のレコードに位置指定します。

__KEY_NEXTEQ
現在位置で、長さが *klen_rrn* のキー値に等しいキーを持つ次のレコードに位置指定します。 *key* パラメーターは無視されます。

__KEY_NEXTUNQ
アクセス・パス内の現在位置から、ユニーク・キーを持つ次のレコードに位置指定します。 *key* パラメーターは無視されます。

__KEY_PREVEQ
現在位置で、長さが *klen_rrn* のキー値に等しいキーを持つ以前のレコードに位置指定します。 *key* パラメーターは無視されます。

__KEY_PREVUNQ
アクセス・パス内の現在位置から、ユニーク・キーを持つ以前のレコードに位置指定します。 *key* パラメーターは無視されます。

__LAST
現在 *fp* で使用されているアクセス・パス内の最後のレコードに位置指定します。 *key* パラメーターは無視されます。

__NEXT
現在 *fp* で使用されているアクセス・パス内の次のレコードに位置指定します。 *key* パラメーターは無視されます。

__PREVIOUS
現在 *fp* で使用されているアクセス・パス内の以前のレコードに位置指定します。 *key* パラメーターは無視されます。

__RRN_EQ
: *klen_rrn* パラメーターで指定された相対レコード番号を持つレコードに位置指定します。

__START
: ファイル内の直前の最初のレコードに位置指定します。この位置に関連したレコードはありません。

__START_FRC
: ファイル内の最初のレコードの直前のレコードに位置指定します。この位置に関連したレコードはありません。バッファーに入れられた変更は、すべて永続します。

__DATA_ONLY
: データ・レコードのみに位置指定します。削除されたレコードは無視されます。

__KEY_NULL_MAP
: キーによってレコードに位置指定する際に、NULL キー・マップを考慮します。

__NO_LOCK
: 位置指定されたレコードは、ロックされません。

__NO_POSITION
: ファイルの位置は変更されませんが、ファイルが更新のためにオープンしている場合、位置指定されたレコードはロックされます。

__PRIOR
: 要求されたレコードの直前に位置指定します。

他のオプションとともに開始または終了オプション (__START、__START_FRC、__END または __END_FRC) を指定する場合は、開始または終了オプションが優先し、他のオプションは無視されます。

ユーザーが __START または __END に位置指定され、_Rreads 操作を実行している場合、errno は EIOERROR に設定されます。

戻り値

`_Rlocate()` 関数は、*fp* に関連した _RIOFB_T 構造体を指すポインターを戻します。 `_Rlocate()` 操作が正常終了した場合、num_bytes フィールドには 1 が含まれます。 __START、__START_FRC、_END または __END_FRC が指定された場合、num_bytes フィールドは EOF に設定されます。 `_Rlocate()` 操作が正常終了しなかった場合、num_bytes フィールドにはゼロが含まれます。 *key* および *rrn* フィールドは更新され、 *key* フィールドには、部分的なキーが指定されていても完全なキーが含まれます。

errno の値は、次のいずれかに設定されます。

表 5. Errno の値

値	意味
EBADKEYLN	指定されたキーの長さが無効です。
ENOTREAD	ファイルは読み取り操作用にオープンされていません。
EIOERROR	リカバリー不能な入出力エラーが発生しました。
EIORECERR	リカバリー可能な入出力エラーが発生しました。

errno の設定については、531 ページの表 12 および 535 ページの表 14 を参照してください。

`_Rlocate()` の使用例

```
#include <stdio.h>
#include <stdlib.h>
#include <recio.h>

int main(void)
{
    _RFILE   *in;
    char     new_purchase[21] = "PEAR      1002022244";

    /* Open the file for processing in keyed sequence.              */

    if ( (in = _Ropen("MYLIB/T1677RD4", "rr+, arrseq=N")) == NULL )
    {
        printf("Open failed¥n");
        exit(1);
    };

    /* Update the first record in the keyed sequence.               */

    _Rlocate(in, NULL, 0, __FIRST);
    _Rupdate(in, new_purchase, 20);

    /* Force the end of data.                                       */

    _Rfeod(in);

    _Rclose(in);
}
```

関連情報

- 『_Ropen() ― レコード・ファイルをオープンして入出力操作を行う』

_Ropen() ― レコード・ファイルをオープンして入出力操作を行う

フォーマット

```
#include <recio.h>

_RFILE *_Ropen(const char * filename, const char * mode, ...);
```

言語レベル: ILE C Extension

スレッド・セーフ: はい。

説明

_Ropen() 関数は、*mode* パラメーターに従って、 *filename* で指定されたレコード・ファイルをオープンします。 *mode* パラメーターに *varparm* キーワード・パラメーターが指定された場合、この後にはオプションのパラメーターが続きます。オープン・モード・パラメーターおよびキーワード・パラメーターは、コンマと 1 つ以上のスペースで区切られます。 _Ropen() 関数は、データベース・ファイルを動的に作成しません。 _Ropen() 関数で参照するファイルは、すべて実在するものでなければなりません。そうでない場合、オープン操作は失敗します。

_Ropen() 関数でオープンされたファイルは、属している活動化グループが終了すると、暗黙的にクローズされます。ある活動化グループ内のオープンしているファイルを指すポインターが、別の活動化グループに渡され、そのオープンしている活動化グループが終了すると、ファイル・ポインターは、以後無効になります。

_Ropen() 関数は、すべてのタイプのファイルに適用されます。 *filename* 変数は、有効な任意の i5/OS システム・ファイル名です。

mode パラメーターは、ファイルに対して要求されるアクセスのタイプを指定します。このパラメーターには、オープン・モードとその後に続くオプションのキーワード・パラメーターが含まれます。 *mode* パラメーターは、以下のいずれかの値になります。

モード 説明

rr 既存のファイルをオープンして、レコードを読み取ります。

wr 既存のファイルをオープンして、レコードを書き込みます。ファイルにデータが含まれている場合、そのファイルが論理ファイルでなければ、その内容はクリアされます。

ar 既存のファイルをオープンして、ファイルの終わりにレコードを書き込みます (追加)。

rr+ 既存のファイルをオープンして、レコードの読み取り、書き込み、または更新を行います。

wr+ 既存のファイルをオープンして、レコードの読み取り、書き込み、または更新を行います。ファイルにデータが含まれている場合、そのファイルが論理ファイルでなければ、その内容はクリアされます。

ar+ 既存のファイルをオープンして、レコードの読み取りおよび書き込みを行います。データは、すべてファイルの終わりに書き込まれます。

mode の後には、以下のいずれかのキーワード・パラメーターが続きます。

キーワード
説明

arrseq=*value*
この場合、指定できる *value* は、以下の通りです。

Y ファイルを到着順に処理することを指定します。

N ファイルの作成時に使用したアクセス・パスを使用してファイルを処理することを指定します。これはデフォルトです。

blkrcd=*value*
この場合、指定できる *value* は、以下の通りです。

Y レコード・ブロッキングを実行します。 i5/OS オペレーティング・システムは、ユーザー向けに最も効率的なブロック・サイズを決定します。このパラメーターは、データベース、DDM、ディスケットおよびテープ・ファイルの場合に有効です。このパラメーターは、入力専用または出力専用 (rr、wr、または ar の各モード) にオープンしているファイルの場合にのみ有効です。

N レコード・ブロッキングを実行しません。これはデフォルトです。

ccsid=*value*
ファイルの変換に使用する CCSID を指定します。デフォルトは 0 で、ジョブ CCSID が使用されることを示します。

commit=*value*
この場合、指定できる *value* は、以下の通りです。

Y コミットメント制御下でデータベース・ファイルをオープンすることを指定します。あらかじめコミットメント制御をセットアップしておく必要があります。

N コミットメント制御下でデータベース・ファイルをオープンしないことを指定します。これはデフォルトです。

dupkey=*value*
指定できる *value* は、以下の通りです。

Y _RIOFB_T 構造体内で重複キー値にフラグを立てます。

N 重複キー値にはフラグを立てません。これはデフォルトです。

indicators=*value*
プリンター、ディスプレイ、および ICF ファイルに対して、標識が有効になります。指定できる *value* は、以下の通りです。

Y ファイルに関連した標識は、入出力バッファー内ではなく、別々の標識域に戻されます。

N 標識は、入出力バッファー内に戻されます。これはデフォルトです。

lrecl=*value*
固定長レコードの長さと、可変長レコードの最大長 (バイト単位)。このパラメーターは、ディスケット、ディスプレイ、プリンター、テープ、および保管ファイルの場合に有効です。

nullcap=*value*
この場合、指定できる *value* は、以下の通りです。

Y プログラムは、レコード内のヌル・フィールドを処理できます。これは、データベースと DDM ファイルの場合に有効です。

N プログラムは、レコード内のヌル・フィールドを処理できません。これはデフォルトです。

riofb=*value*
この場合、指定できる *value* は、以下の通りです。

Y _RIOFB_T 構造体内のすべてのフィールドが、_RIOFB_T 構造体を指すポインターを戻す入出力操作によって更新されます。ただし、_Rreadk 関数を使用している場合、blk_filled_by フィールドは更新されません。これはデフォルトです。

N _RIOFB_T 構造体内の num_bytes フィールドのみが更新されます。

rtncode=*value*
この場合、指定できる *value* は、以下の通りです。

Y このオプションを使用すると、例外の生成と処理をバイパスします。これにより、ファイルの終わりの場合とレコードが見つからない場合のパフォーマンスが向上します。ファイルの終わりが検出された場合、num_bytes は EOF に設定されますが、errno 値は生成されません。レコードが見つからない場合、num_bytes はゼロに設定され、errno は EIORECERR に設定されます。このパラメーターは、データベースと DDM ファイルの場合にのみ有効です。 DDM ファイルの場合、num_bytes は _Rfeod については更新されません。

N ファイルの終わりの場合とレコードが見つからない場合に、例外の生成と処理の通常プロセスが行われます。これはデフォルトです。

secure=*value*
この場合、指定できる *value* は、以下の通りです。

Y ファイルをオーバーライドから保護します。

N ファイルをオーバーライドから保護しません。これはデフォルトです。

splfname=(*value*)
スプール出力専用です。この場合、値は以下のいずれかになります。

***FILE** スプール出力ファイル名にプリンター・ファイルの名前を使用します。

spool-file-name
スプール出力ファイルの名前を指定します。使用できるのは最大 10 文字です。

usrdta=(*value*)
スプール出力専用に、ファイルを識別するユーザー指定のデータを指定します。

user-data
10 文字までのユーザー指定のテキストを指定します。

varparm=(*list*)
この場合、*(list)* は、 `_Ropen()` にどのオプション・パラメーターを渡すかを示すオプション・キーワードのリストです。リスト内のキーワードの順序は、*mode* パラメーターの後に表示されるオプション・パラメーターの順序を表します。有効なオプション・キーワードは、以下のとおりです。

lvlchk lvlchk キーワードは、#pragma mapinc で lvlchk オプションと結合して使用します。このキーワードを使用する場合、タイプ _LVLCHK_T (#pragma mapinc で生成) のオブジェクトを指すポインターは、 `_Ropen()` 関数のモード・パラメーターの後に指定する必要があります。このポインターについての詳細は、「*IBM Rational Development Studio for i: ILE C/C++ Programmer's Guide*」の #pragma mapinc の lvlchk オプションを参照してください。

vlr=*value*
可変長レコード。*value* は、ファイルに書き込まれるレコードの最小バイト長を表します。この値は -1 になるか、または 0 からファイルの最大レコード長までの範囲の値になります。このパラメーターは、データベースと DDM ファイルの場合に有効です。

VLR 処理が必要な場合、`_Ropen()` は min_length フィールドを設定します。デフォルト値が使用されない場合は、ユーザーが提供する最小値が直接 min_length フィールドにコピーされます。デフォルト値が指定された場合、`_Ropen()` は、オープン・データ・パスの DB 部分から最小長を取得します。

戻り値

`_Ropen()` 関数は、ファイルが正常にオープンされた場合、タイプ _RFILE の構造体を指すポインターを戻します。ファイルのオープンが正常に実行されなかった場合は NULL を戻します。

errno の値は、次のいずれかに設定されます。

値 **意味**

EBADMODE
指定されたファイル・モードが無効です。

EBADNAME
指定されたファイル名が無効です。

ECONVERT
変換エラーが発生しました。

ENOTOPEN
ファイルはオープンされていません。

EIOERROR
: リカバリー不能な入出力エラーが発生しました。

EIORECERR
: リカバリー可能な入出力エラーが発生しました。

errno の設定については、 531 ページの表 12 および 535 ページの表 14 を参照してください。

_Ropen() の使用例

```
#include <stdio.h>
#include <stdlib.h>
#include <recio.h>

int main(void)
{
    _RFILE      *fp;

    /* Open the file for processing in arrival sequence.             */
    if (( fp = _Ropen ( "MYLIB/T1677RD1", "rr+, arrseq=Y" )) == NULL )
    {
        printf ( "Open failed¥n" );
        exit ( 1 );
    }
    else
    /* Do some processing */;


    _Rclose ( fp );
}
```

関連情報

- 270 ページの『_Rclose() ― ファイルのクローズ』
- 10 ページの『<recio.h>』

_Ropnfbk() ― オープン・フィードバック情報の取得

フォーマット

```
#include <recio.h>
#include <xxfdbk.h>

_XXOPFB_T *_Ropnfbk(_RFILE *fp);
```

言語レベル: ILE C Extension

スレッド・セーフ: はい。

説明

_Ropnfbk() 関数は、*fp* で指定されたファイルのオープン・フィードバック域のコピーを指すポインターを戻します。

_Ropnfbk() 関数は、すべてのタイプのファイルの場合に有効です。

戻り値

_Ropnfbk() 関数は、エラーが発生した場合に NULL を戻します。 errno の設定については、 531 ページの表 12 および 535 ページの表 14 を参照してください。

_Ropnfbk() の使用例

```
#include <stdio.h>
#include <stdlib.h>
#include <recio.h>

int main(void)
{
    _RFILE      *fp;
    _XXOPFB_T   *opfb;

    /* Open the file for processing in arrival sequence.            */
    if (( fp = _Ropen ( "MYLIB/T1677RD1", "rr+, arrseq=Y" )) == NULL )
    {
        printf ( "Open failed¥n" );
        exit ( 1 );
    }

    /* Get the library and file names of the file opened.           */
    opfb = _Ropnfbk ( fp );
    printf ( "Library: %10.10s¥nFile:    %10.10s¥n",
              opfb->library_name,
              opfb->file_name);


    _Rclose ( fp );
}
```

関連情報

- 334 ページの『_Rupfb() ― 最終入出力操作についての情報の説明』

_Rpgmdev() ― デフォルトのプログラム装置の設定

フォーマット

```
#include <recio.h>
int _Rpgmdev(_RFILE *fp, char *dev);
```

言語レベル: ILE C Extension

スレッド・セーフ: いいえ。

ジョブ CCSID インターフェース: この関数に送信される文字データは、すべてジョブの CCSID 内にあると想定されます。この関数によって戻された文字データは、すべてジョブの CCSID 内にあります。詳細については、 549 ページの『CCSID およびロケールの理解』を参照してください。

説明

_Rpgmdev() 関数は、*fp* に関連したファイルの現行プログラム装置を *dev* に設定します。装置は、大文字で指定してください。

dev パラメーターは、ヌル終了 C ストリングです。

_Rpgmdev() 関数は、ディスプレイ、ICF、およびプリンター・ファイルの場合に有効です。

戻り値

_Rpgmdev() 関数は、操作が正常終了した場合には 1、指定された装置がファイル用に獲得されなかった場合にはゼロを戻します。 errno の設定については、 531 ページの表 12 および 535 ページの表 14 を参照してください。

_Rpgmdev() の使用例

```
#include <stdio.h>
#include <recio.h>
#include <string.h>
#include <stdlib.h>

typedef struct {
    char name[20];
    char address[25];
} format1 ;

typedef struct {
    char name[8];
    char password[10];
} format2 ;

typedef union {
    format1 fmt1;
    format2 fmt2;
} formats ;

int main(void)
{
    _RFILE   *fp; /* File pointer                                       */
    _RIOFB_T *rfb; /*Pointer to the file's feedback structure           */
    formats  buf, in_buf, out_buf; /* Buffers to hold data              */

    /* Open the device file.                                            */
    if (( fp = _Ropen ( "MYLIB/T1677RD2", "ar+" )) == NULL )
    {
        printf ( "Could not open file¥n" );
        exit ( 1 );
    }

    _Rpgmdev ( fp,"DEVICE2" );/* Change the default program device.   */
                              /* Replace with actual device name.     */

    _Rformat ( fp,"FORMAT2" );   /* Set the record format for the     */
                                 /* display file.                     */

    rfb = _Rwrite ( fp, "", 0 );   /* Set up the display.             */
    rfb = _Rwriterd ( fp, &buf, sizeof(buf) );

    rfb = _Rwrread ( fp, &in_buf, sizeof(in_buf), &out_buf,
                     sizeof(out_buf ));

    /* Continue processing.                                             */


    _Rclose ( fp );
}
```

関連情報

- 269 ページの『_Racquire() ー プログラム装置の獲得』
- 328 ページの『_Rrelease() ー プログラム装置の解除』

_Rreadd() ― 相対レコード番号によるレコードの読み取り

フォーマット

```
#include <recio.h>

_RIOFB_T *_Readd (_RFILE *fp, void *buf, size_t size,
                        int opts, long rrn);
```

言語レベル: ILE C Extension

スレッド・セーフ: はい。ただし、スレッド間でファイル・ポインターが渡された場合、入出力フィードバック域はそれらのスレッド間で共有されます。

説明

| `_Rreadd()` 関数は、*fp* に関連したファイルの到着順アクセス・パス内にある、*rrn* で指定されたレコード
| を読み取ります。ファイルが更新用にオープンされているときに **__NO_LOCK** が指定されない場合、
| `_Rreadd()` 関数は、*rrn* で指定されたレコードをロックします。ファイルがキー付きファイルである場合、
| キー順アクセス・パスは無視されます。最大で *size* のバイト数が、レコードから *buf* にコピーされます
| (移動モードのみ)。

`_Rreadd()` 関数の有効なパラメーターは、以下のとおりです。

buf
: 読み取ったデータを保管するバッファーを指定します。位置指定モードを使用する場合、このパラメーターは NULL に設定する必要があります。

size
: 読み取って *buf* に保管するバイト数を指定します。位置指定モードを使用する場合、このパラメーターは無視されます。

rrn
: 読み取りの対象となるレコードの相対レコード番号。

opts
: ファイルの処理とアクセスのオプションを指定します。以下のオプションを指定できます。

__DFT ファイルが更新用にオープンしている場合、読み取り中のレコードは更新のためにロックされます。以前にロックしたレコードは、以後ロックされなくなります。

__NO_LOCK
位置指定されているレコードをロックしません。

`_Rreadd()` 関数は、データベース、DDM およびディスプレイ (サブファイル) ファイルの場合に有効です。

戻り値

`_Rreadd()` 関数は、*fp* に関連した _RIOFB_T 構造体を指すポインターを戻します。 `_Rreadd()` 操作が正常終了した場合、 *num_bytes* フィールドは、システム・バッファーからユーザーのバッファーに転送されたバイト数 (移動モード)、またはファイルのレコード長 (位置指定モード) に設定されます。 blkrcd=Y および riofb=Y が指定された場合は、 _RIOFB_T 構造体の blk_count および blk_filled_by フィールドが更新されます。 *key* および *rrn* フィールドも更新されます。 *fp* に関連したファイルがディスプレイ・ファイルである場合は、sysparm フィールドが更新されます。これが正常終了しなかった場合、*num_bytes* フィールドは *size* より小さい値に設定され、errno が変更されます。

errno の値は、次のいずれかに設定されます。

値 意味

ENOTREAD
: ファイルは読み取り操作用にオープンされていません。

ETRUNC
: 入出力操作で切り捨てが発生しました。

EIOERROR
: リカバリー不能な入出力エラーが発生しました。

EIORECERR
: リカバリー可能な入出力エラーが発生しました。

errno の設定については、 531 ページの表 12 および 535 ページの表 14 を参照してください。

_Readd() の使用例

```
#include <stdio.h>
#include <stdlib.h>
#include <recio.h>

int main(void)
{
    _RFILE      *fp;
    _XXOPFB_T   *opfb;

    /* Open the file for processing in arrival sequence.            */
    if (( fp = _Ropen ( "MYLIB/T1677RD1", "rr+, arrseq=Y" )) == NULL )
    {
        printf ( "Open failed¥n" );
        exit ( 1 );
    }

    /* Get the library and file names of the file opened.           */
    opfb = _Ropnfbk ( fp );
    printf ( "Library: %10.10s¥nFile:    %10.10s¥n",
              opfb->library_name,
              opfb->file_name);

    /* Get the second record.                                       */
    _Readd ( fp, NULL, 20, __DFT, 2 );
    printf ( "Second record: %10.10s¥n", *(fp->in_buf) );

    _Rclose ( fp );
}
```

関連情報

- 310 ページの『_Readf() ― 最初のレコードの読み取り』
- 312 ページの『_Readindv() ― 送信勧誘された装置からの読み取り』
- 314 ページの『_Readk() ― キーによるレコードの読み取り』
- 318 ページの『_Readl() ― 最終レコードの読み取り』
- 319 ページの『_Readn() ― 次のレコードの読み取り』
- 322 ページの『_Readnc() ― サブファイル内の次の変更済みレコードの読み取り』
- 324 ページの『_Readp() ― 前のレコードの読み取り』
- 326 ページの『_Reads() ― 同じレコードの読み取り』

_Rreadf() ― 最初のレコードの読み取り

フォーマット

```
#include <recio.h>

_RIOFB_T *_Rreadf (_RFILE *fp, void *buf, size_t size, int opts);
```

言語レベル: ILE C Extension

スレッド・セーフ: はい。ただし、スレッド間でファイル・ポインターが渡された場合、入出力フィードバック域はそれらのスレッド間で共有されます。

説明

| `_Rreadf()` 関数は、*fp* で指定されたファイルの現在使用されているアクセス・パス内の最初のレコードを
| 読み取ります。アクセス・パスは、キー・シーケンスまたは到着順です。ファイルが更新用にオープンされ
| ているときに __NO_LOCK が指定されない場合、`_Rreadf()` 関数は、最初のレコードをロックします。最
| 大で *size* のバイト数が、レコードから *buf* にコピーされます (移動モードのみ)。

以下に、`_Rreadf()` 関数の有効なパラメーターを示します。

buf このパラメーターは、読み取ったデータを保管するバッファーを指定します。位置指定モードを使用する場合、このパラメーターは NULL に設定する必要があります。

size このパラメーターは、読み取って *buf* に保管するバイト数を指定します。位置指定モードを使用する場合、このパラメーターは無視されます。

opts このパラメーターは、ファイルの処理とアクセスのオプションを指定します。以下のオプションを指定できます。

__DFT ファイルが更新用にオープンしている場合、読み取り中または位置指定中のレコードは更新のためにロックされます。以前にロックしたレコードは、以後ロックされなくなります。

__NO_LOCK
位置指定されているレコードをロックしません。

`_Rreadf()` 関数は、データベースと DDM ファイルの場合に有効です。

戻り値

`_Rreadf()` 関数は、*fp* で指定された _RIOFB_T 構造体を指すポインターを戻します。 `_Rreadf()` 操作が正常終了した場合、 *num_bytes* フィールドは、システム・バッファーからユーザーのバッファーに転送されたバイト数 (移動モード)、またはファイルのレコード長 (位置指定モード) に設定されます。 *key* および *rrn* フィールドは更新されます。レコード・ブロッキングが行われている場合は、blk_count および blk_filled_by フィールドが更新されます。ファイルが空である場合、*num_bytes* フィールドは EOF に設定されます。これが正常終了しなかった場合、 *num_bytes* フィールドは *size* より小さい値に設定され、errno が変更されます。

errno の値は、次のいずれかに設定されます。

値 意味

ENOTREAD
ファイルは読み取り操作用にオープンされていません。

ETRUNC
入出力操作で切り捨てが発生しました。

EIOERROR
リカバリー不能な入出力エラーが発生しました。

EIORECERR
リカバリー可能な入出力エラーが発生しました。

errno の設定については、 531 ページの表 12 および 535 ページの表 14 を参照してください。

_Rreadf() の使用例

```
#include <stdio.h>
#include <stdlib.h>
#include <recio.h>

int main(void)
{
    _RFILE      *fp;
    _XXOPFB_T   *opfb;

    /* Open the file for processing in arrival sequence.            */
    if (( fp = _Ropen ( "MYLIB/T1677RD1", "rr+, arrseq=Y" )) == NULL )
    {
        printf ( "Open failed¥n" );
        exit ( 1 );
    }

    /* Get the library and file names of the file opened.           */
    opfb = _Ropnfbk ( fp );
    printf ( "Library: %10.10s¥nFile:    %10.10s¥n",
              opfb->library_name,
              opfb->file_name);

    /* Get the first record.                                        */
    _Rreadf ( fp, NULL, 20, __DFT );
    printf ( "First record:  %10.10s¥n", *(fp->in_buf) );

    /* Delete the first record.                                     */
    _Rdelete ( fp );

    _Rclose ( fp );
}
```

関連情報

- 308 ページの『_Rreadd() ― 相対レコード番号によるレコードの読み取り』
- 312 ページの『_Rreadindv() ― 送信勧誘された装置からの読み取り』
- 314 ページの『_Rreadk() ― キーによるレコードの読み取り』
- 318 ページの『_Rreadl() ― 最終レコードの読み取り』
- 319 ページの『_Rreadn() ― 次のレコードの読み取り』
- 322 ページの『_Rreadnc() ― サブファイル内の次の変更済みレコードの読み取り』
- 324 ページの『_Rreadp() ― 前のレコードの読み取り』
- 326 ページの『_Rreads() ― 同じレコードの読み取り』

_Rreadindv() — 送信勧誘された装置からの読み取り

フォーマット

```
#include <recio.h>

_RIOFB_T *_Rreadindv(_RFILE *fp, void *buf, size_t size, int opts);
```

言語レベル: ILE C Extension

スレッド・セーフ: いいえ。

説明

`_Rreadindv()` 関数は、送信勧誘された装置からデータを読み取ります。

以下に、`_Rreadindv()` 関数の有効なパラメーターを示します。

buf
: 読み取ったデータを保管するバッファーを指定します。位置指定モードを使用する場合、このパラメーターは NULL に設定する必要があります。

size
: 読み取って *buf* に保管するバイト数を指定します。位置指定モードを使用する場合、このパラメーターは無視されます。

opts
: ファイルの処理オプションを指定します。指定できる値は、以下のとおりです。

: **__DFT** ファイルが更新用にオープンしている場合、読み取り中または位置指定中のレコードはロックされます。そうでない場合、このオプションは無視されます。

`_Rreadindv()` 関数は、ディスプレイ・ファイルと ICF ファイルの場合に有効です。

戻り値

`_Rreadindv()` 関数は、*fp* に関連した _RIOFB_T 構造体を指すポインターを戻します。 `_Rreadindv()` 操作が正常終了した場合、num_bytes フィールドは、システム・バッファーからユーザーのバッファーに転送されたバイト数 (移動モード)、またはファイルのレコード長 (位置指定モード) に設定されます。 sysparm および rrn (サブファイルの場合) フィールドも更新されます。ファイルが空である場合、*num_bytes* フィールドは EOF に設定されます。 `_Rreadindv()` 関数が正常終了しなかった場合、num_bytes フィールドは *size* の値より小さい値に設定され、 errno が変更されます。

errno の値は、次のいずれかに設定されます。

値　　意味

ENOTREAD
: ファイルは読み取り操作用にオープンされていません。

ETRUNC
: 入出力操作で切り捨てが発生しました。

EIOERROR
: リカバリー不能な入出力エラーが発生しました。

EIORECERR
: リカバリー可能な入出力エラーが発生しました。

errno の設定については、 531 ページの表 12 および 535 ページの表 14 を参照してください。

_Rreadindv() の使用例

```
#include <stdio.h>
#include <recio.h>
#include <string.h>
#include <stdlib.h>
typedef struct {
    char name[20];
    char address[25];
} format1 ;
typedef struct {
    char name[8];
    char password[10];
} format2 ;
typedef union {
    format1 fmt1;
    format2 fmt2;
} formats ;
int main(void)
{
    _RFILE   *fp;               /* File pointer
    */
    _RIOFB_T *rfb;        /* Pointer to the file's feedback structure
    */
    _XXIOFB_T *iofb;          /* Pointer to the file's feedback area
    */
    formats  buf, in_buf, out_buf          /* Buffers to hold data
    */
 /* Open the device file.                                          */
    if (( fp = _Ropen ( "MYLIB/T1677RD2", "ar+" )) == NULL )
    {
        printf ( "Could not open file¥n" );
        exit ( 1 );
    }
   _Racquire ( fp,"DEVICE1" );    /* Acquire another device. Replace */
                                   /* with actual device name.       */
    _Rformat ( fp,"FORMAT1" );     /* Set the record format for the   */
                                   /* display file.                   */
    rfb = _Rwrite ( fp, "", 0 );   /* Set up the display.             */
    _Rpgmdev ( fp,"DEVICE2" );  /* Change the default program device. */
                               /* Replace with actual device name.   */
    _Rformat ( fp,"FORMAT2" );   /* Set the record format for the     */
                                 /* display file.                     */
    rfb = _Rwrite ( fp, "", 0 );   /* Set up the display.             */
    rfb = _Rwriterd ( fp, &buf, sizeof(buf) );
    rfb = _Rwrread ( fp, &in_buf, sizeof(in_buf), &out_buf,
                     sizeof(out_buf ));
    _Rreadindv ( fp, &buf, sizeof(buf), __DFT );
                                  /* Read from the first device that  */
                                  /* enters data - device becomes     */
                                  /* default program device.          */
 /* Determine which terminal responded first.                      */
    iofb = _Riofbk ( fp );
    if ( !strncmp ( "FORMAT1  ", iofb -> rec_format, 10 ))
    {
        _Rrelease ( fp, "DEVICE1" );
    }
    else
    {
        _Rrelease(fp, "DEVICE2" );
    }
 /* Continue processing.                                           */
    printf ( "Data displayed is %45.45s¥n", &buf);
    _Rclose ( fp );
}
```

関連情報

- 308 ページの『_Readd() ― 相対レコード番号によるレコードの読み取り』
- 310 ページの『_Readf() ― 最初のレコードの読み取り』
- 『_Rreadk() ― キーによるレコードの読み取り』
- 318 ページの『_Readl() ― 最終レコードの読み取り』
- 319 ページの『_Readn() ― 次のレコードの読み取り』
- 322 ページの『_Readnc() ― サブファイル内の次の変更済みレコードの読み取り』
- 324 ページの『_Readp() ― 前のレコードの読み取り』
- 326 ページの『_Reads() ― 同じレコードの読み取り』

_Rreadk() ― キーによるレコードの読み取り

フォーマット

```
#include <recio.h>

_RIOFB_T *_Rreadk(_RFILE *fp, void *buf, size_t size,
                  int opts, void *key, unsigned int keylen);
```

言語レベル: ILE C Extension

スレッド・セーフ: はい。ただし、スレッド間でファイル・ポインターが渡された場合、入出力フィードバック域はそれらのスレッド間で共有されます。

説明

| `_Rreadk()` 関数は、*fp* に関連したファイルの現在使用されているキー順アクセス・パス内のレコードを読み取ります。最大で *size* のバイト数が、レコードから *buf* にコピーされます (移動モードのみ)。ファイルが更新用にオープンされているときに __NO_LOCK が指定されない場合、`_Rreadk()` 関数は、位置指定されたレコードをロックします。ファイルは、キー・シーケンス・パスを使用して処理する必要があります。

以下に、`_Rreadk()` 関数の有効なパラメーターを示します。

buf 読み取ったデータを保管するバッファーを指定します。位置指定モードを使用する場合、このパラメーターは NULL に設定する必要があります。

size 読み取って *buf* に保管するバイト数を指定します。位置指定モードを使用する場合、このパラメーターは無視されます。

key 読み取りに使用するキーを指します。

keylen 使用するキーの長さの合計を指定します。

opts ファイルの処理オプションを指定します。指定できる値は、以下のとおりです。

__DFT デフォルトは __KEY_EQ に設定されます。

__KEY_EQ
指定されたキーを持つ最初のレコードに位置指定、それを読み取ります。

__KEY_GE
指定されたキー以上のキーを持つ最初のレコードに位置指定し、それを読み取ります。

__KEY_GT
: 指定されたキーより大きいキーを持つ最初のレコードに位置指定し、それを読み取ります。

__KEY_LE
: 指定されたキー以下のキーを持つ最初のレコードに位置指定し、それを読み取ります。

__KEY_LT
: 指定されたキーより小さいキーを持つ最初のレコードに位置指定し、それを読み取ります。

__KEY_NEXTEQ
: 現在位置で、キー値に等しいキーを持つ次のレコードに位置指定し、それを読み取ります。 *key* パラメーターは無視されます。

__KEY_NEXTUNQ
: アクセス・パス内の現在位置から、ユニーク・キーを持つ次のレコードに位置指定し、それを読み取ります。 *key* パラメーターは無視されます。

__KEY_PREVEQ
: 現在位置で、キー値に等しいキーを持つ最後のレコードに位置指定し、それを読み取ります。 *key* パラメーターは無視されます。

__KEY_PREVUNQ
: アクセス・パス内の現在位置から、ユニーク・キーを持つ以前のレコードに位置指定し、それを読み取ります。 *key* パラメーターは無視されます。

__NO_LOCK
: 更新のためにレコードをロックしません。

位置決めオプションは、相互に排他的です。

以下のオプションは、bit-wise OR (|) 演算子を使用して、位置決めオプションと結合することができます。

__KEY_NULL_MAP
: キーによってレコードを読み取る場合に、NULL キー・マップを考慮します。

__NO_LOCK
: 位置指定されたレコードは、ロックされません。

`_Rreadk()` 関数は、データベースと DDM ファイルの場合に有効です。

戻り値

| `_Rreadk()` 関数は、*fp* に関連した _RIOFB_T 構造体を指すポインターを戻します。 `_Rreadk()` 操作が正
| 常終了した場合、 *num_bytes* フィールドは、システム・バッファーからユーザーのバッファーに転送され
| たバイト数 (移動モード)、またはファイルのレコード長 (位置指定モード) に設定されます。 *key* および
| *rrn* フィールドは更新されます。キー・フィールドには、部分的なキーが指定されている場合、常に完全な
| キーが含まれます。レコード・ブロッキングを `_Rreadk()` と共に使用すると、ブロック内にレコードが 1
| つだけ読み込まれます。したがって、ブロック内に残されるレコードはゼロ個になり、_RIOFB_T 構造体
| の blk_count フィールドが 0 で更新されます。 blk_filled_by フィールドは `_Rreadk()` には適用されず、
| 更新もされません。 *key* で指定されたレコードが見つからない場合、*num_bytes* フィールドはゼロまたは
| EOF に設定されます。部分的なキーによってレコードを読み取る場合は、キー全体がフィードバック構造
| 体に戻されます。これが正常終了しなかった場合、*num_bytes* フィールドは *size* より小さい値に設定さ
| れ、errno が変更されます。

errno の値は、次のいずれかに設定されます。

値 **意味**

EBADKEYLN
指定されたキーの長さが無効です。

ENOTREAD
ファイルは読み取り操作用にオープンされていません。

ETRUNC
入出力操作で切り捨てが発生しました。

EIOERROR
リカバリー不能な入出力エラーが発生しました。

EIORECERR
リカバリー可能な入出力エラーが発生しました。

errno の設定については、531 ページの表 12 および 535 ページの表 14 を参照してください。

_Rreadk() の使用例

```
#include <stdio.h>
#include <recio.h>
#include <stdlib.h>

int main(void)
{
    _RFILE   *fp;
    _RIOFB_T *fb;
    char      buf[4];
    /* Create a physical file                                     */
    system("CRTPF FILE(QTEMP/MY_FILE)");
    /* Open the file for write                                    */
    if ( (fp = _Ropen("QTEMP/MY_FILE", "wr")) == NULL )
    {
        printf("open for write fails¥n");
        exit(1);
    }
    /* write some records into the file                           */
    _Rwrite(fp, "KEY9", 4);
    _Rwrite(fp, "KEY8", 4);
    _Rwrite(fp, "KEY7", 4);
    _Rwrite(fp, "KEY6", 4);
    _Rwrite(fp, "KEY5", 4);
    _Rwrite(fp, "KEY4", 4);
    _Rwrite(fp, "KEY3", 4);
    _Rwrite(fp, "KEY2", 4);
    _Rwrite(fp, "KEY1", 4);
    /* Close the file                                             */
    _Rclose(fp);
    /* Open the file for read                                     */
    if ( (fp = _Ropen("QTEMP/MY_FILE", "rr")) == NULL )
    {
        printf("open for read fails¥n");
        exit(2);
    }
    /* Read the record with key KEY3                              */
    fb = _Rreadk(fp, buf, 4, __KEY_EQ, "KEY3", 4);
    printf("record %d with value %4.4s¥n", fb->rrn, buf);
    /* Read the next record with key less than KEY3               */
    fb = _Rreadk(fp, buf, 4, __KEY_LT, "KEY3", 4);
    printf("record %d with value %4.4s¥n", fb->rrn, buf);
    /* Read the next record with key greater than KEY3            */
    fb = _Rreadk(fp, buf, 4, __KEY_GT, "KEY3", 4);
    printf("record %d with value %4.4s¥n", fb->rrn, buf);
    /* Read the next record with different key                    */
    fb = _Rreadk(fp, buf, 4, __KEY_NEXTUNQ, "", 4);
    printf("record %d with value %4.4s¥n", fb->rrn, buf);
    /* Close the file                                             */
    _Rclose(fp);
}
```

関連情報

- 308 ページの『_Readd() ― 相対レコード番号によるレコードの読み取り』
- 310 ページの『_Rreadf() ― 最初のレコードの読み取り』
- 312 ページの『_Rreadindv() ― 送信勧誘された装置からの読み取り』
- 318 ページの『_Rreadl() ― 最終レコードの読み取り』
- 319 ページの『_Rreadn() ― 次のレコードの読み取り』
- 322 ページの『_Rreadnc() ― サブファイル内の次の変更済みレコードの読み取り』
- 324 ページの『_Rreadp() ― 前のレコードの読み取り』
- 326 ページの『_Rreads() ― 同じレコードの読み取り』

_Readl() ― 最終レコードの読み取り

フォーマット

```
#include <recio.h>

_RIOFB_T *_Readl(_RFILE *fp, void *buf, size_t size, int opts);
```

言語レベル: ILE C Extension

スレッド・セーフ: はい。ただし、スレッド間でファイル・ポインターが渡された場合、入出力フィードバック域はそれらのスレッド間で共有されます。

説明

| `_Readl()` 関数は、*fp* で指定されたファイルの現在使用されているアクセス・パス内の最後のレコードを
| 読み取ります。アクセス・パスは、キー・シーケンスまたは到着順です。最大で *size* のバイト数が、レコ
| ードから *buf* にコピーされます (移動モードのみ)。ファイルが更新用にオープンされているときに
| __NO_LOCK が指定されない場合、`_Readl()` 関数は、最後のレコードをロックします。

以下に、`_Readl()` 関数の有効なパラメーターを示します。

buf 読み取ったデータを保管するバッファーを指定します。位置指定モードを使用する場合、このパラメーターは NULL に設定する必要があります。

size 読み取って *buf* に保管するバイト数を指定します。位置指定モードを使用する場合、このパラメーターは無視されます。

opts ファイルの処理オプションを指定します。指定できる値は、以下のとおりです。

__DFT ファイルが更新用にオープンしている場合、読み取り中または位置指定中のレコードはロックされます。以前にロックしたレコードは、以後ロックされなくなります。

__NO_LOCK
位置指定されているレコードをロックしません。

`_Readl()` 関数は、データベースと DDM ファイルの場合に有効です。

戻り値

`_Readl()` 関数は、*fp* に関連した _RIOFB_T 構造体を指すポインターを戻します。 `_Readl()` 操作が正常終了した場合、 *num_bytes* フィールドは、システム・バッファーからユーザーのバッファーに転送されたバイト数 (移動モード)、またはファイルのレコード長 (位置指定モード) に設定されます。 *key* および *rrn* フィールドは更新されます。レコード・ブロッキングが行われている場合は、blk_count および blk_filled_by フィールドが更新されます。ファイルが空の場合、*num_bytes* フィールドは EOF に設定されます。これが正常終了しなかった場合、*num_bytes* フィールドは *size* より小さい値に設定され、errno が変更されます。

errno の値は、次のいずれかに設定されます。

値 **意味**

ENOTREAD
ファイルは読み取り操作用にオープンされていません。

ETRUNC
入出力操作で切り捨てが発生しました。

EIOERROR
リカバリー不能な入出力エラーが発生しました。

EIORECERR
リカバリー可能な入出力エラーが発生しました。

errno の設定については、 531 ページの表 12 および 535 ページの表 14 を参照してください。

_Readl() の使用例

```
#include <stdio.h>
#include <stdlib.h>
#include <recio.h>

int main(void)
{
    _RFILE      *fp;
    _XXOPFB_T   *opfb;

    /* Open the file for processing in arrival sequence.             */
    if (( fp = _Ropen ( "MYLIB/T1677RD1", "rr+, arrseq=Y" )) == NULL )
    {
        printf ( "Open failed¥n" );
        exit ( 1 );
    }

    /* Get the library and file names of the file opened.           */
    opfb = _Ropnfbk ( fp );
    printf ( "Library: %10.10s¥nFile:    %10.10s¥n",
              opfb->library_name,
              opfb->file_name);

    /* Get the last record.                                          */
    _Readl ( fp, NULL, 20, __DFT );
    printf ( "Last record: %10.10s¥n", *(fp->in_buf) );

    _Rclose ( fp );
}
```

関連情報

- 308 ページの『_Readd() ― 相対レコード番号によるレコードの読み取り』
- 310 ページの『_Readf() ― 最初のレコードの読み取り』
- 312 ページの『_Readindv() ― 送信勧誘された装置からの読み取り』
- 314 ページの『_Readk() ― キーによるレコードの読み取り』
- 『_Readn() ― 次のレコードの読み取り』
- 322 ページの『_Readnc() ― サブファイル内の次の変更済みレコードの読み取り』
- 324 ページの『_Readp() ― 前のレコードの読み取り』
- 326 ページの『_Reads() ― 同じレコードの読み取り』

_Readn() ― 次のレコードの読み取り

フォーマット

```
#include <recio.h>

_RIOFB_T *_Readn (_RFILE *fp, void *buf, size_t size, int opts);
```

言語レベル: ILE C Extension

スレッド・セーフ: はい。ただし、スレッド間でファイル・ポインターが渡された場合、入出力フィードバック域はそれらのスレッド間で共有されます。

説明

| _Rreadn() 関数は、*fp* に関連したファイルの現在使用されているアクセス・パス内の次のレコードを読み
| 取ります。アクセス・パスは、キー・シーケンスまたは到着順です。最大で *size* のバイト数が、レコード
| から *buf* にコピーされます (移動モードのみ)。ファイルが更新用にオープンされているときに
| __NO_LOCK が指定されない場合、_Rreadn() 関数は、位置指定されたレコードをロックします。

fp に関連したファイルが順次メンバー処理用にオープンされ、現行レコードの位置が、メンバーの最後のレコードである場合 (但し、ファイル内の最後のメンバーは除く)、 _Rreadn() は、ファイルの次のメンバーの最初のレコードを読み取ります。

_Rlocate() 操作が、__PRIOR オプションを指定してレコードに位置指定された場合、 _Rreadn() は、_Rlocate() 操作によって位置指定されたレコードを読み取ります。

ファイルがレコード・ブロッキング用にオープンされ、_Readp() への呼び出しによってブロックが埋められた場合に、ブロック内にレコードが残っていると _Rreadn() 関数は無効になります。_RIOFB_T 内の blk_count をチェックすると、残っているレコードがないかを確認できます。

以下に、_Rreadn() 関数の有効なパラメーターを示します。

buf 読み取ったデータを保管するバッファーを指定します。位置指定モードを使用する場合、このパラメーターは NULL に設定する必要があります。

size 読み取って *buf* に保管するバイト数を指定します。位置指定モードを使用する場合、このパラメーターは無視されます。

opts ファイルの処理オプションを指定します。指定できる値は、以下のとおりです。

__DFT ファイルが更新用にオープンしている場合、読み取り中または位置指定中のレコードはロックされます。以前にロックしたレコードは、以後ロックされなくなります。

__NO_LOCK
位置指定されているレコードをロックしません。

_Rreadn() 関数は、プリンター・ファイルを除くすべてのタイプのファイルの場合に有効です。

戻り値

_Rreadn() 関数は、*fp* に関連した _RIOFB_T 構造体を指すポインターを戻します。 _Rreadn() 操作が正常終了した場合、num_bytes フィールドは、システム・バッファーからユーザーのバッファーに転送されたバイト数 (移動モード)、またはファイルのレコード長 (位置指定モード) に設定されます。 *key* および *rrn* フィールドは更新されます。 *fp* に関連したファイルがディスプレイ・ファイルである場合は、sysparm フィールドも更新されます。レコード・ブロッキングが行われている場合は、 _RIOFB_T 構造体の blk_count および the blk_filled_by フィールドが更新されます。ファイル内の最後のレコード以外を読み取ろうとすると、num_bytes フィールドが EOF に設定されます。これが正常終了しなかった場合、num_bytes フィールドは、*size* より小さい値に設定され、errno が変更されます。装置ファイルを使用しており、*size* としてゼロを指定する場合は、 errno をチェックして、関数が正常終了したかどうかを判別してください。

errno の値は、次のいずれかに設定されます。

値 **意味**

ENOTREAD
: ファイルは読み取り操作用にオープンされていません。

ETRUNC
: 入出力操作で切り捨てが発生しました。

EIOERROR
: リカバリー不能な入出力エラーが発生しました。

EIORECERR
: リカバリー可能な入出力エラーが発生しました。

errno の設定については、 531 ページの表 12 および 535 ページの表 14 を参照してください。

_Readn() の使用例

```
#include <stdio.h>
#include <stdlib.h>
#include <recio.h>

int main(void)
{
    _RFILE      *fp;
    _XXOPFB_T   *opfb;

    /* Open the file for processing in arrival sequence.            */
    if (( fp = _Ropen ( "MYLIB/T1677RD1", "rr+, arrseq=Y" )) == NULL )
    {
        printf ( "Open failed¥n" );
        exit ( 1 );
    }

    /* Get the library and file names of the file opened.           */
    opfb = _Ropnfbk ( fp );
    printf ( "Library: %10.10s¥nFile:    %10.10s¥n",
              opfb->library_name,
              opfb->file_name);

    /* Get the first record.                                        */
    _Readf ( fp, NULL, 20, __DFT );
    printf ( "First record:  %10.10s¥n", *(fp->in_buf) );

    /* Delete the second record.                                    */
    _Readn ( fp, NULL, 20, __DFT );
    _Rdelete ( fp );


    _Rclose ( fp );
}
```

関連情報

- 308 ページの『_Readd() ― 相対レコード番号によるレコードの読み取り』
- 310 ページの『_Readf() ― 最初のレコードの読み取り』
- 312 ページの『_Readindv() ― 送信勧誘された装置からの読み取り』
- 314 ページの『_Readk() ― キーによるレコードの読み取り』
- 318 ページの『_Readl() ― 最終レコードの読み取り』
- 322 ページの『_Readnc() ― サブファイル内の次の変更済みレコードの読み取り』
- 324 ページの『_Readp() ― 前のレコードの読み取り』

• 326 ページの『_Rreads() ― 同じレコードの読み取り』

_Rreadnc() ― サブファイル内の次の変更済みレコードの読み取り

フォーマット

```
#include <recio.h>

_RIOFB_T *_Rreadnc(_RFILE *fp, void *buf, size_t size);
```

言語レベル: ILE C Extension

スレッド・セーフ: いいえ。

説明

`_Rreadnc()` 関数は、*fp* に関連したサブファイル内の現在位置から次のレコードを読み取ります。画面から読み取られるデータの最小 *size* が、システム・バッファーから *buf* にコピーされます。

以下に、`_Rreadnc()` 関数の有効なパラメーターを示します。

buf
: 読み取ったデータを保管するバッファーを指定します。位置指定モードを使用する場合、このパラメーターは NULL に設定する必要があります。

size
: 読み取って *buf* に保管するバイト数を指定します。

`_Rreadnc()` 関数は、サブファイルの場合に有効です。

戻り値

`_Rreadnc()` 関数は、*fp* に関連した _RIOFB_T 構造体を指すポインターを戻します。 `_Rreadnc()` 操作が正常終了した場合、num_bytes フィールドは、システム・バッファーからユーザーのバッファーに転送されたバイト数 (移動モード)、またはファイルのレコード長 (位置指定モード) に設定されます。 rrn および sysparm フィールドは更新されます。現在位置とファイルの終わりの間で変更されたレコードがない場合、num_bytes フィールドは EOF に設定されます。これが正常終了しなかった場合、num_bytes フィールドは、*size* より小さい値に設定され、errno が変更されます。

errno の値は、次のいずれかに設定されます。

値　　意味

ENOTREAD
: ファイルは読み取り操作用にオープンされていません。

ETRUNC
: 入出力操作で切り捨てが発生しました。

EIOERROR
: リカバリー不能な入出力エラーが発生しました。

EIORECERR
: リカバリー可能な入出力エラーが発生しました。

errno の設定については、 531 ページの表 12 および 535 ページの表 14 を参照してください。

`_Rreadnc()` の使用例

```
#include <stdio.h>
#include <stdlib.h>
#include <recio.h>
#define LEN          10
#define NUM_RECS     20
#define SUBFILENAME  "MYLIB/T1677RD6"
#define PFILENAME    "MYLIB/T1677RDB"
typedef struct  {
    char name[LEN];
    char phone[LEN];
} pf_t;
#define RECLEN sizeof(pf_t)
void init_subfile(_RFILE *, _RFILE *);

int main(void)
{
    _RFILE          *pf;
    _RFILE          *subf;
    /************************************************
     * Open the subfile and the physical file.      *
     ************************************************/
    if ((pf = _Ropen(PFILENAME, "rr")) == NULL)  {
        printf("can't open file %s¥n", PFILENAME);
        exit(1);
    }
    if ((subf = _Ropen(SUBFILENAME, "ar+")) == NULL)  {
        printf("can't open file %s¥n", SUBFILENAME);
        exit(2);
    }
    /************************************************
     * Initialize the subfile with records          *
     * from the physical file.                      *
     ************************************************/
    init_subfile(pf, subf);
    /************************************************
     * Write the subfile to the display by writing  *
     * a record to the subfile control format.      *
     ************************************************/
     _Rformat(subf, "SFLCTL");
     _Rwrite(subf, "", 0);
     _Rreadnc(subf, "", 0);
    /************************************************
     * Close the physical file and the subfile.     *
     ************************************************/
    _Rclose(pf);
    _Rclose(subf);
}
```

関連情報

- 308 ページの『_Readd() ― 相対レコード番号によるレコードの読み取り』
- 310 ページの『_Rreadf() ― 最初のレコードの読み取り』
- 312 ページの『_Rreadindv() ― 送信勧誘された装置からの読み取り』
- 314 ページの『_Rreadk() ― キーによるレコードの読み取り』
- 318 ページの『_Rreadl() ― 最終レコードの読み取り』
- 319 ページの『_Rreadn() ― 次のレコードの読み取り』
- 324 ページの『_Rreadp() ― 前のレコードの読み取り』
- 326 ページの『_Rreads() ― 同じレコードの読み取り』

_Rreadp() ― 前のレコードの読み取り

フォーマット

```
#include <recio.h>

_RIOFB_T *_Rreadp(_RFILE *fp, void *buf, size_t size, int opts);
```

言語レベル: ILE C Extension

スレッド・セーフ: はい。ただし、スレッド間でファイル・ポインターが渡された場合、入出力フィードバック域はそれらのスレッド間で共有されます。

説明

`_Rreadp()` 関数は、*fp* に関連したファイルで現在使用されているアクセス・パス内の、前のレコードを読み取ります。アクセス・パスは、キー・シーケンスまたは到着順です。最大で *size* のバイト数が、レコードから *buf* にコピーされます (移動モードのみ)。ファイルが更新用にオープンされているときに __NO_LOCK が指定されない場合、`_Rreadp()` 関数は、位置指定されたレコードをロックします。

fp に関連したファイルが順次メンバー処理用にオープンされ、現行レコードの位置が、最初のレコードである場合 (但しファイル内の最初のメンバーは除く)、 `_Rreadp()` は、ファイルの以前のメンバー内の最後のレコードを読み取ります。

ファイルがレコード・ブロッキング用にオープンされ、`_Rreadn()` への呼び出しによってブロックが埋められた場合、 `_Rreadp()` 関数は、ブロック内にレコードが残っていると無効です。 _RIOFB_T 内の blk_count をチェックすると、残っているレコードがないかを確認できます。

以下に、`_Rreadp()` 関数の有効なパラメーターを示します。

buf
: 読み取ったデータを保管するバッファーを指定します。位置指定モードを使用する場合、このパラメーターは NULL に設定する必要があります。

size
: 読み取って *buf* に保管するバイト数を指定します。位置指定モードを使用する場合、このパラメーターは無視されます。

opts
: ファイルの処理オプションを指定します。指定できる値は、以下のとおりです。

  __DFT ファイルが更新用にオープンしている場合、読み取り中または位置指定中のレコードはロックされます。以前にロックしたレコードは、以後ロックされなくなります。

  __NO_LOCK
  位置指定されているレコードをロックしません。

`_Rreadp()` 関数は、データベースと DDM ファイルの場合に有効です。

戻り値

`_Rreadp()` 関数は、*fp* に関連した _RIOFB_T 構造体を指すポインターを戻します。 `_Rreadp()` 操作が正常終了した場合、num_bytes フィールドは、システム・バッファーからユーザーのバッファーに転送されたバイト数 (移動モード)、またはファイルのレコード長 (位置指定モード) に設定されます。 *key* および *rrn* フィールドも更新されます。レコード・ブロッキングが行われている場合は、 _RIOFB_T 構造体の blk_count および the blk_filled_by フィールドが更新されます。ファイル内の最初のレコードより前に読み取ろうとすると、num_bytes フィールドが EOF に設定されます。これが正常終了しなかった場合、num_bytes フィールドは、*size* より小さい値に設定され、errno が変更されます。

errno の値は、次のいずれかに設定されます。

値 **意味**

ENOTREAD
ファイルは読み取り操作用にオープンされていません。

ETRUNC
入出力操作で切り捨てが発生しました。

EIOERROR
リカバリー不能な入出力エラーが発生しました。

EIORECERR
リカバリー可能な入出力エラーが発生しました。

errno の設定については、 531 ページの表 12 および 535 ページの表 14 を参照してください。

_Rreadp() の使用例

```
#include <stdio.h>
#include <stdlib.h>
#include <recio.h>

int main(void)
{
    _RFILE      *fp;
    _XXOPFB_T   *opfb;

    /* Open the file for processing in arrival sequence.             */
    if (( fp = _Ropen ( "MYLIB/T1677RD1", "rr+, arrseq=Y" )) == NULL )
    {
        printf ( "Open failed¥n" );
        exit ( 1 );
    }

    /* Get the library and file names of the file opened.            */
    opfb = _Ropnfbk ( fp );
    printf ( "Library: %10.10s¥nFile:    %10.10s¥n",
              opfb->library_name,
              opfb->file_name);

    /* Get the last record.                                          */
    _Readl ( fp, NULL, 20, __DFT );
    printf ( "Last record: %10.10s¥n", *(fp->in_buf) );

    /* Get the previous record.                                      */

    _Rreadp ( fp, NULL, 20, __DFT );
    printf ( "Next to last record:  %10.10s¥n", *(fp->in_buf) );

    _Rclose ( fp );
}
```

関連情報

- 308 ページの『_Rreadd() — 相対レコード番号によるレコードの読み取り』
- 310 ページの『_Rreadf() — 最初のレコードの読み取り』
- 312 ページの『_Rreadindv() — 送信勧誘された装置からの読み取り』
- 314 ページの『_Rreadk() — キーによるレコードの読み取り』
- 318 ページの『_Rreadl() — 最終レコードの読み取り』

- 319 ページの『_Rreadn() ― 次のレコードの読み取り』
- 322 ページの『_Rreadnc() ― サブファイル内の次の変更済みレコードの読み取り』
- 『_Rreads() ― 同じレコードの読み取り』

_Rreads() ― 同じレコードの読み取り

フォーマット

```
#include <recio.h>

_RIOFB_T *_Rreads(_RFILE *fp, void *buf, size_t size, int opts);
```

言語レベル: ILE C Extension

スレッド・セーフ: はい。ただし、スレッド間でファイル・ポインターが渡された場合、入出力フィードバック域はそれらのスレッド間で共有されます。

説明

| `_Rreads()` 関数は、*fp* に関連したファイルの現在使用されているアクセス・パス内の現行レコードを読み
| 取ります。アクセス・パスは、キー・シーケンスまたは到着順です。最大で *size* のバイト数が、レコード
| から *buf* にコピーされます (移動モードのみ)。ファイルが更新用にオープンされているときに
| __NO_LOCK が指定されない場合、`_Rreads()` 関数は、位置指定されたレコードをロックします。

fp に関連したファイル内の現在位置に、それに関連したレコードがない場合、 `_Rreads()` は失敗します。

`_Rreads()` 関数は、ファイルがレコード・ブロッキング用にオープンしている場合は無効です。

以下に、`_Rreads()` 関数の有効なパラメーターを示します。

buf 読み取ったデータを保管するバッファーを指定します。位置指定モードを使用する場合、このパラメーターは NULL に設定する必要があります。

size 読み取って *buf* に保管するバイト数を指定します。位置指定モードを使用する場合、このパラメーターは無視されます。

opts ファイルの処理オプションを指定します。指定できる値は、以下のとおりです。

__DFT ファイルが更新用にオープンしている場合、読み取り中または位置指定中のレコードはロックされます。以前にロックしたレコードは、以後ロックされなくなります。

__NO_LOCK
位置指定されているレコードをロックしません。

`_Rreads()` 関数は、データベースと DDM ファイルの場合に有効です。

戻り値

`_Rreads()` 関数は、*fp* に関連した _RIOFB_T 構造体を指すポインターを戻します。 `_Rreads()` 操作が正常終了した場合、num_bytes フィールドは、システム・バッファーからユーザーのバッファーに転送されたバイト数 (移動モード)、またはファイルのレコード長 (位置指定モード) に設定されます。 *key* および *rrn* フィールドも更新されます。これが正常終了しなかった場合、num_bytes フィールドは、*size* より小さい値に設定され、errno が変更されます。

errno の値は、次のいずれかに設定されます。

値 **意味**

ENOTREAD
ファイルは読み取り操作用にオープンされていません。

ETRUNC
入出力操作で切り捨てが発生しました。

EIOERROR
リカバリー不能な入出力エラーが発生しました。

EIORECERR
リカバリー可能な入出力エラーが発生しました。

errno の設定については、 531 ページの表 12 および 535 ページの表 14 を参照してください。

_Rreads() の使用例

```
#include <stdlib.h>
#include <recio.h>

int main(void)
{
    _RFILE      *fp;
    _XXOPFB_T   *opfb;

    /* Open the file for processing in arrival sequence.             */
    if (( fp = _Ropen ( "MYLIB/T1677RD1", "rr+, arrseq=Y" )) == NULL )
    {
        printf ( "Open failed¥n" );
        exit ( 1 );
    }

    /* Get the library and file names of the file opened.            */
    opfb = _Ropnfbk ( fp );
    printf ( "Library: %10.10s¥nFile:    %10.10s¥n",
              opfb->library_name,
              opfb->file_name);

    /* Get the last record.                                          */
    _Readl ( fp, NULL, 20, __DFT );
    printf ( "Last record: %10.10s¥n", *(fp->in_buf) );

    /* Get the same record without locking it.                       */
    _Rreads ( fp, NULL, 20, __NO_LOCK);
    printf ( "Same record: %10.10s¥n", *(fp->in_buf) );

    _Rclose ( fp );
}
```

関連情報

- 308 ページの『_Rreadd() ― 相対レコード番号によるレコードの読み取り』
- 310 ページの『_Rreadf() ― 最初のレコードの読み取り』
- 312 ページの『_Rreadindv() ― 送信勧誘された装置からの読み取り』
- 314 ページの『_Rreadk() ― キーによるレコードの読み取り』
- 318 ページの『_Rreadl() ― 最終レコードの読み取り』
- 319 ページの『_Rreadn() ― 次のレコードの読み取り』
- 322 ページの『_Rreadnc() ― サブファイル内の次の変更済みレコードの読み取り』
- 324 ページの『_Rreadp() ― 前のレコードの読み取り』

_Rrelease() ― プログラム装置の解除

フォーマット

```
#include <recio.h>

int _Rrelease(_RFILE *fp, char *dev);
```

言語レベル: ILE C Extension

スレッド・セーフ: いいえ。

ジョブ CCSID インターフェース: この関数に送信される文字データは、すべてジョブの CCSID 内にあると想定されます。この関数によって戻された文字データは、すべてジョブの CCSID 内にあります。詳細については、 549 ページの『CCSID およびロケールの理解』を参照してください。

説明

_Rrelease() 関数は、*dev* で指定されたプログラム装置を、*fp* に関連したファイルから解放します。装置名は、大文字で指定してください。

dev パラメーターは、ヌル終了 C ストリングです。

_Rrelease() 関数は、ディスプレイ・ファイルと ICF ファイルの場合に有効です。

戻り値

_Rrelease() 関数は、正常終了した場合には 1、正常終了しなかった場合にはゼロを戻します。 errno の値は、EIOERROR (リカバリー不能な入出力エラーの発生) または EIORECERR (リカバリー可能な入出力エラーの発生) に設定されます。 errno の設定については、 531 ページの表 12 および 535 ページの表 14 を参照してください。

_Rrelease() の使用例

```
#include <stdio.h>
#include <recio.h>
#include <string.h>
#include <stdlib.h>
typedef struct {
    char name[20];
    char address[25];
} format1 ;
typedef struct {
    char name[8];
    char password[10];
} format2 ;
typedef union {
    format1 fmt1;
    format2 fmt2;
} formats ;

int main(void)
{
    _RFILE   *fp; /* File pointer                                     */
    _RIOFB_T *rfb; /*Pointer to the file's feedback structure         */
    _XXIOFB_T *iofb; /* Pointer to the file's feedback area           */
    formats  buf, in_buf, out_buf; /* Buffers to hold data            */
 /* Open the device file.                                          */
    if (( fp = _Ropen ( "MYLIB/T1677RD2", "ar+" )) == NULL )
    {
        printf ( "Could not open file¥n" );
```

```
        exit ( 1 );
    }
    _Racquire ( fp,"DEVICE1" );    /* Acquire another device. Replace */
                                   /* with actual device name.        */
    _Rformat ( fp,"FORMAT1" );     /* Set the record format for the   */
                                   /* display file.                   */
    rfb = _Rwrite ( fp, "", 0 );   /* Set up the display.             */
    _Rpgmdev ( fp,"DEVICE2" );  /* Change the default program device. */
                                /* Replace with actual device name.   */
    _Rformat ( fp,"FORMAT2" );   /* Set the record format for the     */
                                 /* display file.                     */
    rfb = _Rwrite ( fp, "", 0 );   /* Set up the display.             */
    rfb = _Rwriterd ( fp, &buf, sizeof(buf) );
    rfb = _Rwrread ( fp, &in_buf, sizeof(in_buf), &out_buf,
                     sizeof(out_buf ));
    _Rreadindv ( fp, &buf, sizeof(buf), __DFT );
                                  /* Read from the first device that  */
                                  /* enters data - device becomes     */
                                  /* default program device.          */
 /* Determine which terminal responded first.                      */
    iofb = _Riofbk ( fp );
    if ( !strncmp ( "FORMAT1  ", iofb -> rec_format, 10 ))
    {
        _Rrelease ( fp, "DEVICE1" );
    }
    else
    {
        _Rrelease(fp, "DEVICE2" );
    }
 /* Continue processing.                                           */
    printf ( "Data displayed is %45.45s¥n", &buf);
    _Rclose ( fp );
}
```

関連情報

- 269 ページの『_Racquire() ― プログラム装置の獲得』

_Rrlslck() ― レコード・ロックの解除

フォーマット

```
#include <recio.h>

int _Rrlslck(_RFILE *fp);
```

言語レベル: ILE C Extension

スレッド・セーフ: はい。

説明

`_Rrlslck()` 関数は、*fp* で指定されたファイルの現在ロックされているレコードに対するロックを解除します。ファイルは、更新のためにオープンされていなければなりません。また、レコードはロックされている必要があります。レコードをロックした `_Rlocate()` 操作に _NO_POSITION オプションが指定された場合、解放されたレコードは、現在位置指定されているレコードではない場合があります。

`_Rrlslck()` 関数は、データベースと DDM ファイルの場合に有効です。

戻り値

_Rrlslck() 関数は、操作が正常終了した場合には 1、操作が正常終了しなかった場合にはゼロを戻します。

errno の値は、次のいずれかに設定されます。

値 **意味**

ENOTUPD
ファイルがオープンされていないため、更新操作が行えません。

EIOERROR
リカバリー不能な入出力エラーが発生しました。

EIORECERR
リカバリー可能な入出力エラーが発生しました。

errno の設定については、 531 ページの表 12 および 535 ページの表 14 を参照してください。

_Rrlslck() の使用例

```
#include <stdio.h>
#include <stdlib.h>
#include <recio.h>

int main(void)
{
    char        buf[21];
    _RFILE      *fp;
    _XXOPFB_T   *opfb;
    int         result;

    /* Open the file for processing in arrival sequence.            */

    if (( fp = _Ropen ( "MYLIB/T1677RD1", "rr+, arrseq=Y" )) == NULL )
    {
        printf ( "Open failed¥n" );
        exit ( 1 );
    };

    /* Get the library and file names of the file opened.           */
    opfb = _Ropnfbk ( fp );
    printf ( "Library: %10.10s¥nFile:    %10.10s¥n",
             opfb->library_name,
             opfb->file_name);

    /* Get the last record.                                         */
    _Readl ( fp, NULL, 20, __DFT );
    printf ( "Last record: %10.10s¥n", *(fp->in_buf) );

    /* _Rrlslck example.                                            */
    result = _Rrlslck ( fp );
    if ( result == 0 )
       printf("_Rrlslck failed.¥n");

    _Rclose ( fp );
}
```

関連情報

- 273 ページの『_Rdelete() ― レコードの削除』

_Rrollbck() ― コミットメント制御の変更のロールバック

フォーマット

```
#include <recio.h>

int _Rrollbck(void);
```

言語レベル: ILE C Extension

スレッド・セーフ: いいえ。

説明

_Rrollbck() 関数は、最後のコミットメント境界を現行コミットメント境界として再設定します。ジョブ内のコミットメント制御下でファイルに加えられた変更は、すべて反転されます。ロック・レコードはすべて解除されます。ジョブ内のコミットメント制御の下でオープンしているファイルが影響を受けます。ファイルがコミットメント制御されるようにオープンしている場合は、キーワード・パラメーターを commit=y を指定する必要があります。コミットメント制御環境は、この前にすでにセットアップされていなければなりません。

_Rrollbck() 関数は、データベースと DDM ファイルの場合に有効です。

戻り値

_Rrollbck() 関数は、操作が正常終了した場合には 1、操作が正常終了しなかった場合にはゼロを戻します。 errno の値は、EIOERROR (リカバリー不能な入出力エラーの発生) または EIORECERR (リカバリー可能な入出力エラーの発生) に設定されます。 errno の設定については、 531 ページの表 12 および 535 ページの表 14 を参照してください。

_Rrollbck() の使用例

```
#include <stdio.h>
#include <recio.h>
#include <stdlib.h>
#include <string.h>

int main(void)
{
  char      buf[40];
  int       rc = 1;
  _RFILE    *purf;
  _RFILE    *dailyf;

  /* Open purchase display file and daily transaction file         */
  if ( ( purf = _Ropen ( "MYLIB/T1677RD3", "ar+,indicators=y" )) == NULL )
  {
      printf ( "Display file did not open.¥n" );
      exit ( 1 );
  }

  if ( ( dailyf = _Ropen ( "MYLIB/T1677RDA", "wr,commit=y") ) == NULL )
  {
      printf ( "Daily transaction file did not open.¥n" );
      exit ( 2 );
  }

  /* Select purchase record format */
  _Rformat ( purf, "PURCHASE" );

  /* Invite user to enter a purchase transaction.                   */
```

```
  /* The _Rwrite function writes the purchase display.              */
  _Rwrite ( purf, "", 0 );
  _Rreadn ( purf, buf, sizeof(buf), __DFT );

  /* Update daily transaction file                                  */
  rc = (( _Rwrite ( dailyf, buf, sizeof(buf) ))->num_bytes );

  /* If the databases were updated, then commit the transaction.    */
  /* Otherwise, rollback the transaction and indicate to the        */
  /* user that an error has occurred and end the application.       */
  if ( rc )
    {
        _Rcommit ( "Transaction complete" );
    }
  else
    {
        _Rrollbck ( );
        _Rformat ( purf, "ERROR" );
    }

  _Rclose ( purf );
  _Rclose ( dailyf );
}
```

関連情報

- 271 ページの『_Rcommit() ― 現行レコードのコミット』
- *Recovering your system*のマニュアル

_Rupdate() ― レコードの更新

フォーマット

```
#include <recio.h>

_RIOFB_T *_Rupdate(_RFILE *fp, void *buf, size_t size);
```

言語レベル: ILE C Extension

スレッド・セーフ: はい。ただし、スレッド間でファイル・ポインターが渡された場合、入出力フィードバック域はそれらのスレッド間で共有されます。

説明

`_Rupdate()` 関数は、*fp* で指定されたファイル内で更新のために現在ロックされているレコードを更新します。ファイルは更新のためにオープンしていなければなりません。レコードは、読み取りまたは位置指定オプションで __NO_LOCK が指定されない場合、その読み取りまたは位置指定によって、更新のためにロックされます。位置指定操作で __NO_POSITION オプションが指定された場合、更新されたレコードは、そのファイルが現在位置指定されているレコードではない場合があります。更新操作の後、更新されたレコードは、以後ロックされなくなります。

buf からレコードにコピーされたバイト数は、 *size* の最小値とファイルのレコード長です (移動モードのみ)。 *size* がレコード長より大きい場合、データは切り捨てられ、errno が ETRUNC に設定されます。常に 1 つの完全なレコードがファイルに書き込まれます。 *size* がファイルのレコード長より小さい場合、レコード内の残りのデータは、レコードをロックした読み取りによって、システム・バッファーに読み取られた元のデータになります。位置指定オプションがレコードをロックした場合、残りのデータは、位置指定の前にシステム入力バッファー内にあったものになります。

_Rupdate() 関数を使用して、削除済みレコードと任意のキー・フィールドを更新できます。更新される削除済みレコードは、以後削除済みレコードとしてマークされなくなります。これらの両方の場合に、*fp* 用に定義されたキー順アクセス・パスが変更されます。

注: 位置指定モードが使用されている場合、_Rupdate() は、ファイルの入力バッファー内のデータに対して機能します。

_Rupdate() 関数は、データベース、ディスプレイ (サブファイル) および DDM ファイルの場合に有効です。

戻り値

_Rupdate() 関数は、*fp* に関連した _RIOFB_T 構造体を指すポインターを戻します。 _Rupdate() 操作が正常終了した場合、num_bytes フィールドは、システム・バッファーからユーザーのバッファーに転送されたバイト数 (移動モード)、またはファイルのレコード長 (位置指定モード) に設定されます。 *fp* がディスプレイ・ファイルである場合は、sysparm フィールドが更新されます。 _Rupdate() 関数が正常終了しなかった場合、 num_bytes フィールドは、指定された *size* より小さい値 (移動モード) またはゼロ (位置指定モード) に設定されます。 errno 値も変更されます。

errno の値は、次のいずれかに設定されます。

値 **意味**

ENOTUPD
ファイルがオープンされていないため、更新操作が行えません。

EIOERROR
リカバリー不能な入出力エラーが発生しました。

EIORECERR
リカバリー可能な入出力エラーが発生しました。

errno の設定については、 531 ページの表 12 および 535 ページの表 14 を参照してください。

_Rupdate() の使用例

```
#include <stdio.h>
#include <stdlib.h>
#include <recio.h>

int main(void)
{
    _RFILE   *in;
    char     new_purchase[21] = "PEAR      1002022244";

    /* Open the file for processing in keyed sequence.             */

    if ( (in = _Ropen("MYLIB/T1677RD4", "rr+, arrseq=N")) == NULL )
    {
        printf("Open failed¥n");
        exit(1);
    };

    /* Update the first record in the keyed sequence.              */

    _Rlocate(in, NULL, 0, __FIRST);
    _Rupdate(in, new_purchase, 20);

    /* Force the end of data.                                      */

    _Rfeod(in);

    _Rclose(in);
}
```

関連情報

- 308 ページの『_Readd() ― 相対レコード番号によるレコードの読み取り』
- 310 ページの『_Readf() ― 最初のレコードの読み取り』
- 312 ページの『_Readindv() ― 送信勧誘された装置からの読み取り』
- 314 ページの『_Readk() ― キーによるレコードの読み取り』
- 318 ページの『_Readl() ― 最終レコードの読み取り』
- 319 ページの『_Readn() ― 次のレコードの読み取り』
- 322 ページの『_Readnc() ― サブファイル内の次の変更済みレコードの読み取り』
- 324 ページの『_Readp() ― 前のレコードの読み取り』
- 326 ページの『_Reads() ― 同じレコードの読み取り』

_Rupfb() ― 最終入出力操作についての情報の説明

フォーマット

```
#include <recio.h>

_RIOFB_T *_Rupfb(_RFILE *fp);
```

言語レベル: ILE C Extension

スレッド・セーフ: はい。ただし、スレッド間でファイル・ポインターが渡された場合、入出力フィードバック域はそれらのスレッド間で共有されます。

説明

`_Rupfb()` 関数は、*fp* で指定されたファイルに関連したフィードバック構造体を、最後の入出力操作に関する情報で更新します。 _RIOFB_T 構造体は、ファイルのオープン時に riofb=N が指定された場合であって

も更新されます。 _RIOFB_T 構造体の num_bytes フィールドは、更新されません。 _RIOFB_T 構造体の説明については、 10 ページの『<recio.h>』を参照してください。

_Rupfb() 関数は、すべてのタイプのファイルの場合に有効です。

戻り値

_Rupfb() 関数は、*fp* で指定された _RIOFB_T 構造体を指すポインターを戻します。 errno の設定については、 531 ページの表 12 および 535 ページの表 14 を参照してください。

_Rupfb() の使用例

```
#include <stdio.h>
#include <recio.h>
#include <stdlib.h>

int main(void)
{
    _RFILE   *fp;
    _RIOFB_T *fb;
    /* Create a physical file                                          */
    system("CRTPF FILE(QTEMP/MY_FILE) RCDLEN(80)");
    /* Open the file for write                                         */
    if ( (fp = _Ropen("QTEMP/MY_FILE", "wr")) == NULL )
    {
        printf("open for write fails¥n");
        exit(1);
    }
    /* Write some records into the file                                */
    _Rwrite(fp, "This is record 1", 16);
    _Rwrite(fp, "This is record 2", 16);
    _Rwrite(fp, "This is record 3", 16);
    _Rwrite(fp, "This is record 4", 16);
    _Rwrite(fp, "This is record 5", 16);
    _Rwrite(fp, "This is record 6", 16);
    _Rwrite(fp, "This is record 7", 16);
    _Rwrite(fp, "This is record 8", 16);
    _Rwrite(fp, "This is record 9", 16);
    /* Close the file                                                  */
    _Rclose(fp);
    /* Open the file for read                                          */
    if ( (fp = _Ropen("QTEMP/MY_FILE", "rr, blkrcd = y")) == NULL )
    {
        printf("open for read fails¥n");
        exit(2);
    }
    /* Read some records                                               */
    _Rreadn(fp, NULL, 80, __DFT);
    _Rreadn(fp, NULL, 80, __DFT);
    /* Call _Rupfb and print feed back information                     */
    fb = _Rupfb(fp);
    printf("record number -------------------------- %d¥n",
           fb->rrn);
    printf("number of bytes read ------------------- %d¥n",
           fb->num_bytes);
    printf("number of records remaining in block --- %hd¥n",
           fb->blk_count);
    if ( fb->blk_filled_by == __READ_NEXT )
    {
        printf("block filled by ------------------------ __READ_NEXT¥n");
    }
    else
    {
        printf("block filled by ------------------------ __READ_PREV¥n");
```

```
    }
    /* Close the file                                              */
    _Rclose(fp);
}
```

関連情報

- 305 ページの『_Ropnfbk() ― オープン・フィードバック情報の取得』

_Rwrite() ― 次のレコードの書き込み

フォーマット

```
#include <recio.h>

_RIOFB_T * _Rwrite(_RFILE *fp, void *buf, size_t size);
```

言語レベル: ILE C Extension

スレッド・セーフ: はい。ただし、スレッド間でファイル・ポインターが渡された場合、入出力フィードバック域はそれらのスレッド間で共有されます。

説明

`_Rwrite()` 関数には、移動と位置指定の 2 つのモードがあります。 *buf* がユーザー・バッファーを指す場合、`_Rwrite()` は移動モードです。 *buf* が NULL である場合、関数は位置指定モードです。

`_Rwrite()` 関数は、*fp* で指定されたファイルにレコードを付加します。 *buf* からレコードにコピーされたバイト数は、 *size* の最小値とファイルのレコード長です (移動モードのみ)。サイズがレコード長より大きい場合、データは切り捨てられ、errno が ETRUNC に設定されます。操作が正常終了した場合、必ず 1 つの完全なレコードが書き込まれます。

`_Ropen()` を使用してから `_Rwrite()` を使用してレコードをソース物理ファイルに出力する場合、シーケンス番号を手動で付加する必要があります。

`_Rwrite()` 関数は、以後の読み取り操作のファイルの位置には影響しません。

以下の項目に該当する場合、`_Rwrite()` 関数が正常終了を示していても、レコードが失われる場合があります。

- レコード・ブロッキングが行われている。
- *fp* に関連したファイルに含めることができるレコード数が限界に近づいていて、かつファイルを拡張できない。
- 複数のライターが同じファイルに書き込んでいる。

出力がバッファーに入れられると、`_Rwrite` ルーチンは正常終了を戻します。これは、レコードが正常にバッファーにコピーされたことを示します。ただし、バッファーがフラッシュされた場合、ルーチンは失敗します。これは、別のライターによって、ファイルの容量がいっぱいになってしまったためです。この場合、`_Rwrite()` 関数は、ファイルにデータを送信する `_Rwrite()` 関数への呼び出しでのみエラーが発生したことを示します。

`_Rwrite()` 関数は、すべてのタイプのファイルの場合に有効です。

戻り値

_Rwrite() 関数は、*fp* に関連した _RIOFB_T 構造体を指すポインターを戻します。 _Rwrite() 操作が正常終了した場合、移動モードと位置指定モードの両方で、num_bytes フィールドは書き込まれたバイト数に設定されます。この関数は、ユーザーのバッファーからシステム・バッファーにバイトを転送します。レコード・ブロッキングが行われている場合、この関数は、ブロックをデータベースに送信する際に *rrn* フィールドと *key* フィールドのみを更新します。 *fp* がディスプレイ・ファイル、ICF ファイル、またはプリンター・ファイルである場合、この関数は sysparm フィールドを更新します。この関数が正常終了しなかった場合、num_bytes フィールドは、指定された *size* より小さい値 (移動モード) またはゼロ (位置指定モード) に設定され、 errno が変更されます。

errno の値は、次のいずれかに設定されます。

値　　意味

ENOTWRITE
ファイルは書き込み操作用にオープンされません。

ETRUNC
入出力操作で切り捨てが発生しました。

EIOERROR
リカバリー不能な入出力エラーが発生しました。

EIORECERR
リカバリー可能な入出力エラーが発生しました。

errno の設定については、 531 ページの表 12 および 535 ページの表 14 を参照してください。

_Rwrite() の使用例

```
#include <stdio.h>
#include <recio.h>
#include <string.h>
#include <stdlib.h>
typedef struct {
    char name[20];
    char address[25];
} format1 ;
typedef struct {
    char name[8];
    char password[10];
} format2 ;
typedef union {
    format1 fmt1;
    format2 fmt2;
} formats ;

int main(void)
{
    _RFILE   *fp; /* File pointer                                     */
    _RIOFB_T *rfb; /*Pointer to the file's feedback structure         */
    _XXIOFB_T *iofb; /* Pointer to the file's feedback area           */
    formats  buf, in_buf, out_buf; /* Buffers to hold data            */
 /* Open the device file.                                          */
    if (( fp = _Ropen ( "MYLIB/T1677RD2", "ar+" )) == NULL )
    {
        printf ( "Could not open file¥n" );
        exit ( 1 );
    }
    _Racquire ( fp,"DEVICE1" );    /* Acquire another device. Replace */
                                   /* with actual device name.        */
    _Rformat ( fp,"FORMAT1" );     /* Set the record format for the   */
                                   /* display file.                   */
```

```
    rfb = _Rwrite ( fp, "", 0 );   /* Set up the display.            */
    _Rpgmdev ( fp,"DEVICE2" );  /* Change the default program device. */
                               /* Replace with actual device name.   */
    _Rformat ( fp,"FORMAT2" );   /* Set the record format for the     */
                                /* display file.                      */
    rfb = _Rwrite ( fp, "", 0 );   /* Set up the display.            */
    rfb = _Rwriterd ( fp, &buf, sizeof(buf) );
    rfb = _Rwrread ( fp, &in_buf, sizeof(in_buf), &out_buf,
                     sizeof(out_buf ));
   _Rreadindv ( fp, &buf, sizeof(buf), __DFT );
                                  /* Read from the first device that  */
                                  /* enters data - device becomes     */
                                  /* default program device.          */
 /* Determine which terminal responded first.                     */
    iofb = _Riofbk ( fp );
    if ( !strncmp ( "FORMAT1  ", iofb -> rec_format, 10 ))
    {
        _Rrelease ( fp, "DEVICE1" );
    }
    else
    {
        _Rrelease(fp, "DEVICE2" );
    }
 /* Continue processing.                                          */
    printf ( "Data displayed is %45.45s¥n", &buf);
    _Rclose ( fp );
}
```

関連情報

- 『_Rwrited() ― レコード・ディレクトリーの書き込み』
- 341 ページの『_Rwriterd() ― レコードの書き込みと読み取り』
- 342 ページの『_Rwrread() ― レコードの書き込みと読み取り (分離バッファー)』

_Rwrited() ― レコード・ディレクトリーの書き込み

フォーマット

```
#include <recio.h>

_RIOFB_T *_Rwrited(_RFILE *fp, void *buf, size_t size, unsigned long rrn);
```

言語レベル: ILE C Extension

スレッド・セーフ: はい。ただし、スレッド間でファイル・ポインターが渡された場合、入出力フィードバック域はそれらのスレッド間で共有されます。

説明

`_Rwrited()` 関数は、*rrn* で指定された位置で、*fp* に関連したファイルにレコードを書き込みます。`_Rwrited()` 関数は、削除されたレコードのみを上書きします。 *buf* からレコードにコピーされたバイト数は、 *size* の最小値とファイルのレコード長です (移動モードのみ)。サイズがレコード長より大きい場合、データは切り捨てられ、errno が ETRUNC に設定されます。操作が正常終了した場合、必ず 1 つの完全なレコードが書き込まれます。

`_Rwrited()` 関数は、読み取り操作の場合、ファイルの位置には影響しません。

`_Rwrited()` 関数は、データベース、DDM およびサブファイルの場合に有効です。

戻り値

`_Rwrited()` 関数は、*fp* に関連した _RIOFB_T 構造体を指すポインターを戻します。 `_Rwrited()` 操作が正常終了した場合、num_bytes フィールドは、ユーザーのバッファーからシステム・バッファーに転送されたバイト数 (移動モード)、またはファイルのレコード長 (位置指定モード) に設定されます。 *rrn* フィールドは更新されます。 *fp* がディスプレイ・ファイルである場合は、sysparm フィールドが更新されます。この関数が正常終了しなかった場合、num_bytes フィールドは、指定された *size* より小さい値 (移動モード) またはゼロ (位置指定モード) に設定され、 errno が変更されます。

errno の値は、次のいずれかに設定されます。

値 **意味**

ENOTWRITE
ファイルは書き込み操作用にオープンされません。

ETRUNC
入出力操作で切り捨てが発生しました。

EIOERROR
リカバリー不能な入出力エラーが発生しました。

EIORECERR
リカバリー可能な入出力エラーが発生しました。

errno の設定については、 531 ページの表 12 および 535 ページの表 14 を参照してください。

`_Rwrited()` の使用例

```
#include <stdio.h>
#include <stdlib.h>
#include <recio.h>
#define LEN          10
#define NUM_RECS     20
#define SUBFILENAME  "MYLIB/T1677RD6"
#define PFILENAME    "MYLIB/T1677RDB"
typedef struct  {
    char name[LEN];
    char phone[LEN];
} pf_t;
#define RECLEN sizeof(pf_t)
void init_subfile(_RFILE *, _RFILE *);
int main(void)
{
    _RFILE          *pf;
    _RFILE          *subf;
 /* Open the subfile and the physical file.       */
    if ((pf = _Ropen(PFILENAME, "rr")) == NULL)  {
        printf("can't open file %s¥n", PFILENAME);
        exit(1);
    }
    if ((subf = _Ropen(SUBFILENAME, "ar+")) == NULL)  {
        printf("can't open file %s¥n", SUBFILENAME);
        exit(2);
    }
 /* Initialize the subfile with records           *
  * from the physical file.                       */
    init_subfile(pf, subf);
 /* Write the subfile to the display by writing   *
  * a record to the subfile control format.       */
     _Rformat(subf, "SFLCTL");
     _Rwrite(subf, "", 0);
     _Rreadnc(subf, "", 0);
 /* Close the physical file and the subfile.      */
    _Rclose(pf);
    _Rclose(subf);
}
void init_subfile(_RFILE *pf, _RFILE *subf)
    {
        _RIOFB_T      *fb;
        int           i;
        pf_t          record;
 /* Select the subfile record format.             */
        _Rformat(subf, "SFL");
        for (i = 1; i <= NUM_RECS; i++)  {
            fb = _Rreadn(pf, &record, RECLEN, __DFT);
            if (fb->num_bytes != RECLEN)  {
            printf("%d¥n", fb->num_bytes);
            printf("%d¥n", RECLEN);
                printf("error occurred during read¥n");
                exit(3);
            }
            fb = _Rwrited(subf, &record, RECLEN, i);
            if (fb->num_bytes != RECLEN)  {
                printf("error occurred during write¥n");
                exit(4);
            }
        }
    }
```

関連情報

- 336 ページの『_Rwrite() ― 次のレコードの書き込み』
- 341 ページの『_Rwriterd() ― レコードの書き込みと読み取り』
- 342 ページの『_Rwrread() ― レコードの書き込みと読み取り (分離バッファー)』

_Rwriterd() — レコードの書き込みと読み取り

フォーマット

```
#include <recio.h>
_RIOFB_T *_Rwriterd(_RFILE *fp, void *buf, size_t size);
```

言語レベル: ILE C Extension

スレッド・セーフ: いいえ。

説明

`_Rwriterd()` 関数は、*fp* で指定されたファイルについて、書き込みの後に読み取り操作を行います。現行レコード形式の最小サイズと長さによって、システム・バッファーと、操作の読み取りおよび書き込みの両方の部分用の *buf* の間でコピーされるデータの量が決まります。 *size* が現行形式のレコード長より大きい場合、操作の書き込み部分に関する errno は ETRUNC に設定されます。 *size* が現行レコード形式の長さより小さい場合、操作の読み取り部分に関する errno は ETRUNC に設定されます。

`_Rwriterd()` 関数は、ディスプレイ・ファイルと ICF ファイルの場合に有効です。

戻り値

`_Rwriterd()` 関数は、*fp* に関連した _RIOFB_T 構造体を指すポインターを戻します。 `_Rwriterd()` 操作が正常終了した場合、num_bytes フィールドは、システム・バッファーから操作の読み取り部分の *buf* に転送されたバイト数 (移動モード)、またはファイルのレコード長 (位置指定モード) に設定されます。

errno の値は、次のいずれかに設定されます。

値 **意味**

ENOTUPD
ファイルがオープンされていないため、更新操作が行えません。

ETRUNC
入出力操作で切り捨てが発生しました。

EIOERROR
リカバリー不能な入出力エラーが発生しました。

EIORECERR
リカバリー可能な入出力エラーが発生しました。

errno の設定については、 531 ページの表 12 および 535 ページの表 14 を参照してください。

`_Rwriterd()` の使用例

```
#include <stdio.h>
#include <recio.h>
#include <string.h>
#include <stdlib.h>

typedef struct {
    char name[20];
    char address[25];
} format1 ;

typedef struct {
    char name[8];
    char password[10];
} format2 ;

typedef union {
    format1 fmt1;
    format2 fmt2;
} formats ;

int main(void)
{
    _RFILE   *fp; /* File pointer                                      */
    _RIOFB_T *rfb; /*Pointer to the file's feedback structure          */
    formats  buf, in_buf, out_buf; /* Buffers to hold data             */

    /* Open the device file.                                           */
    if (( fp = _Ropen ( "MYLIB/T1677RD2", "ar+" )) == NULL )
    {
        printf ( "Could not open file¥n" );
        exit ( 1 );
    }

    _Rpgmdev ( fp,"DEVICE2" );/* Change the default program device.   */
                              /* Replace with actual device name.     */

    _Rformat ( fp,"FORMAT2" );   /* Set the record format for the     */
                                 /* display file.                     */

    rfb = _Rwrite ( fp, "", 0 );   /* Set up the display.              */
    rfb = _Rwriterd ( fp, &buf, sizeof(buf) );

    rfb = _Rwrread ( fp, &in_buf, sizeof(in_buf), &out_buf,
                     sizeof(out_buf ));

    /* Continue processing.                                            */


    _Rclose ( fp );
}
```

関連情報

- 336 ページの『_Rwrite() ― 次のレコードの書き込み』
- 338 ページの『_Rwrited() ― レコード・ディレクトリーの書き込み』
- 『_Rwrread() ― レコードの書き込みと読み取り (分離バッファー)』

_Rwrread() ― レコードの書き込みと読み取り (分離バッファー)

フォーマット

```
#include <recio.h>

_RIOFB_T *_Rwrread(_RFILE *fp, void *in_buf, size_t in_buf_size,
                   void *out_buf, size_t out_buf_size);
```

言語レベル: ILE C Extension

スレッド・セーフ: いいえ。

説明

`_Rwrread()` 関数は、*fp* で指定されたファイルについて、書き込みの後に読み取り操作を行います。入力データと出力データ用に、別々のバッファーが指定されます。現行レコード形式の最小サイズと長さによって、システム・バッファーと、読み取りおよび書き込みの操作用のバッファーの間でコピーされるデータの量が決まります。 *out_buf_size* が現行形式のレコード長より大きい場合、操作の書き込み部分に関する errno は ETRUNC に設定されます。 *in_buf_size* が現行レコード形式の長さより小さい場合、操作の読み取り部分に関する errno は ETRUNC に設定されます。

`_Rwrread()` 関数は、ディスプレイ・ファイルと ICF ファイルの場合に有効です。

戻り値

`_Rwrread()` 関数は、*fp* に関連した _RIOFB_T 構造体を指すポインターを戻します。 `_Rwrread()` 操作が正常終了した場合、num_bytes フィールドは、システム・バッファーから操作の読み取り部分の *in_buf* に転送されたバイト数 (移動モード)、またはファイルのレコード長 (位置指定モード) に設定されます。

errno の値は、次のいずれかに設定されます。

値　　意味

ENOTUPD
: ファイルがオープンされていないため、更新操作が行えません。

ETRUNC
: 入出力操作で切り捨てが発生しました。

EIOERROR
: リカバリー不能な入出力エラーが発生しました。

EIORECERR
: リカバリー可能な入出力エラーが発生しました。

errno の設定については、 531 ページの表 12 および 535 ページの表 14 を参照してください。

`_Rwrread()` の使用例

```
#include <stdio.h>
#include <recio.h>
#include <string.h>
#include <stdlib.h>

typedef struct {
    char name[20];
    char address[25];
} format1 ;

typedef struct {
    char name[8];
    char password[10];
} format2 ;

typedef union {
    format1 fmt1;
    format2 fmt2;
} formats ;

int main(void)
{
    _RFILE   *fp; /* File pointer                                      */
    _RIOFB_T *rfb; /*Pointer to the file's feedback structure          */
    formats  buf, in_buf, out_buf; /* Buffers to hold data             */

    /* Open the device file.                                           */
    if (( fp = _Ropen ( "MYLIB/T1677RD2", "ar+" )) == NULL )
    {
        printf ( "Could not open file¥n" );
        exit ( 1 );
    }

    _Rpgmdev ( fp,"DEVICE2" );/* Change the default program device.   */
                              /* Replace with actual device name.     */

    _Rformat ( fp,"FORMAT2" );   /* Set the record format for the     */
                                 /* display file.                     */

    rfb = _Rwrite ( fp, "", 0 );   /* Set up the display.             */
    rfb = _Rwriterd ( fp, &buf, sizeof(buf) );

    rfb = _Rwrread ( fp, &in_buf, sizeof(in_buf), &out_buf,
                     sizeof(out_buf ));

    /* Continue processing.                                            */

    _Rclose ( fp );
}
```

関連情報

- 336 ページの『_Rwrite() ― 次のレコードの書き込み』
- 338 ページの『_Rwrited() ― レコード・ディレクトリーの書き込み』
- 341 ページの『_Rwriterd() ― レコードの書き込みと読み取り』

scanf() ― データの読み取り

フォーマット

```
#include <stdio.h>
int scanf(const char *format-string, argument-list);
```

言語レベル: ANSI

スレッド・セーフ: はい。

ロケール依存: この関数の振る舞いは、現行ロケールの LC_CTYPE カテゴリーおよび LC_NUMERIC カテゴリーの影響を受ける可能性があります。また、この振る舞いは、LOCALETYPE(*LOCALEUCS2) または LOCALETYPE(*LOCALEUTF) がコンパイル・コマンドに対して指定されている場合は、現行ロケールの LC_UNI_CTYPE カテゴリーの影響を受ける可能性もあります。詳細については、 549 ページの『CCSID およびロケールの理解』を参照してください。

説明

`scanf()` 関数は、標準入力ストリーム `stdin` から *argument-list* 内の各エントリーで指定された位置へ、データを読み取ります。各 *argument* は、 *format-string* 内の型指定子に対応する型の変数を指すポインターでなければなりません。 *format-string* は、入力フィールドの変換処理を制御し、始まりと終わりが初期シフト状態のマルチバイト文字ストリングです。

format-string には、以下の 1 つ以上が含まれます。

- `isspace()` 関数で指定された空白文字 (ブランクおよび改行文字など)。空白文字の場合、`scanf()` 関数は、空白ではない次の文字まで、入力内の連続したすべての空白文字を読み取りますが、保管はしません。 *format-string* 内の空白文字 1 文字は、入力データ内の空白文字を複数組み合わせたものに相当します。
- 空白ではない文字。ただし、パーセント記号文字 (%) は除く。空白ではない文字の場合、 `scanf()` 関数は、一致する空白ではない文字を読み取りますが、保管はしません。 `stdin` の次の文字が一致しない場合、`scanf()` 関数は終了します。
- パーセント記号 (%) で始まる形式指定。形式指定によって、`scanf()` 関数は、入力された文字を読み取り、指定された型の値に変換します。値は、引数リスト内の引数へ割り当てられます。

`scanf()` 関数は、*format-string* を左から右へ読み取ります。形式指定以外の文字は、`stdin` 内の文字シーケンスと一致することを想定しています。 `stdin` 内の一致した文字は走査されますが、保管はされません。 `stdin` 内の文字が *format-string* と矛盾する場合、`scanf()` は終了します。矛盾する文字は、読み取られなかった場合と同様に、`stdin` に残ります。

最初の形式指定が見つかると、最初の入力フィールドの値が形式指定に従って変換され、 *argument-list* 内の最初の項目で指定された場所に保管されます。 2 つ目の形式指定によって、2 つ目の入力フィールドが変換され、 *argument-list* の 2 番目のエントリーに保管されます。この作業は、*format-string* の最後まで行われます。

入力フィールドは、最初の空白文字 (スペース、タブ、または改行) まで、形式指定に従って変換できない最初の文字まで、もしくはフィールド *width* に達するまでのすべての文字として定義されます。この内、最初に登場するものが優先されます。形式指定のための引数が多すぎる場合、余分な引数は無視されます。形式指定に対して引数が足りない場合は、予期しない結果が引き起こされることになります。

形式指定の形式は、以下のとおりです。

形式指定の各フィールドは、単一文字または数字で、特殊な形式オプションを示します。最後のオプションの形式フィールドの後に表示される *type* 文字は、入力フィールドが文字、ストリング、または数字として解釈されるかどうかを判別するものです。最もシンプルな形式指定には、パーセント記号と *type* 文字のみが含まれます (例えば、%s)。

形式指定の各フィールドについては、以下で詳細に説明します。パーセント記号 (%) の後に、形式制御文字として意味を持たない文字が続いた場合、その文字とその後に続くパーセント記号までの文字は、通常の文字シーケンス、つまり、入力と一致しなければならない文字のシーケンスとして処理されます。例えば、パーセント記号文字を指定するには、%% を使用します。

ポインターの出力と走査には、次の制約事項が適用されます。

- ポインターを出力し、同じ活動化グループから走査し直した場合、走査し直したポインターは、比較により、出力したポインターと等しくなります。
- `scanf()` ファミリー関数が異なる活動化グループによって出力されたポインターを走査する場合、`scanf()` ファミリー関数はポインターを NULL に設定します。

i5/OS ポインターの使用法についての詳細は、「*IBM Rational Development Studio for i: ILE C/C++ Programmer's Guide*」を参照してください。

パーセント記号の後のアスタリスク (*) は、指定された *type* のフィールドとして解釈される、次の入力フィールドの割り当てを抑止します。フィールドは走査されますが、保管はされません。

width は正の 10 進整数で、`stdin` から読み取られる文字の最大数を制御します。 *width* を超える文字は変換されず、対応する *引数* へも保管されません。空白文字 (スペース、タブ、または改行)、または指定された形式に従って変換できない文字が、 *width* に達する前に発生すると、*width* より少ない文字が読み取られます。

オプションのサイズ修飾子 h、l、ll、L、H、D、および DD は、受信側のオブジェクトのサイズを示します。対応する引数が、整数を指すポインターではなく short 型整数を指すポインターである場合、型変換文字 d、i、および n の前には、h がなければなりません。また、long 型整数を指すポインターである場合は l、 long long 型整数を指すポインターである場合は ll がなければなりません。同様に、対応する引数が、符号なし int を指すポインターではなく、符号なし short 型整数を指すポインターである場合、型変換文字 o、u、x、および X の前には、h がなければなりません。また、符号なし long 型整数を指すポインターである場合は l、符号なし long long 型整数を指すポインターである場合は ll がなければなりません。対応する引数が、浮動小数点ではなく double を指すポインターである場合、型変換文字 e、E、f、F、g、および G の前には、l がなければなりません。また、long double を指すポインターである場合は L、 _Decimal32 を指すポインターである場合は H、 _Decimal64 を指すポインターである場合は D、あるいは _Decimal128 を指すポインターである場合は DD がなければなりません。最後に、対応する引数が、1 バイト文字型を指すポインターではなく wchar_t を指すポインターである場合、型変換文字 c、s、および [の前には、l がなければなりません。 h、l、L、ll、H、D、または DD が他の型変換文字とともに表示される場合、その動作は予期できません。

次の表に、*type* 文字とその意味を示します。

文字	入力される型	引数の型
d	符号付き 10 進整数	int を指すポインター。
o	符号なし 8 進整数	符号なし int を指すポインター。
x、X	符号なし 16 進整数	符号なし int を指すポインター。
i	10 進、16 進、または 8 進整数	int を指すポインター。
u	符号なし 10 進整数。	符号なし int を指すポインター。
e、E、 f、F、 g、G	浮動小数点値。これは、符号 (+ または -)、1 つ以上の 10 進数字の並び (小数点を含む場合がある)、およびオプションの指数 (e または E) とそれに続く符号付き整数値から構成されます。	浮動小数点を指すポインター。
D(n,p)	パック 10 進数。これは、オプションの記号 (+ または -) と、その後に続く一連の空でない桁、オプションの 1 つ以上の 10 進数字の並び (小数点を含む場合がある) で構成されます。ただし、10 進の接尾部は含まれません。サブジェクト・シーケンスは、最初の非空白文字から始まる、入力ストリングの最長の初期サブシーケンスとして定義されています。これが想定される形式です。入力ストリングが空であるか、すべて空白文字から成っているか、あるいは最初の非空白文字が符号、数字、または 10 進小数点文字以外である場合には、サブジェクト・シーケンスには文字は含まれません。	decimal(n,p) を指すポインター。 2 進化 10 進数オブジェクトの内部表記は、パック 10 進数データ型の内部表記と同じであるため、型文字 D(n,p) を使用できます。
c	文字。 c が指定されると、通常スキップされる空白文字が読み取られます。	入力フィールドに十分な大きさの char を指すポインター。
s	ストリング	入力フィールドに加えて、自動的に付加される終了ヌル文字 (¥0) を入れるのに十分な大きさの文字配列を指すポインター。
n	*stream* またはバッファーから読み取られる入力はありません。	int を指すポインター。ここに、`scanf()` を呼び出した点までの文字数、 *stream* またはバッファーから正常に読み取られた文字数が保管されます。
p	一連の文字に変換される void を指すポインター。	void へのポインター
lc	マルチバイト文字定数	wchar_t を指すポインター。
ls	マルチバイト・ストリング定数	wchar_t ストリングを指すポインター。

スペース文字で区切られていないストリングを読み取るには、大括弧 ([ ]) 内の文字を *s* (ストリング) 型文字で置換します。対応する入力フィールドは、括弧内の文字セットで現れない最初の文字まで、読み取られます。セット内の最初の文字が脱字記号 (^) である場合は、結果は反転されます。入力フィールドは、残りの文字セット内に現れる最初の文字まで読み取られます。

終了ヌル文字 (¥0) を保管しないで、ストリングを保管するには、%*a*c をしてください。ここで、*a* は 10 進整数です。このインスタンスでは、c 型文字は、引数が文字配列を指すポインターであることを意味します。次の *a* 文字が、入力ストリームから指定された場所へ読み取られ、ヌル文字は追加されません。

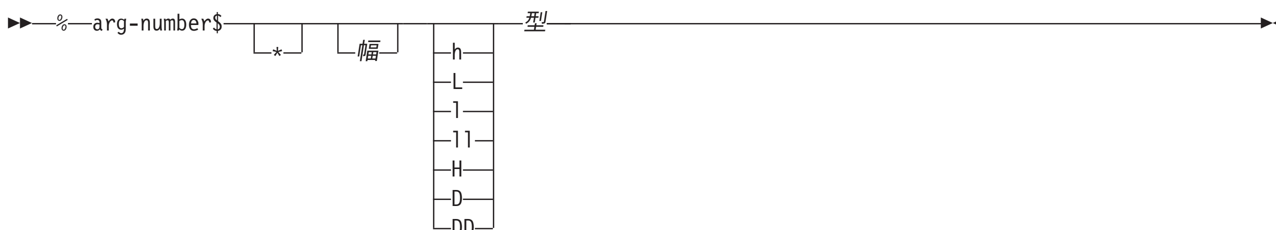
%*x* 形式指定子の入力は、16 進数として解釈されます。

scanf() 関数は、各入力フィールドを文字ごとに走査します。指定された *width* に達したか、または次の文字が指定されたように変換できない場合は、空白文字に達する前に、特定の入力フィールドを読み取るのを停止します。指定と入力文字との間に矛盾が生じた場合は、次の入力フィールドは、まだ読み取られていない最初の文字で始まります。矛盾した文字 (もしあった場合) は、読み取られていない文字と見なされ、次の入力フィールドの最初の文字、または stdin での次の読み取り操作での最初の文字となります。

%lc および %ls の場合は、読み取られるデータを指定しますが、これはマルチバイト・ストリングで、mbtowc への呼び出しの場合と同様に、ワイド文字に変換されます。

%e、%E、%f、%F、%g、および %G の各形式指定子の場合、文字シーケンス INFINITY または NAN (大/小文字は無視されます) が許可され、それぞれ、値 INFINITY または静止非数 (NaN) が生成されます。

代替の形式指定の形式は、以下のとおりです。



代替として、前述のダイアグラムで概要が示された形式指定を使用して、引数リスト内の特定のエントリーを割り当てることも可能です。この形式指定と以前の形式指定は、同じ scanf() への呼び出しで混用しないようにします。そうでないと、予期不能な結果になる場合があります。

arg-number は正整数定数で、この場合、1 は引数リスト内の最初のエントリーを示します。 arg-number は、引数リスト内のエントリー数より多くならないようにします。そうでないと、予期しない結果になります。 arg-number はまた、NL_ARGMAX より多くならないようにします。

戻り値

scanf() 関数は、正常に変換され、割り当てられたフィールドの数を戻します。戻り値には、読み取りは行われたが、割り当てられなかったフィールドは含まれません。

変換が行われていない場合に、ファイルの終わりを読み取ろうとすると、戻り値は EOF になります。戻り値 0 は、フィールドが割り当てられなかったことを意味します。

エラー条件

割り当てられる引数の型が形式指定と異なる場合は、予測不能な結果になる可能性があります。例えば、浮動小数点値を読み取り、それを型 int の変数に割り当てることは誤りであり、予測不能な結果になります。

形式指定にあるよりも多くの引数が存在する場合は、余分な引数は無視されます。形式指定に対して引数が足りない場合は、予期しない結果が引き起こされることになります。

書式ストリングに無効な形式指定が含まれている場合に、定位置形式指定が使用されていると、errno が EILSEQ に設定されます。

定位置形式指定が使用され、十分な引数がない場合、errno は EINVAL に設定されます。

変換エラーが発生した場合、errno は **ECONVERT** に設定される可能性があります。

scanf() の使用例

この例では、さまざまなタイプのデータを走査します。

```
#include <stdio.h>

int main(void)
{
   int i;
   float fp;
   char c, s[81];

   printf("Enter an integer, a real number, a character "
          "and a string : ¥n");
   if (scanf("%d %f %c %s", &i, &fp, &c, s) != 4)
      printf("Not all fields were assigned¥n");
   else
   {
      printf("integer = %d¥n", i);
      printf("real number = %f¥n", fp);
      printf("character = %c¥n", c);
      printf("string = %s¥n",s);
   }
}

/*****************  If input is: 12 2.5 a yes,  *******************
**************  then output should be similar to:  ****************

Enter an integer, a real number, a character and a string :
integer = 12
real number = 2.500000
character = a
string = yes
*/
```

この例では、16 進整数を 10 進整数に変換します。 while ループは、入力値が 16 進整数でない場合に終了します。

```
#include <stdio.h>

int main(void)
{
   int number;

   printf("Enter a hexadecimal number or anything else to quit:¥n");
   while (scanf("%x",&number))
      {
      printf("Hexadecimal Number = %x¥n",number);
      printf("Decimal Number     = %d¥n",number);
      }
}

/***************  If input is: 0x231 0xf5e 0x1 q,  **************      **
****************  then output should be similar to:  **************

Enter a hexadecimal number or anything else to quit:
Hexadecimal Number = 231
Decimal Number     = 561
Hexadecimal Number = f5e
Decimal Number     = 3934
Hexadecimal Number = 1
Decimal Number     = 1
*/
```

この例では、stdin から読み取り、代替の定位置書式ストリングを使用してデータを割り当てます。

```
#include <stdio.h>
int main(int argc, char *argv[])
{
int i;
char s[20];
float f;

scanf("%2$s %3$f %1$d",&i, s, &f);

printf("The data read was %i\n%s\n%f\n,i,s,f);

return 0;
 }

 /* If the input is : test 0.2 100
    then the output will be similar to: */
   The data read was
 100
 test
 0.20000

 --------------------------------------------------------------------
```

この例では、マルチバイト文字ストリングをワイド Unicode ストリングに読み取ります。この例は、LOCALETYPE(*LOCALEUCS2) または LOCALETYPE(*LOCALEUTF) のいずれかによってコンパイルできます。

```
 #include <locale.h>
 #include <stdio.h>
 #include <wchar.h>

 void main(void)
{
wchar_t uString[20];

setlocale(LC_UNI_ALL, "");
scanf("Enter a string %ls",uString);

printf("String read was %ls\n",uString);
}

/* if the input is : ABC
   then the output will be similiar to:

   String read was ABC

 */
```

関連情報

- 138 ページの『fscanf() ― フォーマット済みデータの読み取り』
- 238 ページの『printf() ― 定様式の文字の出力』
- 371 ページの『sscanf() ― データの読み取り』
- 528 ページの『wscanf() ― ワイド文字書式ストリングを使用したデータの読み取り』
- 153 ページの『fwscanf() ― ワイド文字を使用したストリームからのデータの読み取り』
- 425 ページの『swscanf() ― ワイド文字データの読み取り』
- 17 ページの『<stdio.h>』

setbuf() ― バッファリングの制御

フォーマット

```
#include <stdio.h>
void setbuf(FILE *, char *buffer);
```

言語レベル: ANSI

スレッド・セーフ: はい。

説明

オペレーティング・システムがユーザー定義のバッファーをサポートしている場合、 `setbuf()` は指定された *stream* のバッファリングを制御します。 `setbuf()` 関数は、統合ファイル・システムを使用する場合、ILE C でのみ機能します。 *stream* ポインターは、入出力または位置変更を行う前に、オープン・ファイルを指す必要があります。

buffer 引数が NULL の場合、*stream* はバッファーに入れられません。そうでない場合、*buffer* は、長さが BUFSIZ の文字配列を指している必要があります。これは、<stdio.h> インクルード・ファイルで定義されたバッファー・サイズです。システムは、指定された *stream* のためのデフォルト・システム割り振りバッファーの代わりに、入出力バッファリングのための *buffer* (ユーザーが指定) を使用します。 `stdout`、`stderr`、および `stdin` は、ユーザー定義のバッファーをサポートしません。

`setvbuf()` 関数は、`setbuf()` 関数より柔軟です。

戻り値

戻り値はありません。

setbuf() の使用例

この例では、書き込みのためにファイル setbuf.dat をオープンします。その後、setbuf() 関数を呼び出して、長さが BUFSIZ のバッファーを設定します。ストリームに string が書き込まれると、バッファー buf が使用されます。このバッファーには、それがファイルにフラッシュされるまで、ストリングが含まれます。

```
#include <stdio.h>

int main(void)
{
   char buf[BUFSIZ];
   char string[] = "hello world";
   FILE *stream;

   memset(buf,'¥0',BUFSIZ);  /* initialize buf to null characters */

   stream = fopen("setbuf.dat", "wb");

   setbuf(stream,buf);       /* set up buffer */

   fwrite(string, sizeof(string), 1, stream);

   printf("%s¥n",buf);       /* string is found in buf now */

   fclose(stream);           /* buffer is flushed out to myfile.dat */
}
```

関連情報

- 95 ページの『fclose() ― ストリームのクローズ』
- 101 ページの『fflush() ― ファイルへのバッファーの書き込み』
- 114 ページの『fopen() ― ファイルのオープン』
- 360 ページの『setvbuf() ― バッファリングの制御』
- 17 ページの『<stdio.h>』

setjmp() ― 環境の保存

フォーマット

```
#include <setjmp.h>
int setjmp(jmp_buf env);
```

言語レベル: ANSI

スレッド・セーフ: はい。

説明

setjmp() 関数は、スタック環境を保存します。これは、その後 longjmp() 関数により復元できます。setjmp() および longjmp() 関数は、非ローカル goto を実行する方法を提供します。これらは、シグナル・ハンドラーでよく使用されます。

setjmp() 関数を呼び出すと、現行スタック環境が、*env* に保存されることになります。以後の longjmp() 関数への呼び出しにより、保存済み環境が復元されて、制御は setjmp() 呼び出しに対応するポイントへ戻されます。制御を受信する関数に使用可能な (レジスター変数を除いた) すべての変数の値に、 longjmp() 関数の呼び出し時に保持していた値が含まれています。レジスター変数の値は、予測不可能です。setjmp() 関数と longjmp() 関数の間の呼び出しで変更される、不揮発性 auto 変数も予測不可能です。

戻り値

setjmp() 関数は、スタック環境を保管してから、0 を戻します。 longjmp() 呼び出しの結果として setjmp() 関数が戻ると、 setjmp() 関数は longjmp() 関数の *value* 引数を戻すか、 longjmp() 関数の *value* 引数が 0 の場合は、1 を戻します。エラーの戻り値はありません。

setjmp() の使用例

この例では、下のステートメントで、スタック環境を保存します。

```
if(setjmp(mark) != 0) ...
```

システムは、最初に if ステートメントを実行する場合、環境を mark に保管し、条件を FALSE に設定します。これは、setjmp() 関数が環境を保管するとき 0 を戻すためです。プログラムで次のメッセージが出力されます。

```
setjmp has been called
```

関数 p() への以降の呼び出しで、longjmp() 関数が呼び出されます。制御は、*mark* 変数内に保存された環境を使用して、setjmp() 関数への呼び出しの直後に main() 関数内のポイントに渡されます。この場合、条件は TRUE ですが、これは戻り値がスタック上に置かれるよう、longjmp() 関数呼び出しの 2 番目のパラメーターで -1 が指定されているためです。次に、この例ではブロック中のステートメントが実行され、"longjmp() has been called" というメッセージが出力されます。その後、recover() 関数が実行され、プログラムが終了します。

```
#include <stdio.h>
#include <setjmp.h>
#include <stdlib.h>

jmp_buf mark;

void p(void);
void recover(void);

int main(void)
{
   if (setjmp(mark) != 0)
   {
      printf("longjmp has been called¥n");
      recover();
      exit(1);
      }
   printf("setjmp has been called¥n");
   printf("Calling function p()¥n");
   p();
   printf("This point should never be reached¥n");
}

void p(void)
{
   printf("Calling longjmp() from inside function p()¥n");
   longjmp(mark, -1);
   printf("This point should never be reached¥n");
}

void recover(void)
{
   printf("Performing function recover()¥n");
}
/*******************Output should be as follows: **********************
 setjmp has been called
 Calling function p()
 Calling longjmp() from inside function p()
 longjmp has been called
 Performing function recover()
**********************************************************************/
```

関連情報

- 201 ページの『longjmp() ― スタック環境の復元』
- 14 ページの『<setjmp.h>』

setlocale() ― ロケールの設定

フォーマット

```
#include <locale.h>
char *setlocale(int category, const char *locale);
```

言語レベル: ANSI

スレッド・セーフ: いいえ。

ロケール依存: 詳細については、 549 ページの『CCSID およびロケールの理解』を参照してください。

説明

`setlocale()` 関数は、<locale.h> インクルード・ファイルで定義された、ロケーションを示す変数を変更または照会します。以下に、*category* の値をリストします。

カテゴリー	目的
LC_ALL	プログラムのロケール全体を指定します。
LC_COLLATE	`strcoll()` および `strxfrm()` 関数の振る舞いに影響を与えます。
LC_CTYPE	文字処理関数の振る舞いに影響を与えます。
LC_MONETARY	`localeconv()` および `nl_langinfo()` 関数によって戻される通貨情報に影響を与えます。
LC_NUMERIC	以下のような定様式入出力の小数点文字およびストリング変換関数、ならびに `localeconv()` および `nl_langinfo()` 関数によって戻される非通貨フォーマット設定情報に影響を与えます。
LC_TIME	`strftime()` 関数の振る舞い、および `nl_langinfo()` 関数によって戻される時刻形式設定情報に影響を与えます。
LC_TOD	時間関数の振る舞いに影響を与えます。 カテゴリー LC_TOD には、複数のフィールドがあります。 TNAME フィールドは時間帯名です。 TZDIFF フィールドは、地方時とグリニッジ標準時との差です。 TNAME フィールドが非ブランクの場合、一部の時間関数によって戻された値を判別する際に TZDIFF フィールドが使用されます。この値は、システム値 QUTCOFFSET に優先します。
LC_UNI_ALL*	このカテゴリーによって、`setlocale()` は指定されたロケールからすべての LC_UNI_ カテゴリーをロードします。このカテゴリーは、UCS-2 または UTF-32 の CCSID のロケールのみを受け入れます。
LC_UNI_COLLATE*	`wcscoll()` および `wcsxfrm()` 関数の振る舞いに影響を与えます。このカテゴリーは、UCS-2 または UTF-32 の CCSID のロケールのみを受け入れます。 **注:** このカテゴリーは、UCS-2 ではサポートされません。
LC_UNI_CTYPE*	ワイド文字処理関数の振る舞いに影響を与えます。このカテゴリーは、UCS-2 または UTF-32 の CCSID のロケールのみを受け入れます。
LC_UNI_MESSAGES*	`_WCS_nl_langinfo()` 関数によって戻されるメッセージ・フォーマット設定情報に影響を与えます。このカテゴリーは、UCS-2 または UTF-32 の CCSID のロケールのみを受け入れます。
LC_UNI_MONETARY*	`wcslocaleconv()` および `_WCS_nl_langinfo()` 関数によって戻される通貨情報に影響を与えます。このカテゴリーは、UCS-2 または UTF-32 の CCSID のロケールのみを受け入れます。
LC_UNI_NUMERIC*	以下のようなワイド文字定様式入出力の小数点文字およびワイド文字ストリング変換関数、ならびに `wcslocaleconv()` および `_WCS_nl_langinfo()` 関数によって戻される非通貨フォーマット設定情報に影響を与えます。このカテゴリーは、UCS-2 または UTF-32 の CCSID のロケールのみを受け入れます。
LC_UNI_TIME*	`wcsftime()` 関数の振る舞い、および `_WCS_nl_langinfo()` 関数によって戻される時刻形式情報に影響を与えます。このカテゴリーは、UCS-2 または UTF-32 の CCSID のロケールのみを受け入れます。
LC_UNI_TOD*	ワイド文字時間関数の振る舞いに影響を与えます。このカテゴリーは、UCS-2 または UTF-32 の CCSID のロケールのみを受け入れます。
*	名前にカテゴリーと UNI を使用するには、コンパイル・コマンドで LOCALETYPE(*LOCALEUCS2) または LOCALETYPE(*LOCALEUTF) を指定する必要があります。 LOCALETYPE(*LOCALEUCS2) を使用する場合、指定されたロケールは UCS-2 ロケールでなければなりません。 LOCALETYPE(*LOCALEUTF) を使用する場合、指定されたロケールは UTF-32 ロケールでなければなりません。

注: System i® プラットフォームで `setlocale()` とその他のロケール依存 C 関数を定義する場合は、 2 とおりの方法があります。 `setlocale()` を定義する 1 つ目の方法は、*CLD ロケール・オブジェクトを使用してロケールを設定し、ロケール依存データを検索するという方法です。 `setlocale()` を定義する 2 つ目の方法は、*LOCALE オブジェクトを使用してロケールを設定し、ロケール依存データを検索するという方法です。元の方法には、コンパイル・コマンドで LOCALETYPE(*CLD) を指定してアクセスします。 2 番目の方法には、コンパイル・コマンドで LOCALETYPE(*LOCALE) 、LOCALETYPE(*LOCALEUCS2)、または LOCALETYPE(*LOCALEUTF) を指定してアクセスします。 ILE C でのロケール定義における 2 つの方法についての詳細は、「*IBM Rational Development Studio for i: ILE C/C++ Programmer's Guide*」の『International Locale Support』のセクションを参照してください。

***CLD ロケール・オブジェクトを使用した Setlocale**

locale の値は、"C"、""、LC_C、 LC_C_GERMANY、LC_C_FRANCE、LC_C_SPAIN、LC_C_ITALY、LC_C_USA または LC_C_UK に設定できます。 *locale* 値 ″C″ は、デフォルトの C 環境を示します。 *locale* 値 "" は、 `setlocale()` 関数での実装にデフォルト・ロケールを使用することを示します。

***LOCALE オブジェクトによる Setlocale**

locale の値は、""、"C"、"POSIX"、または *LOCALE オブジェクトの完全修飾された統合ファイル・システム・パス名 (二重引用符で囲む) に設定できます。*locale* 値 "C" または "POSIX" は、デフォルトの C *LOCALE オブジェクトを示します。 *locale* 値 "" は、 `setlocale()` 関数でプロセスにデフォルト・ロケールを使用することを示します。

プロセスのデフォルト・ロケールは、次の表を使用して決定します。

LC_ALL	1. LC_ALL 環境変数[1] を確認します。この環境変数が定義済みで非ヌルの場合は、すべての POSIX ロケール・カテゴリーに、指定されたロケール[2] を使用します。それ以外の場合は、次のステップに進みます。 2. 各 POSIX ロケール・カテゴリー (LC_CTYPE、LC_COLLATE、LC_TIME、LC_NUMERIC、LC_MESSAGES、LC_MONETARY、および LC_TOD) について、同じ名前[1] を持つ環境変数を確認します。この環境変数が定義済みで非ヌルの場合は、指定されたロケール[2] を使用します。 3. LANG 環境変数[1] を確認します。前のステップで設定されなかった各ロケール・カテゴリーごとに、 LANG 環境変数が定義済みで非ヌルの場合は、ロケール・カテゴリーを指定されたロケール[2] に設定します。それ以外の場合は、それをデフォルトの C *LOCALE オブジェクトに設定します。

LC_CTYPE LC_COLLATE LC_TIME LC_NUMERIC LC_MESSAGES LC_MONETARY LC_TOD	1. LC_ALL 環境変数[1] を確認します。この環境変数が定義済みで非ヌルの場合は、指定されたロケール[2] を使用します。それ以外の場合は、次のステップに進みます。 2. 指定されたロケール・カテゴリーと同じ名前を持つ環境変数[1] を確認します。この環境変数が定義済みで非ヌルの場合は、指定されたロケール[2] を使用します。それ以外の場合は、次のステップに進みます。 3. LANG 環境変数[1] を確認します。この環境変数が定義済みで非ヌルの場合は、ロケール・カテゴリーを指定されたロケール[2] に設定します。それ以外の場合は、次のステップに進みます。 4. ロケール・カテゴリーをデフォルトの C *LOCALE オブジェクトに設定します。
LC_UNI_ALL	ユーザーのモジュールが LOCALETYPE(*LOCALEUCS2) オプションによってコンパイルされている場合は、以下のようにします。 1. LC_UCS2_ALL 環境変数[1] を確認します。この環境変数が定義済みで非ヌルの場合は、すべての Unicode ロケール・カテゴリーに、指定されたロケールを使用します。それ以外の場合は、次のステップに進みます。 2. 各 Unicode ロケール・カテゴリーごとに、対応する環境変数[1] (LC_UCS2_CTYPE、 LC_UCS2_COLLATE、 LC_UCS2_TIME、 LC_UCS2_NUMERIC、 LC_UCS2_MESSAGES、 LC_UCS2_MONETARY、または LC_UCS2_TOD)[3] を確認します。この環境変数が定義済みで非ヌルの場合は、指定されたロケールを使用します。 3. ロケール・カテゴリーをデフォルトの UCS-2 *LOCALE オブジェクトに設定します。 ユーザーのモジュールが LOCALETYPE(*LOCALEUTF) オプションによってコンパイルされている場合は、以下のようにします。 1. LC_UTF_ALL 環境変数[1] を確認します。この環境変数が定義済みで非ヌルの場合は、すべての Unicode ロケール・カテゴリーに、指定されたロケールを使用します。それ以外の場合は、次のステップに進みます。 2. 各 Unicode ロケール・カテゴリーごとに、対応する環境変数[1] (LC_UTF_CTYPE、 LC_UTF_COLLATE、LC_UTF_TIME、LC_UTF_NUMERIC、 LC_UTF_MESSAGES、LC_UTF_MONETARY、または LC_UTF_TOD)[3] を確認します。この環境変数が定義済みで非ヌルの場合は、指定されたロケールを使用します。 3. LANG 環境変数[1] を確認します。前のステップで設定されなかった各ロケール・カテゴリーごとに、 LANG 環境変数が定義済みで非ヌルの場合は、ロケール・カテゴリーを指定されたロケールに設定します。それ以外の場合は、それをデフォルトの UTF *LOCALE オブジェクトに設定します。

LC_UNI_CTYPE LC_UNI_COLLATE LC_UNI_TIME LC_UNI_NUMERIC LC_UNI_MESSAGES LC_UNI_MONETARY LC_UNI_TOD	ユーザーのモジュールが LOCALETYPE(*LOCALEUCS2) オプションによってコンパイルされている場合は、以下のようにします。 1. 指定されたロケール・カテゴリー[1] (LC_UCS2_CTYPE、LC_UCS2_COLLATE、LC_UCS2_TIME、LC_UCS2_NUMERIC、LC_UCS2_MESSAGES、LC_UCS2_MONETARY、または LC_UCS2_TOD)[3] に対応する環境変数を確認します。この環境変数が定義済みで非ヌルの場合は、指定されたロケールを使用します。それ以外の場合は、次のステップに進みます。 2. LC_UCS2_ALL 環境変数[1] を確認します。この環境変数が定義済みで非ヌルの場合は、指定されたロケールを使用します。それ以外の場合は、次のステップに進みます。 3. ロケール・カテゴリーをデフォルトの UCS-2 *LOCALE オブジェクトに設定します。 ユーザーのモジュールが LOCALETYPE(*LOCALEUTF) オプションによってコンパイルされている場合は、以下のようにします。 1. 指定されたロケール・カテゴリー[1] (LC_UTF_CTYPE、LC_UTF_COLLATE、LC_UTF_TIME、LC_UTF_NUMERIC、LC_UTF_MESSAGES、LC_UTF_MONETARY、または LC_UTF_TOD)[3] に対応する環境変数を確認します。この環境変数が定義済みで非ヌルの場合は、指定されたロケールを使用します。それ以外の場合は、次のステップに進みます。 2. LC_UTF_ALL 環境変数[1] を確認します。この環境変数が定義済みで非ヌルの場合は、指定されたロケールを使用します。それ以外の場合は、次のステップに進みます。 3. LANG 環境変数[1] を確認します。 LANG 環境変数が定義済みで非ヌルの場合は、ロケール・カテゴリーを指定されたロケールに設定します。それ以外の場合は、それをデフォルトの UTF *LOCALE オブジェクトに設定します。

注: [1] ロケール・カテゴリーに対応する名前を持つ環境変数は、ユーザーによって作成されます。 LANG 環境変数は、以下のいずれかのロケール・パス名を指定する際、ジョブ開始時に自動的に作成されます。

- ユーザー・プロファイル内の LOCALE パラメーター (i5/OS Information Center の『CHGUSRPRF (ユーザー・プロファイル変更) コマンド』の情報を参照)。
- QLOCALE システム値 (i5/OS Information Center の『QLOCALE システム値』の情報を参照)。

ロケール環境変数には、 /QSYS.LIB/<locname>.LOCALE または /QSYS.LIB/<libname>.LIB/<locname>.LOCALE という形式のロケール・パス名が含まれると想定されます。ユーザーのモジュールが LOCALETYPE(*LOCALEUTF) オプションによってコンパイルされている場合、環境変数は、パスの <locname> 部分が 8 文字を超えると無視されます。このような制約があるのは、対応する UTF ロケールの名前を取得する場合に、ロケール名に 2 文字の接尾部を追加する必要があるためです。

注: [2] コンパイル・コマンドで LOCALETYPE(*LOCALEUTF) が指定された場合、 setlocale() 関数は、LC_ALL、LC_CTYPE、LC_COLLATE、LC_TIME、LC_NUMERIC、LC_MESSAGES、LC_MONETARY、 LC_TOD、および LANG 環境変数に、後書きの _8 を追加します。このロケールが見つからない場合は、UTF デフォルト・ロケール・オブジェクトが使用されます。例えば、LANG が /QSYS.LIB/EN_US.LOCALE に設定されている場合に setlocale(LC_ALL、"") を指定すると、setlocale() によってロケール /QSYS.LIB/EN_US_8.LOCALE のロードが試行されます。

LANG 環境変数を使用していずれかの Unicode ロケール・カテゴリー (LC_UNI_ALL、LC_UNI_CTYPE、LC_UNI_COLLATE、LC_UNI_TIME、LC_UNI_NUMERIC、LC_UNI_MESSAGES、 LC_UNI_MONETARY、または LC_UNI_TOD) を設定した場合、 setlocale() は、環境変数に保管されたロケール名の最後に _4 を追加します。これは、対応する UTF-32 ロケールを見つける試みです。このロケールが見つからない場合は、デフォルトの UTF-32 ロケール・オブジェクトが使用されます。例えば、LANG が /QSYS.LIB/EN_US.LOCALE に設定されている場合に setlocale(LC_UNI_TIME、"") を指定すると、setlocale() によってロケール /QSYS.LIB/EN_US_4.LOCALE のロードが試行されます。 _4 および _8 で終わるロケール名は、 CCSID(*UTF) で作成されたロケールの場合には、 CRTLOCALE CL コマンド (i5/OS Information Center のトピック『CRTLOCALE (ロケール作成) コマンド』の情報を参照) によって導入された命名規則に従っています。

注: [3] LC_UNI_ALL、LC_UNI_COLLATE、LC_UNI_CTYPE、LC_UNI_TIME、LC_UNI_NUMERIC、LC_UNI_MESSAGES、LC_UNI_MONETARY、および LC_UNI_TOD の各ロケール・カテゴリー名は、UCS-2 と UTF の間で共用されます。これらのカテゴリーに対応する環境変数は、共用できません。したがって、環境変数の名前は、ロケール・カテゴリー名と完全には一致しません。 UCS-2 環境変数名の場合は、UNI が UCS2 で置換されます (例えば、 LC_UNI_ALL ロケール・カテゴリーは LC_UCS2_ALL 環境変数になります)。 UTF 環境変数の場合は、UNI が UTF で置換されます (例えば、 LC_UNI_ALL ロケール・カテゴリーは LC_UTF_ALL 環境変数になります)。

LOCALETYPE(*LOCALEUCS2) または LOCALETYPE(*LOCALEUTF) でコンパイルされた場合、ロケールは、LC_UNI_ で始まるカテゴリーの有効な Unicode ロケールを指すポインターでなければならず、その他のカテゴリーの Unicode ロケールであってはなりません。

戻り値

setlocale() 関数は、現行ロケールの設定を表すストリングを指すポインターを戻します。戻されたストリングが保管された場合、保管されたストリング値を setlocale() 関数への入力として使用し、ロケール設定を随時復元することができます。ただし、ストリングをユーザー定義のバッファーにコピーする必要があります。そうしないと、以後 setlocale() を呼び出したときに、ストリングが上書きされます。

注: setlocale() への正常な呼び出しによって指されたストリングが、以後の setlocale() 関数への呼び出しによって上書きされることがあるため、後でそのストリングを使用する予定がある場合は、これをコピーしてください。ロケール・ストリングの正確な形式は、ロケール・タイプ *CLD、*LOCALE、*LOCALEUCS2、および *LOCALEUTF の間で異なります。

ロケールを照会するには、NULL を 2 番目のパラメーターとして指定してください。例えば、ユーザーのロケールのすべてのカテゴリーを照会するには、次のステートメントを入力します。

```
char *string = setlocale(LC_ALL, NULL);
```

エラー条件

エラーの場合、setlocale() 関数は NULL を戻します。プログラムのロケールは変更されません。

*CLD ロケール・オブジェクトの使用例

```
/************************************************************************

This example sets the locale of the program to
LC_C_FRANCE *CLD and prints the string
that is associated with the locale. This example must be compiled with
the LOCALETYPE(*CLD) parameter on the compilation command.
```

```
  *
 ***********************************************************************/

#include <stdio.h>
#include <locale.h>

char *string;

int main(void)
{
   string = setlocale(LC_ALL, LC_C_FRANCE);
   if (string != NULL)
      printf(" %s ¥n",string);
}
```

*LOCALE オブジェクトの使用例

```
/***********************************************************************

This example sets the locale of the program to be "POSIX" and prints
the string that is associated with the locale.  This example must be
compiled with the LOCALETYPE(*LOCALE) parameter on the CRTCMOD or
CRTBNDC command.

 ***********************************************************************/

#include <stdio.h>
#include <locale.h>

char *string;

int main(void)
{

   string = setlocale(LC_ALL, "POSIX");
   if (string != NULL)
      printf(" %s ¥n",string);

}
```

関連情報

- 160 ページの『getenv() — 環境変数の検索』
- 187 ページの『localeconv() — 環境からの情報の取得』
- 234 ページの『nl_langinfo() — ロケール情報の検索』
- 8 ページの『<locale.h>』

setvbuf() — バッファリングの制御

フォーマット

```
#include <stdio.h>
int setvbuf(FILE *stream, char *buf, int type, size_t size);
```

言語レベル: ANSI

スレッド・セーフ: はい。

説明

`setvbuf()` 関数を使用すると、指定したストリームのバッファリング戦略およびバッファー・サイズを制御できます。 `setvbuf()` 関数は、統合ファイル・システムを使用する場合、ILE C でのみ機能します。ストリームは、オープンされているが、読み取りや書き込みが行われていないファイルを示す必要があります。

buf が指す配列は、ストリーム用のバッファーとして使用するために C ライブラリーが選択できる、ユーザー提供のエリアを指定します。 *buf* 値が NULL である場合は、そのようなエリアが提供されていないため、ストリーム用の独自のバッファーの管理を、C ライブラリーが担当することを示しています。バッファーを提供した場合には、ストリームがクローズされるまで、存在していることが必要です。

type は、以下のいずれかにする必要があります。

値　　意味

_IONBF
: バッファーが使用されていません。

_IOFBF
: フル・バッファリングは、入出力のために使用されます。*buf* をバッファーとして、*size* をバッファーのサイズとして使用してください。

_IOLBF
: 行バッファリングが使用されます。バッファーが削除されるのは、改行文字が書き込まれるとき、バッファーがいっぱいのとき、または入力が要求されるときです。

type が _IOFBF または _IOLBF の場合、*size* は提供されたバッファーのサイズです。*buf* が NULL の場合には、 C ライブラリーでは、*size* がそれ自体のバッファーの提示サイズと見なされます。 *type* が _IONBF の場合、*buf* と *size* は両方とも無視されます。

size の値は、0 より大きいことが必要です。

戻り値

`setvbuf()` 関数は、正常に実行された場合には 0 を戻します。パラメーター・リストで無効な値が指定されたか、要求を実行できない場合には、この関数はゼロ以外を戻します。

`setvbuf()` 関数は、`stdout`、`stdin`、または `stderr` には影響しません。

重要: バッファーとして使用される配列は、指定された *stream* がクローズしても、引き続き存在しています。例えば、バッファーが関数ブロックのスコープ内で宣言される場合には、 *stream* は、関数の終了前にクローズし、バッファーに割り振られたストレージを解放する必要があります。

`setvbuf()` の使用例

この例では、stream1 のバッファー buf をセットアップし、 stream2 への入力をバッファーに入れないことを指定します。

```
#include <stdio.h>

#define  BUF_SIZE  1024

char buf[BUF_SIZE];
FILE *stream1, *stream2;

int main(void)
{
   stream1 = fopen("myfile1.dat", "r");
   stream2 = fopen("myfile2.dat", "r");

   /* stream1 uses a user-assigned buffer of BUF_SIZE bytes */
   if (setvbuf(stream1, buf, _IOFBF, sizeof(buf)) != 0)
      printf("Incorrect type or size of buffer¥n");

   /* stream2 is unbuffered                                  */
   if (setvbuf(stream2, NULL, _IONBF, 0) != 0)
      printf("Incorrect type or size of buffer¥n");

/*  This is a program fragment and not a complete function example  */

}
```

関連情報

- 95 ページの『fclose() ― ストリームのクローズ』
- 101 ページの『fflush() ― ファイルへのバッファーの書き込み』
- 114 ページの『fopen() ― ファイルのオープン』
- 351 ページの『setbuf() ― バッファリングの制御』
- 17 ページの『<stdio.h>』

signal() ― 割り込みシグナルの処理

フォーマット

```
#include <signal.h>
void ( *signal (int sig, void(*func)(int)) )(int);
```

言語レベル: ANSI

スレッド・セーフ: はい。

説明

signal() 関数を使用すると、プログラムは、オペレーティング・システムまたは raise() 関数からの割り込みシグナルを処理するいくつかの方法のうち 1 つを選択できるようになります。 SYSIFCOPT (*ASYNCSIGNAL) オプションでコンパイルされた場合、この関数は非同期シグナルを使用します。この関数の非同期バージョンは、sigaction() に SA_NODEFER および SA_RESETHAND オプションを指定した場合のように動作します。非同期シグナル・ハンドラーは、abort() や exit() を呼び出しません。この関数についての残りの説明は、同期シグナルに関するものです。

sig 引数は、マクロ
SIGABRT、SIGALL、SIGILL、SIGINT、SIGFPE、SIGIO、SIGOTHER、SIGSEGV、SIGTERM、SIGUSR1、または SIGUSR2 のいずれかでなければなりません。これらは、signal.h インクルード・ファイルに定義されています。SIGALL、SIGIO、および SIGOTHER は、ILE C/C++ ランタイム・ライブラリーでのみサポ

ートされています。*func* 引数は、<signal.h> インクルード・ファイルまたは関数アドレスに定義された、マクロ SIG_DFL または SIG_IGN のいずれかでなければなりません。

sig の値の意味は、以下のとおりです。

値　　意味

SIGABRT
異常終了

SIGALL
現在の処理の処置が SIG_DFL であるシグナルに対するキャッチ・オール。

SYSIFCOPT(*ASYNCSIGNAL) が指定された場合、SIGALL はキャッチ・オール・シグナルではありません。 SIGALL のシグナル・ハンドラーは、ユーザー発信の SIGALL シグナルの場合にのみ呼び出されます。

SIGILL
無効な関数イメージの検出

SIGFPE
オーバーフロー、ゼロによる割り算、および無効な演算などの、マスクされない算術例外

SIGINT
対話式アテンション

SIGIO レコード・ファイル入出力エラー

SIGOTHER
ILE C シグナル

SIGSEGV
無効なメモリーへのアクセス

SIGTERM
プログラムに送信された終了要求

SIGUSR1
ユーザー・アプリケーションによる使用目的 (ANSI への拡張)

SIGUSR2
ユーザー・アプリケーションによる使用目的 (ANSI への拡張)

割り込みシグナルが受信されたときに取られる処置は、*func* の値によって異なります。

値　　意味

SIG_DFL
シグナルのデフォルト処理が行われます。

SIG_IGN
シグナルが無視されます。

戻り値

戻り値 SIG_ERR は、`signal()` への呼び出しでエラーが発生したことを示します。正常に実行された場合、`signal()` への呼び出しは、*func* の最新の値を戻します。 errno の値は、EINVAL に設定されます (シグナルが無効です)。

`signal()` の使用例

この例では、シグナル・ハンドラーの設定方法を示します。

```
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#define  ONE_K  1024
#define OUT_OF_STORAGE       (SIGUSR1)
/* The SIGNAL macro does a signal() checking the return code */
#define SIGNAL(SIG, StrCln)       {                    ¥
  if (signal((SIG), (StrCln)) == SIG_ERR) {            ¥
    perror("Could not signal user signal");            ¥
    abort();                                           ¥
  }                                                    \
}

void StrCln(int);
void DoWork(char **, int);

int main(int argc, char *argv[]) {
  int size;
  char *buffer;
  SIGNAL(OUT_OF_STORAGE, StrCln);
  if (argc != 2) {
    printf("Syntax: %s size ¥n", argv[0]);
    return(-1);
  }
  size = atoi(argv[1]);
  DoWork(&buffer, size);
  return(0);
}

void StrCln(int SIG_TYPE) {
  printf("Failed trying to malloc storage¥n");
  SIGNAL(SIG_TYPE, SIG_DFL);
  exit(0);
}

void DoWork(char **buffer, int size) {
  int rc;
  *buffer = malloc(size*ONE_K);    /* get the size in number of K */
  if (*buffer == NULL) {
     if (raise(OUT_OF_STORAGE)) {
        perror("Could not raise user signal");
        abort();
     }
  }
  return;
}
/*  This is a program fragment and not a complete function example  */
```

関連情報

- 38 ページの『abort() ― プログラムの停止』
- 48 ページの『atexit() ― プログラム終了の記録関数』
- 92 ページの『exit() ― プログラムの終了』
- 267 ページの『raise() ― シグナルの送信』
- 15 ページの『<signal.h>』

- i5/OS Information Center のトピック『API』内の signal() API。

sin() ― 正弦の計算

フォーマット

```
#include <math.h>
double sin(double x);
```

言語レベル: ANSI

スレッド・セーフ: はい。

説明

`sin()` 関数は、ラジアンで表された *x* で、*x* の正弦を計算します。 *x* が大きすぎる場合、結果の有効数字が部分的に消失することがあります。

戻り値

`sin()` 関数は、*x* の正弦の値を戻します。 errno の値は、EDOM または ERANGE のいずれかに設定されます。

`sin()` の使用例

この例では、*y* を π/2 の正弦として計算します。

```
#include <math.h>
#include <stdio.h>

int main(void)
{
   double pi, x, y;

   pi = 3.1415926535;
   x = pi/2;
   y = sin(x);

   printf("sin( %lf ) = %lf¥n", x, y);
}
/*********************  Output should be similar to: *************

sin( 1.570796 ) = 1.000000
*/
```

関連情報

- 40 ページの『acos() ― 逆余弦の計算』
- 45 ページの『asin() ― 逆正弦の計算』
- 47 ページの『atan() - atan2() ― 逆正接の計算』
- 68 ページの『cos() ― 余弦の計算』
- 69 ページの『cosh() ― 双曲線余弦の計算』
- 366 ページの『sinh() ― 双曲線正弦の計算』
- 427 ページの『tan() ― 正接の計算』
- 428 ページの『tanh() ― 双曲線正接の計算』
- 9 ページの『<math.h>』

sinh() ― 双曲線正弦の計算

フォーマット

```
#include <math.h>
double sinh(double x);
```

言語レベル: ANSI

スレッド・セーフ: はい。

説明

sinh() 関数は、ラジアンで表された *x* で、*x* の双曲線正弦を計算します。

戻り値

sinh() 関数は、*x* の双曲線正弦の値を戻します。結果が大きすぎると、sinh() 関数は errno を ERANGE に設定して、値 HUGE_VAL (*x* の値によって正または負) を戻します。

sinh() の使用例

この例では、*y* を $\pi/2$ の双曲線正弦として計算します。

```
#include <math.h>
#include <stdio.h>

int main(void)
{
   double pi, x, y;

   pi = 3.1415926535;
   x = pi/2;
   y = sinh(x);

   printf("sinh( %lf ) = %lf¥n", x, y);
}

/*********************  Output should be similar to:  *************

sinh( 1.570796 ) = 2.301299
*/
```

関連情報

- 40 ページの『acos() ― 逆余弦の計算』
- 45 ページの『asin() ― 逆正弦の計算』
- 47 ページの『atan() - atan2() ― 逆正接の計算』
- 68 ページの『cos() ― 余弦の計算』
- 69 ページの『cosh() ― 双曲線余弦の計算』
- 365 ページの『sin() ― 正弦の計算』
- 427 ページの『tan() ― 正接の計算』
- 428 ページの『tanh() ― 双曲線正接の計算』
- 9 ページの『<math.h>』

snprintf() — フォーマット設定データのバッファーへの出力

フォーマット

```
#include <stdio.h>
int snprintf(char *buffer, size_t n, const char *format-string,
             argument-list);
```

言語レベル: ANSI

スレッド・セーフ: はい。

ロケール依存: この関数の振る舞いは、現行ロケールの LC_CTYPE カテゴリーおよび LC_NUMERIC カテゴリーの影響を受ける可能性があります。また、この振る舞いは、LOCALETYPE(*LOCALEUCS2) または LOCALETYPE(*LOCALEUTF) がコンパイル・コマンドに対して指定されている場合は、現行ロケールの LC_UNI_CTYPE カテゴリーの影響を受ける可能性もあります。詳細については、 549 ページの『CCSID およびロケールの理解』を参照してください。

説明

`snprintf()` 関数は、一連の文字と値をフォーマット設定し、配列 *buffer* に保管します。すべての *argument-list* は、*format-string* の対応する形式指定に従って変換され、出力されます。`snprintf()` 関数は、 `sprintf()` 関数に *n* 引数を加えたものと同じです。この引数は、*buffer* に書き込まれる最大文字数 (終了ヌル文字を含む) を示します。

format-string は、通常の文字で構成され、 `printf()` 関数の書式ストリングと同じ形式と機能を持っています。

戻り値

`snprintf()` 関数は、配列に書き込まれるバイト数 (終了ヌル文字はカウントされません) を戻します。

`snprintf()` の使用例

この例では、`snprintf()` を使用して、さまざまなデータをフォーマット設定し、出力します。

```
#include <stdio.h>

char buffer[200];
int i, j;
double fp;
char *s = "baltimore";
char c;

int main(void)
{
   c = 'l';
   i = 35;
   fp = 1.7320508;

   /* Format and print various data */
   j = snprintf(buffer, 6, "%s¥n", s);
   j += snprintf(buffer+j, 6, "%c¥n", c);
   j += snprintf(buffer+j, 6, "%d¥n", i);
   j += snprintf(buffer+j, 6, "%f¥n", fp);
   printf("string:¥n%s¥ncharacter count = %d¥n", buffer, j);
}

/********************  Output should be similar to:  *************
```

```
string:
baltil
35
1.732
character count = 15            */
```

関連情報

- 122 ページの『fprintf() ― フォーマット済みデータのストリームへの書き込み』
- 238 ページの『printf() ― 定様式の文字の出力』
- 『sprintf() ― フォーマット設定データのバッファーへの出力』
- 455 ページの『vsnprintf() ― 引数データのバッファーへの出力』
- 17 ページの『<stdio.h>』

sprintf() ― フォーマット設定データのバッファーへの出力

フォーマット

```
#include <stdio.h>
int sprintf(char *buffer, const char *format-string, argument-list);
```

言語レベル: ANSI

スレッド・セーフ: はい。

ロケール依存: この関数の振る舞いは、現行ロケールの LC_CTYPE カテゴリーおよび LC_NUMERIC カテゴリーの影響を受ける可能性があります。また、この振る舞いは、LOCALETYPE(*LOCALEUCS2) または LOCALETYPE(*LOCALEUTF) がコンパイル・コマンドに対して指定されている場合は、現行ロケールの LC_UNI_CTYPE カテゴリーの影響を受ける可能性もあります。詳細については、 549 ページの『CCSID およびロケールの理解』を参照してください。

説明

`sprintf()` 関数は、一連の文字と値をフォーマット設定し、配列 *buffer* に保管します。すべての *argument-list* は、*format-string* の対応する形式指定に従って変換され、出力されます。

format-string は、通常の文字で構成され、 `printf()` 関数の *format-string* 引数と同じ形式と機能を持っています。

戻り値

`sprintf()` 関数は、配列に書き込まれるバイト数 (終了ヌル文字はカウントされません) を戻します。

`sprintf()` の使用例

この例では、`sprintf()` を使用して、さまざまなデータをフォーマット設定し、出力します。

```
#include <stdio.h>

char buffer[200];
int i, j;
double fp;
char *s = "baltimore";
char c;

int main(void)
{
   c = 'l';
   i = 35;
   fp = 1.7320508;

   /* Format and print various data */
   j = sprintf(buffer, "%s¥n", s);
   j += sprintf(buffer+j, "%c¥n", c);
   j += sprintf(buffer+j, "%d¥n", i);
   j += sprintf(buffer+j, "%f¥n", fp);
   printf("string:¥n%s¥ncharacter count = %d¥n", buffer, j);
}

/*********************  Output should be similar to:  *************

string:
baltimore
l
35
1.732051

character count = 24           */
```

関連情報

- 122 ページの『fprintf() ― フォーマット済みデータのストリームへの書き込み』
- 238 ページの『printf() ― 定様式の文字の出力』
- 371 ページの『sscanf() ― データの読み取り』
- 423 ページの『swprintf() ― ワイド文字のフォーマット設定とバッファーへの書き込み』
- 444 ページの『vfprintf() ― ストリームへの引数データの出力』
- 452 ページの『vprintf() ― 引数データの出力』
- 456 ページの『vsprintf() ― 引数データのバッファーへの出力』
- 17 ページの『<stdio.h>』

---

sqrt() ― 平方根の計算

フォーマット

```
#include <math.h>
double sqrt(double x);
```

言語レベル: ANSI

スレッド・セーフ: はい。

説明

`sqrt()` 関数は、*x* の平方根の負以外の値を計算します。

戻り値

sqrt() 関数は、平方根の結果を戻します。 *x* が負の場合、関数は errno を EDOM に設定し、0 を戻します。

sqrt() の使用例

この例では、main の最初の引数として渡される数量の平方根を計算します。この関数は、ユーザーが負の値を渡すとエラー・メッセージを出力します。

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

int main(int argc, char ** argv)
{
  char * rest;
  double value;

  if ( argc != 2 )
    printf( "Usage: %s value¥n", argv[0] );
  else
  {
    value = strtod( argv[1], &rest);
    if ( value < 0.0 )
       printf( "sqrt of a negative number¥n" );
    else
       printf("sqrt( %lf ) = %lf¥n", value, sqrt( value ));
  }
}

/********************  If the input is 45,  *****************************
****************  then the output should be similar to:  **********

sqrt( 45.000000 ) = 6.708204
*/
```

関連情報

- 93 ページの『exp() ― 指数関数の計算』
- 175 ページの『hypot() ― 斜辺の計算』
- 198 ページの『log() ― 自然対数の計算』
- 199 ページの『log10() ― 基数 10 の対数の計算』
- 237 ページの『pow() ― 累乗の計算』
- 9 ページの『<math.h>』

srand() ― rand() 関数の seed の設定

フォーマット

```
#include <stdlib.h>
void srand(unsigned int seed);
```

言語レベル: ANSI

スレッド・セーフ: いいえ。

説明

`srand()` 関数は、一連の疑似ランダム整数を生成するための開始点を設定します。 `srand()` が呼び出されない場合は、srand(1) がプログラムの開始で呼び出されたときのように、 `rand()` seed が設定されます。それ以外の値が *seed* に設定された場合には、異なる開始点に生成プログラムが設定されます。

`rand()` 関数は、疑似乱数を生成します。

戻り値

戻り値はありません。

`srand()` の使用例

この例では、最初に値が 1 以外の `srand()` を呼び出し、ランダム値のシーケンスを開始します。次に、プログラムは、**ranvals** と呼ばれる整数配列の 5 つのランダム値を計算します。

```
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
   int i, ranvals[5];

   srand(17);
   for (i = 0; i < 5; i++)
   {
      ranvals[i] = rand();
      printf("Iteration %d ranvals [%d] = %d¥n", i+1, i, ranvals[i]);
   }
}

/******************  Output should be similar to:  ****************

Iteration 1 ranvals [0] = 24107
Iteration 2 ranvals [1] = 16552
Iteration 3 ranvals [2] = 12125
Iteration 4 ranvals [3] = 9427
Iteration 5 ranvals [4] = 13152
*/
```

関連情報

- 268 ページの『rand()、rand_r() ― 乱数の生成』
- 18 ページの『<stdlib.h>』

sscanf() ― データの読み取り

フォーマット

```
#include <stdio.h>
int sscanf(const char *buffer, const char *format, argument-list);
```

言語レベル: ANSI

スレッド・セーフ: はい。

ロケール依存: この関数の振る舞いは、現行ロケールの LC_CTYPE カテゴリーおよび LC_NUMERIC カテゴリーの影響を受ける可能性があります。また、この振る舞いは、LOCALETYPE(*LOCALEUCS2) または LOCALETYPE(*LOCALEUTF) がコンパイル・コマンドに対して指定されている場合は、現行ロケール

の LC_UNI_CTYPE カテゴリーの影響を受ける可能性もあります。詳細については、 549 ページの『CCSID およびロケールの理解』を参照してください。

説明

`sscanf()` 関数は、*buffer* から *argument-list* で指定された位置へ、データを読み取ります。各 *argument* は、 *format-string* 内の型指定子に対応する型の変数を指すポインターでなければなりません。

戻り値

`sscanf()` 関数は、正常に変換され、割り当てられたフィールドの数を戻します。戻り値には、読み取りは行われたが、割り当てられなかったフィールドは含まれません。

変換が行われる前に、ストリングの末尾が検出される場合、戻り値は EOF です。

`sscanf()` の使用例

この例では、`sscanf()` を使用して、*tokenstring* ストリングからさまざまなデータを読み取った後で、そのデータを表示します。

```
#include <stdio.h>
#include <stddef.h>

int main(void)
{
    char    *tokenstring = "15 12 14";
    char    *string = "ABC Z";
    wchar_t  ws[81];
    wchar_t  wc;
    int   i;
    float fp;
    char  s[81];
    char  c;

    /* Input various data                                      */
    /* In the first invocation of sscanf, the format string is */
    /* "%s %c%d%f". If there were no space between %s and %c,  */
    /* sscanf would read the first character following the     */
    /* string, which is a blank space.                         */

    sscanf(tokenstring, "%s %c%d%f", s, &c, &i, &fp);
    sscanf(string, "%ls %lc", ws,&wc);

    /* Display the data */
    printf("¥nstring = %s¥n",s);
    printf("character = %c¥n",c);
    printf("integer = %d¥n",i);
    printf("floating-point number = %f¥n",fp);
    printf("wide-character string = %S¥n",ws);
    printf("wide-character = %C¥n",wc);
}


/*****************  Output should be similar to:  *****************

string = 15
character = 1
integer = 2
floating-point number = 14.000000
wide-character string = ABC
wide-character = Z

*******************************************************************/
```

関連情報

- 138 ページの『fscanf() ― フォーマット済みデータの読み取り』
- 344 ページの『scanf() ― データの読み取り』
- 425 ページの『swscanf() ― ワイド文字データの読み取り』
- 153 ページの『fwscanf() ― ワイド文字を使用したストリームからのデータの読み取り』
- 528 ページの『wscanf() ― ワイド文字書式ストリングを使用したデータの読み取り』
- 368 ページの『sprintf() ― フォーマット設定データのバッファーへの出力』
- 17 ページの『<stdio.h>』

strcasecmp() ― 大/小文字を区別しないストリングの比較

フォーマット

```
#include <strings.h>
int srtcasecmp(const char *string1, const char *string2);
```

言語レベル: XPG4

スレッド・セーフ: はい。

ロケール依存: この関数の振る舞いは、現行ロケールの LC_CTYPE カテゴリーの影響を受ける可能性があります。この関数は、コンパイル・コマンドに対して LOCALETYPE(*CLD) が指定されている場合には使用できません。詳細については、 549 ページの『CCSID およびロケールの理解』を参照してください。

説明

strcasecmp() 関数は、大/小文字を区別せずに *string1* と *string2* を比較します。 *string1* および *string2* 内のすべての英字は、比較の前に小文字に変換されます。

strcasecmp() 関数は、ヌル終了ストリングを扱います。関数のストリング引数には、ストリングの終わりを示すマークであるヌル文字 ('¥0') が含まれると想定されます。

戻り値

strcasecmp() 関数は、2 つのストリング間の関係を示す次のような値を戻します。

表 6. strcasecmp() の戻り値

値	意味
0 より小さい値	*string1* は *string2* より小さい
0	*string1* は *string2* と等しい
0 より大きい値	*string1* は *string2* より大きい

strcasecmp() の使用例

この例では、strcasecmp() を使用して、2 つのストリングを比較します。

```
#include <stdio.h>
#include <strings.h>

int main(void)
{
  char_t *str1 = "STRING";
```

```
  char_t *str2 = "string";
  int result;

  result = strcasecmp(str1, str2);

  if (result == 0)
    printf("Strings compared equal.¥n");
  else if (result < 0)
    printf("¥"%s¥" is less than ¥"%s¥".¥n", str1, str2);
  else
    printf("¥"%s¥" is greater than ¥"%s¥".¥n", str1, str2);

  return 0;
}

/********  The output should be similar to: ***************

Strings compared equal.

***********************************/
```

関連情報

- 393 ページの『strncasecmp() ― 大/小文字を区別しないストリングの比較』
- 395 ページの『strncmp() ― ストリングの比較』
- 390 ページの『stricmp() - 大/小文字を区別しないストリングの比較』
- 474 ページの『wcscmp() ― ワイド文字ストリングの比較』
- 485 ページの『wcsncmp() ― ワイド文字ストリングの比較』
- 480 ページの『__wcsicmp() ― 大/小文字の区別をしないワイド文字ストリングの比較』
- 488 ページの『__wcsnicmp() ― 大/小文字の区別をしないワイド文字ストリングの比較』
- 19 ページの『<strings.h>』

strcat() ― ストリングの連結

フォーマット

```
#include <string.h>
char *strcat(char *string1, const char *string2);
```

言語レベル: ANSI

スレッド・セーフ: はい。

説明

`strcat()` 関数によって、*string2* と *string1* が連結され、結果のストリングがヌル文字で終了されます。

`strcat()` 関数は、ヌル終了ストリング上で作動します。関数のストリング引数には、ストリングの終わりを示すマークであるヌル文字 (¥0) が含まれていなければなりません。長さのチェックは行われません。*string2* がリテラル・ストリングの場合であっても、 *string1* 値にリテラル・ストリングを使用しないでください。

string1 のストレージが、*string2* のストレージとオーバーラップする場合、その振る舞いは予期できません。

戻り値

strcat() 関数は、連結されたストリング (*string1*) を指すポインターを戻します。

strcat() の使用例

この例では、strcat() を使用してストリング ″computer program″ を作成します。

```
#include <stdio.h>
#include <string.h>

#define SIZE 40


int main(void)
{
  char buffer1[SIZE] = "computer";
  char * ptr;

  ptr = strcat( buffer1, " program" );
  printf( "buffer1 = %s¥n", buffer1 );

}

/*****************  Output should be similar to:  *****************

buffer1 = computer program
*/
```

関連情報

- 『strchr() — 文字の検索』
- 376 ページの『strcmp() — ストリングの比較』
- 380 ページの『strcpy() — ストリングのコピー』
- 381 ページの『strcspn() — 最初に一致した文字のオフセットの検索』
- 394 ページの『strncat() — ストリングの連結』
- 471 ページの『wcscat() — ワイド文字ストリングの連結』
- 484 ページの『wcsncat() — ワイド文字ストリングの連結』
- 19 ページの『<string.h>』

strchr() — 文字の検索

フォーマット

```
#include <string.h>
char *strchr(const char *string, int c);
```

言語レベル: ANSI

スレッド・セーフ: はい。

ロケール依存: この関数の振る舞いは、現行ロケールの LC_CTYPE カテゴリーの影響を受ける可能性があります。詳細については、 549 ページの『CCSID およびロケールの理解』を参照してください。

説明

strchr() 関数は、ストリング内の文字の最初の出現を検出します。文字 *c* は、ヌル文字 (¥0) にすることができます。 *string* の終了ヌル文字も検索に含まれます。

strchr() 関数は、ヌル終了ストリング上で作動します。関数のストリング引数には、ストリングの終わりを示すマークであるヌル文字 (¥0) が含まれていなければなりません。

戻り値

strchr() 関数は、*string* 内の文字に変換された *c* の最初の出現を指すポインターを戻します。この関数は、指定された文字が見つからない場合、NULL を戻します。

strchr() の使用例

この例では、″computer program″ 内の文字 ″p″ の最初の出現を検出します。

```
#include <stdio.h>
#include <string.h>

#define SIZE 40

int main(void)
{
  char buffer1[SIZE] = "computer program";
  char * ptr;
  int    ch = 'p';

  ptr = strchr( buffer1, ch );
  printf( "The first occurrence of %c in '%s' is '%s'¥n",
            ch, buffer1, ptr );

}

/*****************  Output should be similar to:  *****************

The first occurrence of p in 'computer program' is 'puter program'
*/
```

関連情報

- 374 ページの『strcat() ― ストリングの連結』
- 『strcmp() ― ストリングの比較』
- 380 ページの『strcpy() ― ストリングのコピー』
- 381 ページの『strcspn() ― 最初に一致した文字のオフセットの検索』
- 395 ページの『strncmp() ― ストリングの比較』
- 401 ページの『strpbrk() ― ストリング内の文字の検索』
- 406 ページの『strrchr() ― ストリング内で文字が最後に現れる位置の検出』
- 407 ページの『strspn() ―最初の不一致文字のオフセットの検索』
- 472 ページの『wcschr() ― ワイド文字の検索』
- 496 ページの『wcsspn() ― 最初の不一致ワイド文字のオフセットの検索』
- 19 ページの『<string.h>』

strcmp() ― ストリングの比較

フォーマット

```
#include <string.h>
int strcmp(const char *string1, const char *string2);
```

言語レベル: ANSI

スレッド・セーフ: はい。

説明

strcmp() 関数は、*string1* と *string2* を比較します。この関数は、ヌル終了ストリング上で作動します。関数のストリング引数には、ストリングの終わりを示すマークであるヌル文字 (¥0) が含まれていなければなりません。

戻り値

strcmp() 関数は、2 つのストリング間の関係を示す次のような値を戻します。

値	意味
0 より小さい値	*string1* は *string2* より小さい
0	*string1* は *string2* と等しい
0 より大きい値	*string1* は *string2* より大きい

strcmp() の使用例

この例では、strcmp() を使用して main() に渡される 2 つのストリングを比較します。

```
#include <stdio.h>
#include <string.h>

int main(int argc, char ** argv)
{
  int  result;

  if ( argc != 3 )
  {
    printf( "Usage: %s string1 string2¥n", argv[0] );
  }
  else
  {

    result = strcmp( argv[1], argv[2] );

    if ( result == 0 )
      printf( "¥"%s¥" is identical to ¥"%s¥"¥n", argv[1], argv[2] );
    else if ( result < 0 )
      printf( "¥"%s¥" is less than ¥"%s¥"¥n", argv[1], argv[2] );
    else
      printf( "¥"%s¥" is greater than ¥"%s¥"¥n", argv[1], argv[2] );
  }
}

/******************  If the input is the strings  ***********************
**********  "is this first?" and "is this before that one?",  ***********
******************  then the expected output is:  *********************

"is this first?" is greater than "is this before that one?"
**********************************************************************/
```

関連情報

- 374 ページの『strcat() ― ストリングの連結』
- 375 ページの『strchr() ― 文字の検索』
- 380 ページの『strcpy() ― ストリングのコピー』

- 381 ページの『strcspn() ― 最初に一致した文字のオフセットの検索』
- 395 ページの『strncmp() ― ストリングの比較』
- 401 ページの『strpbrk() ― ストリング内の文字の検索』
- 406 ページの『strrchr() ― ストリング内で文字が最後に現れる位置の検出』
- 407 ページの『strspn() ―最初の不一致文字のオフセットの検索』
- 472 ページの『wcschr() ― ワイド文字の検索』
- 496 ページの『wcsspn() ― 最初の不一致ワイド文字のオフセットの検索』
- 19 ページの『<string.h>』

strcmpi() - 大/小文字を区別しないストリングの比較

フォーマット

```
#include <string.h>
int strcmpi(const char *string1, const char *string2);
```

注: strcmpi 関数は、C++ プログラムで使用可能です。 **__cplusplus__strings__** マクロがプログラムで定義されている場合にのみ、C でも使用できます。

言語レベル: Extension

スレッド・セーフ: はい。

ロケール依存: この関数の振る舞いは、現行ロケールの LC_CTYPE カテゴリーの影響を受ける可能性があります。詳細については、 549 ページの『CCSID およびロケールの理解』を参照してください。

説明

strcmpi は、大/小文字を区別せずに *string1* と *string2* を比較します。 2 つの引数 *string1* と *string2* 内のすべての英字は、比較の前に小文字に変換されます。

この関数は、ヌル終了ストリング上で作動します。関数のストリング引数には、ストリングの終わりを示すマークであるヌル文字 (¥0) が含まれると想定されます。

戻り値

strcmpi は、2 つのストリング間の関係を示す次のような値を戻します。

値	意味
0 より小さい値	*string1* は *string2* より小さい
0	*string1* は *string2* と等しい
0 より大きい値	*string1* は *string2* より大きい

strcmpi() の使用例

この例では、strcmpi を使用して 2 つのストリングを比較します。

```
#include <stdio.h>
#include <string.h>
int main(void)
{
   /* Compare two strings without regard to case                          */
```

```
   if (0 == strcmpi("hello", "HELLO"))
      printf("The strings are equivalent.\n");
   else
      printf("The strings are not equivalent.\n");
   return 0;
}
```

The output should be:

```
      The strings are equivalent.
```

関連情報

- 『strcoll() — ストリングの比較』
- 381 ページの『strcspn() — 最初に一致した文字のオフセットの検索』
- 383 ページの『strdup - ストリングの複製』
- 390 ページの『stricmp() - 大/小文字を区別しないストリングの比較』
- 395 ページの『strncmp() — ストリングの比較』
- 398 ページの『strnicmp - 大/小文字の区別をしないサブストリングの比較』
- 474 ページの『wcscmp() — ワイド文字ストリングの比較』
- 485 ページの『wcsncmp() — ワイド文字ストリングの比較』
- 373 ページの『strcasecmp() — 大/小文字を区別しないストリングの比較』
- 393 ページの『strncasecmp() — 大/小文字を区別しないストリングの比較』
- 19 ページの『<string.h>』

---

strcoll() — ストリングの比較

フォーマット

```
#include <string.h>
int strcoll(const char *string1, const char *string2);
```

言語レベル: ANSI

スレッド・セーフ: はい。

ロケール依存: この関数の振る舞いは、現行ロケールの LC_COLLATE カテゴリーの影響を受ける可能性があります。詳細については、 549 ページの『CCSID およびロケールの理解』を参照してください。

説明

`strcoll()` 関数は、プログラムのロケールで指定された照合シーケンスを使用して、 2 つのストリングを比較します。

戻り値

`strcoll()` 関数は、2 つのストリング間の関係を示す次のような値を戻します。

値	意味
0 より小さい値	*string1* は *string2* より小さい
0	*string1* は *string2* と等しい
0 より大きい値	*string1* は *string2* より大きい

strcoll() が正常に実行されなかった場合は、errno が変更されます。 errno の値は EINVAL に設定されます (*string1* または *string2* 引数に、現行ロケールで使用できない文字が含まれています)。

strcoll() の使用例

この例では、strcoll() を使用して main() に渡される 2 つのストリングを比較します。

```
#include <stdio.h>
#include <string.h>

int main(int argc, char ** argv)
{
  int  result;

  if ( argc != 3 )
  {
    printf( "Usage: %s string1 string2¥n", argv[0] );
  }
  else
  {

    result = strcoll( argv[1], argv[2] );

    if ( result == 0 )
      printf( "¥"%s¥" is identical to ¥"%s¥"¥n", argv[1], argv[2] );
    else if ( result < 0 )
      printf( "¥"%s¥" is less than ¥"%s¥"¥n", argv[1], argv[2] );
    else
      printf( "¥"%s¥" is greater than ¥"%s¥"¥n", argv[1], argv[2] );
  }
}

/******************  If the input is the strings  ***********************
****************  "firststring" and "secondstring",  ********************
******************  then the expected output is:  *****************

"firststring" is less than "secondstring"
*/
```

関連情報

- 354 ページの『setlocale() ― ロケールの設定』
- 376 ページの『strcmp() ― ストリングの比較』
- 395 ページの『strncmp() ― ストリングの比較』
- 475 ページの『wcscoll() ―言語照合ストリングの比較』
- 19 ページの『<string.h>』

strcpy() ― ストリングのコピー

フォーマット

```
#include <string.h>
char *strcpy(char *string1, const char *string2);
```

言語レベル: ANSI

スレッド・セーフ: はい。

説明

strcpy() 関数は、終了ヌル文字を含め、*string2* を *string1* で指定された位置にコピーします。

strcpy() 関数は、ヌル終了ストリング上で作動します。関数のストリング引数には、ストリングの終わりを示すマークであるヌル文字 (¥0) が含まれていなければなりません。長さのチェックは行われません。*string2* がリテラル・ストリングの場合であっても、 *string1* 値にリテラル・ストリングを使用しないでください。

戻り値

strcpy() 関数は、コピーされたストリング (*string1*) を指すポインターを戻します。

strcpy() の使用例

この例では、source の内容を destination にコピーします。

```
#include <stdio.h>
#include <string.h>

#define SIZE 40

int main(void)
{
  char source[SIZE] = "This is the source string";
  char destination[SIZE] = "And this is the destination string";
  char * return_string;

  printf( "destination is originally = ¥"%s¥"¥n", destination );
  return_string = strcpy( destination, source );
  printf( "After strcpy, destination becomes ¥"%s¥"¥n", destination );
}

/*****************  Output should be similar to:  *****************

destination is originally = "And this is the destination string"
After strcpy, destination becomes "This is the source string"
*/
```

関連情報

- 374 ページの『strcat() ― ストリングの連結』
- 375 ページの『strchr() ― 文字の検索』
- 376 ページの『strcmp() ― ストリングの比較』
- 『strcspn() ― 最初に一致した文字のオフセットの検索』
- 397 ページの『strncpy() ― ストリングのコピー』
- 401 ページの『strpbrk() ― ストリング内の文字の検索』
- 406 ページの『strrchr() ― ストリング内で文字が最後に現れる位置の検出』
- 407 ページの『strspn() ―最初の不一致文字のオフセットの検索』
- 476 ページの『wcscpy() ― ワイド文字ストリングのコピー』
- 487 ページの『wcsncpy() ― ワイド文字ストリングのコピー』
- 19 ページの『<string.h>』

strcspn() ― 最初に一致した文字のオフセットの検索

フォーマット

```
#include <string.h>
size_t strcspn(const char *string1, const char *string2);
```

言語レベル: ANSI

スレッド・セーフ: はい。

ロケール依存: この関数の振る舞いは、現行ロケールの LC_CTYPE カテゴリーの影響を受ける可能性があります。詳細については、 549 ページの『CCSID およびロケールの理解』を参照してください。

説明

strcspn() 関数は、*string2* で指定された一連の文字に属する *string1* 内の文字の最初の出現を検出します。ヌル文字は、検索対象になりません。

strcspn() 関数は、ヌル終了ストリング上で作動します。関数のストリング引数には、ストリングの終わりを示すマークであるヌル文字 (¥0) が含まれていなければなりません。

戻り値

strcspn() 関数は、検出された最初の文字のインデックスを戻します。この値は、すべて *string2* 内にはない文字で構成される、*string1* の最初のサブストリングの長さと同じです。

strcspn() の使用例

この例では、strcspn() を使用して、 *string* 内の文字 ″a″、″x″、″l″、または ″e″ のうちいずれかの最初の出現を検出します。

```
#include <stdio.h>
#include <string.h>

#define SIZE 40

int main(void)
{
  char string[SIZE] = "This is the source string";
  char * substring = "axle";

  printf( "The first %i characters in the string ¥"%s¥" "
          "are not in the string ¥"%s¥" ¥n",
            strcspn(string, substring), string, substring);

}

/**********  Output should be similar to:  **************

The first 10 characters in the string "This is the source string"
are not in the string "axle"
*/
```

関連情報

- 374 ページの『strcat() ― ストリングの連結』
- 375 ページの『strchr() ― 文字の検索』
- 376 ページの『strcmp() ― ストリングの比較』
- 380 ページの『strcpy() ― ストリングのコピー』
- 395 ページの『strncmp() ― ストリングの比較』

- 401 ページの『strpbrk() ― ストリング内の文字の検索』
- 406 ページの『strrchr() ― ストリング内で文字が最後に現れる位置の検出』
- 407 ページの『strspn() ―最初の不一致文字のオフセットの検索』
- 19 ページの『<string.h>』

strdup - ストリングの複製

フォーマット

```
#include <string.h>
char *strdup(const char *string);
```

注: strdup 関数は、C++ プログラムで使用可能です。 __cplusplus__strings__ マクロがプログラムで定義されている場合にのみ、C でも使用できます。

言語レベル: XPG4、Extension

スレッド・セーフ: はい。

説明

strdup は、malloc を呼び出して、*string* のコピー用の予約ストレージ・スペースを予約します。この関数のストリング引数には、ストリングの終わりを示すマークであるヌル文字 (¥0) が含まれると想定されます。 strdup を呼び出して予約したストレージは、忘れずに解放してください。

戻り値

strdup は、コピーされたストリングを含むストレージ・スペースを指すポインターを戻します。ストレージ strdup を予約できない場合は、NULL を戻します。

strdup の使用例

この例では、strdup を使用して、ストリングを複製し、コピーを出力します。

```
#include <stdio.h>
#include <string.h>
int main(void)
{
   char *string = "this is a copy";
   char *newstr;
   /* Make newstr point to a duplicate of string                              */
   if ((newstr = strdup(string)) != NULL)
      printf("The new string is: %s\n", newstr);
   return 0;
}
```

The output should be:

```
   The new string is: this is a copy
```

関連情報

- 380 ページの『strcpy() ― ストリングのコピー』
- 397 ページの『strncpy() ― ストリングのコピー』
- 476 ページの『wcscpy() ― ワイド文字ストリングのコピー』
- 487 ページの『wcsncpy() ― ワイド文字ストリングのコピー』

- 477 ページの『wcscspn() ― 最初に一致したワイド文字のオフセットの検索』
- 19 ページの『<string.h>』

strerror() ― ランタイム・エラー・メッセージを指すポインターの設定

フォーマット

```
#include <string.h>
char *strerror(int errnum);
```

言語レベル: ANSI

スレッド・セーフ: はい。

説明

strerror() 関数は、*errnum* のエラー番号を、エラー・メッセージ・ストリングにマップします。

戻り値

strerror() 関数は、ストリングを指すポインターを戻します。 NULL 値は戻しません。 errno の値は **ECONVERT** (変換エラー) に設定される可能性があります。

strerror() の使用例

この例では、ファイルをオープンし、エラーが発生した場合にランタイム・エラー・メッセージを戻します。

```
#include <stdlib.h>
#include <string.h>
#include <errno.h>

int main(void)
{
   FILE *stream;

   if ((stream = fopen("mylib/myfile", "r")) == NULL)
      printf(" %s ¥n", strerror(errno));

}

/*  This is a program fragment and not a complete function example  */
```

関連情報

- 65 ページの『clearerr() ― エラー標識のリセット』
- 100 ページの『ferror() ― 読み取り/書き込みエラーのテスト』
- 236 ページの『perror() ― エラー・メッセージの出力』
- 19 ページの『<string.h>』

strfmon() ― 通貨値からストリングへの変換

フォーマット

```
#include <monetary.h>
int strfmon(char *s, size_t maxsize, const char *format, argument_list);
```

言語レベル: XPG4

スレッド・セーフ: はい。

ロケール依存: この関数の振る舞いは、現行ロケールの LC_CTYPE カテゴリーおよび LC_MONETARY カテゴリーの影響を受ける可能性があります。この関数は、コンパイル・コマンドに対して LOCALETYPE(*CLD) が指定されている場合には使用できません。詳細については、549 ページの『CCSID およびロケールの理解』を参照してください。

説明

`strfmon()` 関数は、*format* が指す制御ストリングで指定された形式で、 *s* が指す配列に文字を入れます。*maxsize* までの文字が配列に入ります。

文字ストリング *format* には、2 つのタイプのオブジェクトが含まれています。1 つは出力ストリームにコピーされるプレーン文字で、もう 1 つはディレクティブです。それぞれの結果として、変換とフォーマット設定が行われるゼロ個以上の引数が取り出されます。 format の引数が不十分な場合は、結果は予期できません。引数が残っているのに、format が使い尽くされている場合、余分な引数は無視されます。double 値に関する変換で保証される有効数字数は 15 のみです。

ディレクティブは % 文字、オプションの変換指定、およびディレクティブの振る舞いを決定する終了文字で構成されています。

ディレクティブは、次のシーケンスで構成されています。
- % 文字。
- オプションのフラグ。
- オプションのフィールド幅。
- オプションの左方精度。
- オプションの右方精度。
- 実行する変換のタイプを示す必須の変換文字。

表 7. フラグ

フラグ	意味
=f	**=** の後に、数字充てん文字として使用される単一文字 **f** を指定します。 デフォルトでは、数字充てん文字はスペース文字です。このフラグは、常にスペース文字を使用するフィールド幅の充てんには影響を及ぼしません。このフラグは、左方精度が指定されない限り、無視されます。
^	通貨値のフォーマット設定を行うときに、グループ化文字を使用しません。デフォルトでは、現行ロケールで定義されている、グループ化文字が挿入されることになります。
+ または (	正および負の通貨額を表示するスタイルを指定します。**+** が指定されると、通貨数量の + および - のロケールの同値が使用されます。**(** が指定されると、負の額が括弧で囲まれます。デフォルトは **+** です。
!	通貨記号を出力しません。デフォルトでは通貨記号が出力されます。

表 7. フラグ (続き)

-	二重引数に左方位置調整を使用します。デフォルトは右方位置調整です。

フィールド幅

w 10 進数ストリング *w* により、変換の結果が右寄せ (またはフラグ **-** が指定されている場合には左寄せ) の最小フィールド幅がバイトで指定されます。デフォルトは 0 です。

左方精度

#n # の後に 10 進数ストリング *n* を続けることにより、基数文字の左方にフォーマット設定される最大桁数を指定します。このオプションを使用すると、strfmon() への複数回の呼び出しによる出力 (フォーマット設定済み) の位置合わせを、同じ列のままにしておくことができます。またこれを使用して、$***123.45 などの特殊文字で未使用の位置を充てんすることもできます。このオプションによって、 *n* で指定された桁数があるかのように、金額をフォーマット設定できます。 *n* より多い桁位置が必要な場合、この変換指定は無視されます。実際に必要な数より多い桁位置は、数字充てん文字で充てんされます (上記の **=f** フラグを参照してください)。

グループ化が ^ フラグで抑制されておらず、現行ロケールで定義される場合、充てん文字が (ある場合に) 追加される前に、グループ化区切り文字が挿入されます。グループ化区切り文字は、充てん文字が数字である場合でも、充てん文字に適用されることはありません。位置合わせを保証するために、フォーマット設定された出力において、数の前後に表示されるどのような文字も (通貨または符号シンボルなど)、その正または負の形式を等しい長さにするために、必要に応じて、空白文字が埋め込まれます。

右方精度

.p 10 進数ストリング *p* が後に続くピリオドにより、基数文字の後の桁数が指定されます。右方精度 *p* の値が 0 の場合は、基数文字は表示されません。右方精度を指定しない場合は、現行ロケールで指定されたデフォルトが使用されます。フォーマット設定されている金額は、フォーマット設定前の指定桁数に丸められます。

表 8. 変換文字

指定子	意味
%i	二重引数は、ロケールの国際通貨形式に応じてフォーマット設定されます。
%n	二重引数は、ロケールの国の通貨形式に応じてフォーマット設定されます。
%%	% で置き換えられます。引数は変換されません。

戻り値

終了ヌル文字を含む結果バイトの合計数が *maxsize* より大きくない場合、strfmon() 関数は *s* が指す配列に入れられたバイト数を戻しますが、この時は終了ヌル文字は除外します。それ以外の場合は、ゼロが戻され、配列の内容は未定義となります。

errno の値は、次のいずれかに設定されます。

E2BIG バッファーのスペース不足のために停止された変換。

strfmon() の使用例

```
#include <stdio.h>
 #include <monetary.h>
 #include <locale.h>

int main(void)
{
    char string[100];
    double money = 1234.56;
    if (setlocale(LC_ALL, "/qsys.lib/en_us.locale") == NULL) {
        printf("Unable to setlocale().¥n");
        exit(1);
    }

    strfmon(string, 100, "%i", money); /* USD 1,234.56 */
    printf("%s¥n", string);
    strfmon(string, 100, "%n", money); /* $1,234.56 */
    printf("%s¥n", string);
}
/*************************************************************
         The output should be similar to:
         USD 1,234.56
         $1,234.56
*************************************************************/
```

関連情報

- 187 ページの『localeconv() ― 環境からの情報の取得』
- 9 ページの『<monetary.h>』

---

strftime() ― 日付/時刻からストリングへの変換

フォーマット

```
#include <time.h>
size_t strftime(char *s, size_t maxsize, const char *format,
                   const struct tm *timeptr);
```

言語レベル: ANSI

スレッド・セーフ: はい。

ロケール依存: この関数の振る舞いは、現行ロケールの LC_CTYPE、LC_TIME、および LC_TOD カテゴリーの影響を受ける可能性があります。詳細については、 549 ページの『CCSID およびロケールの理解』を参照してください。

説明

`strftime()` 関数は、*format* が指す制御ストリングで指定された形式で、 *s* が指す配列にバイトを入れます。書式ストリングは、ゼロ個以上の変換指定と普通文字で構成されています。変換指定は、% 文字と変換の振る舞いを決定する終了変換文字で構成されています。すべての普通文字 (終了ヌル・バイトおよびマルチバイト文字を含む) は、変更されないまま配列にコピーされます。オーバーラップしたオブジェクト間でコピーが行われる場合には、振る舞いは予期できません。 *maxsize* までのバイトが配列に入ります。該当する文字は、*timeptr* が指す構造体に含まれる値と現行ロケールに保管された値によって決定されます。

標準変換指定は、それぞれ該当する文字で置き換えられます。次の表を参照してください。

指定子	意味
%a	曜日の省略名。

指定子	意味
%A	曜日のフルネーム。
%b	月の省略名。
%B	月のフルネーム。
%c	ロケールの形式の日付/時刻。
%C	世紀数 [00-99] (100 で除算し、整数に切り捨てた年)。
%d	日 [01-31]。
%D	日付形式で、%m/%d/%y と同じ。
%e	1 桁の場合は前にスペースが入ることを除き、 %d と同じ [1-31]。
%g	ISO 日付の 2 桁の年の部分 [00,99]。
%G	ISO 日付の 4 桁の年の部分。負になる場合があります。
%h	%b と同じ。
%H	24 時間形式の時間 [00-23]。
%I	12 時間形式の時間 [01-12]。
%j	ユリウス日付 [001-366]。
%m	月 [01-12]。
%M	分 [00-59]。
%n	改行文字
%p	AM または PM ストリング。
%r	ロケールの AM/PM 形式の時刻。ロケールの時刻形式で使用できない場合、POSIX 時刻 AM/PM 形式である %I:%M:%S %p をデフォルトとします。
%R	秒を含まない 24 時間の時刻形式で、 %H:%M と同じ。
%S	秒 [00-61]。秒の範囲は、うるう秒および二重うるう秒を考慮に入れています。
%t	タブ文字。
%T	秒を含む 24 時間の時刻形式で、 %H:%M:%S と同じ。
%u	曜日 [1,7]。月曜が 1 で、日曜が 7。
%U	1 年の週数 [00-53]。日曜が週の初日。
%V	1 年の ISO 週数 [01-53]。月曜が週の初日。1 月 1 日の週に新年の 4 日以上が含まれる場合、この週は週 1 と考慮されます。そうでない場合、この週は前年の最終週となり、次の週が新年の週 1 となります。
%w	曜日 [0,6] で、日曜が 0。
%W	1 年の週数 [00-53]。月曜が週の初日。
%x	ロケールの形式の日付。
%X	ロケールの形式の時刻。
%y	2 桁の年 [00,99]。
%Y	4 桁の年。負になる場合があります。
%z	UTC オフセット。出力は、+HHMM または -HHMM の形式のストリングで、+ は GMT の東、- は GMT の西、HH は GMT からの時間数、MM は GMT からの分数をそれぞれ表します。
%Z	時間帯名。
%%	% 文字。

変更された変換指定子

いくつかの変換指定子は、E または O 修飾子文字により変更可能です。この変更を行うことで、通常使用されている未変更変換指定子の変わりに、代替の形式や指定方法が使用されるべきであることを示すことができます。変換された変換指定子が現行ロケールで使用不可になっているフィールドを使用する場合、振る舞いは未変更変換指定が使用されたかのようになります。例えば、*era* ストリングが空ストリング "" (ストリングは使用不可であることを意味します) である場合、%EY が %Y のように振る舞います。

指定子	意味
%Ec	現在の時代の日付/時刻。
%EC	年代名。
%Ex	現在の時代の日付。
%EX	現在の時代の時刻。
%Ey	年代の年。これは基本の年からのオフセットです。
%EY	現在の時代の年。
%Od	代替数字を使用した日。
%Oe	%Od と同じ。
%OH	代替数字を使用した 24 時間形式の時間。
%OI	代替数字を使用した 12 時間形式の時間。
%Om	代替数字を使用した月。
%OM	代替数字を使用した分。
%OS	代替数字を使用した秒。
%Ou	代替数字を使用した曜日。月曜が 1 で、日曜が 7。
%OU	代替数字を使用した 1 年の週数。日曜が週の初日。
%OV	代替数字を使用した 1 年の ISO 週数。 ISO 週数の説明については、%V を参照してください。
%Ow	代替数字を使用した曜日。日曜が 0。
%OW	代替数字を使用した 1 年の週数。月曜が週の初日。
%Oy	代替数字を使用した 2 桁の年。
%OZ	時間帯名が現行ロケールに存在する場合、これは %Z と同じです。存在しない場合、現行ジョブの省略された時間帯名が戻されます。

注: %C、%D、%e、%h、%n、%r、%R、%t、%T、%u、%V、および変更された変換指定子は、コンパイル・コマンドで LOCALETYPE(*CLD) が指定されている場合は使用できません。

戻り値

終了ヌル・バイトを含む結果バイトの合計数が maxsize を超えない場合、`strftime()` は、終了ヌル・バイトを含まない、 *s* が指す配列に配置されたバイト数を戻します。それ以外の場合は、0 が戻され、配列の内容は不確定となります。

変換エラーが発生した場合、`errno` は **ECONVERT** に設定される可能性があります。

`strftime()` の使用例

```
#include <stdio.h>
#include <time.h>

int main(void)
{
     char s[100];
     int rc;
     time_t temp;
     struct tm *timeptr;

     temp = time(NULL);
     timeptr = localtime(&temp);

     rc = strftime(s,sizeof(s),"Today is %A, %b %d.¥nTime:  %r", timeptr);
     printf("%d characters written.¥n%s¥n",rc,s);

     return 0;
}

/**************************************************
     The output should be similar to:
     46 characters written
     Today is Wednesday, Oct 24.
     Time:  01:01:15 PM
************************************************************/
```

関連情報

- 41 ページの『asctime() ― 時間から文字ストリングへの変換』
- 43 ページの『asctime_r() ― 時間から文字ストリングへの変換 (再始動可能)』
- 75 ページの『ctime() ― 時間から文字ストリングへの変換』
- 77 ページの『ctime64() ― 時間から文字ストリングへの変換』
- 80 ページの『ctime64_r() ― 時間から文字ストリングへの変換 (再始動可能)』
- 78 ページの『ctime_r() ― 時間から文字ストリングへの変換 (再始動可能)』
- 167 ページの『gmtime() ― 時間の変換』
- 169 ページの『gmtime64() ― 時間の変換』
- 173 ページの『gmtime64_r() ― 時間の変換 (再始動可能)』
- 171 ページの『gmtime_r() ― 時間の変換 (再始動可能)』
- 192 ページの『localtime() ― 時間の変換』
- 194 ページの『localtime64() ― 時間の変換』
- 197 ページの『localtime64_r() ― 時間の変換 (再始動可能)』
- 195 ページの『localtime_r() ― 時間の変換 (再始動可能)』
- 354 ページの『setlocale() ― ロケールの設定』
- 402 ページの『strptime()― ストリングから日付/時刻への変換』
- 429 ページの『time() ― 現在時刻の判別』
- 430 ページの『time64() ― 現在時刻の判別』
- 19 ページの『<time.h>』

stricmp() - 大/小文字を区別しないストリングの比較

フォーマット

```
#include <string.h>
int stricmp(const char *string1, const char *string2);
```

注: stricmp 関数は、 C++ プログラムで使用可能です。__cplusplus__strings__ マクロがプログラムで定義されている場合にのみ、C でも使用できます。

言語レベル: Extension

スレッド・セーフ: はい。

ロケール依存: この関数の振る舞いは、現行ロケールの LC_CTYPE カテゴリーの影響を受ける可能性があります。詳細については、 549 ページの『CCSID およびロケールの理解』を参照してください。

説明

stricmp は、*string1* と *string2* を大/小文字の区別なしで比較します。2 つの引数 *string1* と *string2* 内のすべての英字は、比較の前に小文字に変換されます。

この関数は、ヌル終了ストリング上で作動します。関数のストリング引数には、ストリングの終わりを示すマークであるヌル文字 (¥0) が含まれると想定されます。

戻り値

stricmp は、2 つのストリング間の関係を示す次のような値を戻します。

値	意味
0 より小さい値	*string1* は *string2* より小さい
0	*string1* は *string2* と等しい
0 より大きい値	*string1* は *string2* より大きい

stricmp() の使用例

この例では stricmp を使用して、2 つのストリングを比較します。

```
#include <stdio.h>
#include <string.h>
int main(void)
{
   /* Compare two strings as lowercase                          */
   if (0 == stricmp("hello", "HELLO"))
      printf("The strings are equivalent.\n");
   else
      printf("The strings are not equivalent.\n");
   return 0;
}
```

The output should be:

```
The strings are equivalent.
```

関連情報

- 378 ページの『strcmpi() - 大/小文字を区別しないストリングの比較』
- 379 ページの『strcoll() — ストリングの比較』
- 381 ページの『strcspn() — 最初に一致した文字のオフセットの検索』
- 383 ページの『strdup - ストリングの複製』

- 395 ページの『strncmp() ― ストリングの比較』
- 373 ページの『strcasecmp() ― 大/小文字を区別しないストリングの比較』
- 393 ページの『strncasecmp() ― 大/小文字を区別しないストリングの比較』
- 398 ページの『strnicmp - 大/小文字の区別をしないサブストリングの比較』
- 474 ページの『wcscmp() ― ワイド文字ストリングの比較』
- 485 ページの『wcsncmp() ― ワイド文字ストリングの比較』
- 19 ページの『<string.h>』

strlen() ― ストリング長の判別

フォーマット

```
#include <string.h>
size_t strlen(const char *string);
```

言語レベル: ANSI

スレッド・セーフ: はい。

説明

`strlen()` 関数は、終了ヌル文字以外の *string* の長さを決定します。

戻り値

`strlen()` 関数は *string* の長さを戻します。

`strlen()` の使用例

この例では、`main()` に受け渡されるストリングの長さを判別します。

```
#include <stdio.h>
#include <string.h>

int main(int argc, char ** argv)
{

  if ( argc != 2 )
    printf( "Usage: %s string¥n", argv[0] );
  else
    printf( "Input string has a length of %i¥n", strlen( argv[1] ));
}
/******************  If the input is the string  ***********************
*****************"How long is this string?", *****************
******************  then the expected output is:  ****************

Input string has a length of 24
*/
```

関連情報

- 205 ページの『mblen() ― マルチバイト文字の長さの計算』
- 394 ページの『strncat() ― ストリングの連結』
- 395 ページの『strncmp() ― ストリングの比較』
- 397 ページの『strncpy() ― ストリングのコピー』

- 482 ページの『wcslen() ― ワイド文字ストリング長の計算』
- 19 ページの『<string.h>』

strncasecmp() ― 大/小文字を区別しないストリングの比較

フォーマット

```
#include <strings.h>
int strncasecmp(const char *string1, const char *string2, size_t count);
```

言語レベル: XPG4

スレッド・セーフ: はい。

ロケール依存: この関数の振る舞いは、現行ロケールの LC_CTYPE カテゴリーの影響を受ける可能性があります。この関数は、コンパイル・コマンドに対して LOCALETYPE(*CLD) が指定されている場合には使用できません。詳細については、 549 ページの『CCSID およびロケールの理解』を参照してください。

説明

`strncasecmp()` 関数は、*string1* と *string2* の最大 *count* 文字を大/小文字の区別なしで比較します。*string1* および *string2* 内のすべての英字は、比較の前に小文字に変換されます。

`strncasecmp()` 関数は、ヌル終了ストリング上で作動します。関数のストリング引数には、ストリングの終わりを示すマークであるヌル文字 (`'¥0'`) が含まれると想定されます。

戻り値

`strncasecmp()` 関数は、2 つのストリング間の関係を示す次のような値を戻します。

表 9. strncasecmp() の戻り値

値	意味
0 より小さい値	*string1* は *string2* より小さい
0	*string1* は *string2* と等しい
0 より大きい値	*string1* は *string2* より大きい

`strncasecmp` の使用例

この例では `strncasecmp()` を使用して、2 つのストリングを比較します。

```
#include <stdio.h>
#include <strings.h>

int main(void)
{
  char_t *str1 = "STRING ONE";
  char_t *str2 = "string TWO";
  int result;

  result = strncasecmp(str1, str2, 6);

  if (result == 0)
    printf("Strings compared equal.¥n");
  else if (result < 0)
    printf("¥"%s¥" is less than ¥"%s¥".¥n", str1, str2);
  else
```

```
    printf("¥"%s¥" is greater than ¥"%s¥".¥n", str1, str2);

  return 0;
}

/********  The output should be similar to: ***************

Strings compared equal.

**********************************/
```

関連情報

- 373 ページの『strcasecmp() ― 大/小文字を区別しないストリングの比較』
- 395 ページの『strncmp() ― ストリングの比較』
- 390 ページの『stricmp() - 大/小文字を区別しないストリングの比較』
- 474 ページの『wcscmp() ― ワイド文字ストリングの比較』
- 485 ページの『wcsncmp() ― ワイド文字ストリングの比較』
- 480 ページの『__wcsicmp() ― 大/小文字の区別をしないワイド文字ストリングの比較』
- 488 ページの『__wcsnicmp() ― 大/小文字の区別をしないワイド文字ストリングの比較』
- 19 ページの『<strings.h>』

strncat() ― ストリングの連結

フォーマット

```
#include <string.h>
char *strncat(char *string1, const char *string2, size_t count);
```

言語レベル: ANSI

スレッド・セーフ: はい。

説明

`strncat()` 関数は、*string2* の最初の *count* 文字を *string1* に付加し、結果のストリングをヌル文字 (¥0) で終了します。 *count* が *string2* の長さより大きい場合には、 *count* の代わりに *string2* の長さが使用されます。

`strncat()` 関数は、ヌル終了ストリング上で作動します。関数のストリング引数には、ストリングの終わりを示すマークであるヌル文字 (¥0) が含まれていなければなりません。

戻り値

`strncat()` 関数は、結合されたストリング (*string1*) へのポインターを戻します。

`strncat()` の使用例

この例では、`strcat()` と `strncat()` の違いを説明します。 `strcat()` 関数は 2 番目のストリング全体を最初のストリングに追加し、`strncat()` は 2 番目のストリング内の指定された文字数のみを最初のストリングに追加します。

```
#include <stdio.h>
#include <string.h>

#define SIZE 40

int main(void)
{
  char buffer1[SIZE] = "computer";
  char * ptr;

  /* Call strcat with buffer1 and " program" */

  ptr = strcat( buffer1, " program" );
  printf( "strcat : buffer1 = ¥"%s¥"¥n", buffer1 );

  /* Reset buffer1 to contain just the string "computer" again */

  memset( buffer1, '¥0', sizeof( buffer1 ));
  ptr = strcpy( buffer1, "computer" );

  /* Call strncat with buffer1 and " program" */
  ptr = strncat( buffer1, " program", 3 );
  printf( "strncat: buffer1 = ¥"%s¥"¥n", buffer1 );
}

/*****************  Output should be similar to:  *****************

strcat : buffer1 = "computer program"
strncat: buffer1 = "computer pr"
*/
```

関連情報

- 374 ページの『strcat() — ストリングの連結』
- 『strncmp() — ストリングの比較』
- 397 ページの『strncpy() — ストリングのコピー』
- 401 ページの『strpbrk() — ストリング内の文字の検索』
- 406 ページの『strrchr() — ストリング内で文字が最後に現れる位置の検出』
- 407 ページの『strspn() —最初の不一致文字のオフセットの検索』
- 471 ページの『wcscat() — ワイド文字ストリングの連結』
- 484 ページの『wcsncat() — ワイド文字ストリングの連結』
- 19 ページの『<string.h>』

strncmp() — ストリングの比較

フォーマット

```
#include <string.h>
int strncmp(const char *string1, const char *string2, size_t count);
```

言語レベル: ANSI

スレッド・セーフ: はい。

説明

`strncmp()` 関数は、*string1* および *string2* を *count* の最大数まで比較します。

戻り値

strncmp() 関数は、2 つのストリング間の関係を示す次のような値を戻します。

値	意味
0 より小さい値	*string1* は *string2* より小さい
0	*string1* は *string2* と等しい
0 より大きい値	*string1* は *string2* より大きい

strncmp() の使用例

この例では、strcmp() 関数と strncmp() 関数の違いを説明します。

```
#include <stdio.h>
#include <string.h>

#define SIZE 10

int main(void)
{
  int  result;
  int  index = 3;
  char buffer1[SIZE] = "abcdefg";
  char buffer2[SIZE] = "abcfg";
  void print_result( int, char *, char * );

  result = strcmp( buffer1, buffer2 );
  printf( "Comparison of each character¥n" );
  printf( "  strcmp: " );
  print_result( result, buffer1, buffer2 );

  result = strncmp( buffer1, buffer2, index);
  printf( "¥nComparison of only the first %i characters¥n", index );
  printf( "  strncmp: " );
  print_result( result, buffer1, buffer2 );
}

void print_result( int res, char * p_buffer1, char * p_buffer2 )
{
  if ( res == 0 )
    printf( "¥"%s¥" is identical to ¥"%s¥"¥n", p_buffer1, p_buffer2);
  else if ( res < 0 )
    printf( "¥"%s¥" is less than ¥"%s¥"¥n", p_buffer1, p_buffer2 );
  else
    printf( "¥"%s¥" is greater than ¥"%s¥"¥n", p_buffer1, p_buffer2 );
}

/*****************  Output should be similar to:  *****************

Comparison of each character
  strcmp: "abcdefg" is less than "abcfg"

Comparison of only the first 3 characters
  strncmp: "abcdefg" is identical to "abcfg"
*/
```

関連情報

- 376 ページの『strcmp() ― ストリングの比較』
- 381 ページの『strcspn() ― 最初に一致した文字のオフセットの検索』
- 394 ページの『strncat() ― ストリングの連結』

- 『strncpy() ― ストリングのコピー』
- 401 ページの『strpbrk() ― ストリング内の文字の検索』
- 406 ページの『strrchr() ― ストリング内で文字が最後に現れる位置の検出』
- 407 ページの『strspn() ―最初の不一致文字のオフセットの検索』
- 474 ページの『wcscmp() ― ワイド文字ストリングの比較』
- 485 ページの『wcsncmp() ― ワイド文字ストリングの比較』
- 19 ページの『<string.h>』
- 480 ページの『__wcsicmp() ― 大/小文字の区別をしないワイド文字ストリングの比較』
- 488 ページの『__wcsnicmp() ― 大/小文字の区別をしないワイド文字ストリングの比較』

strncpy() ― ストリングのコピー

フォーマット

```
#include <string.h>
char *strncpy(char *string1, const char *string2, size_t count);
```

言語レベル: ANSI

スレッド・セーフ: はい。

説明

strncpy() 関数では、*string2* の *count* 文字が、*string1* にコピーされます。*count* が *string2* の長さより小さいか等しい場合には、ヌル文字 (¥0) は、コピー・ストリングに付加されません。*count* が *string2* の長さより大きい場合、 *string1* 結果は *count* の長さになるまでヌル文字 (¥0) が埋め込まれます。

戻り値

strncpy() 関数は *string1* へのポインターを戻します。

strncpy() の使用例

この例では、strcpy() と strncpy() の違いを説明します。

```
#include <stdio.h>
#include <string.h>

#define SIZE    40

int main(void)
{
  char source[ SIZE ] = "123456789";
  char source1[ SIZE ] = "123456789";
  char destination[ SIZE ] = "abcdefg";
  char destination1[ SIZE ] = "abcdefg";
  char * return_string;
  int    index = 5;

  /* This is how strcpy works */
  printf( "destination is originally = '%s'¥n", destination );
  return_string = strcpy( destination, source );
  printf( "After strcpy, destination becomes '%s'¥n¥n", destination );


  /* This is how strncpy works */
  printf( "destination1 is originally = '%s'¥n", destination1 );
  return_string = strncpy( destination1, source1, index );
  printf( "After strncpy, destination1 becomes '%s'¥n", destination1 );
}

/*****************  Output should be similar to:  *****************

destination is originally = 'abcdefg'
After strcpy, destination becomes '123456789'

destination1 is originally = 'abcdefg'
After strncpy, destination1 becomes '12345fg'
*/
```

関連情報

- 380 ページの『strcpy() ― ストリングのコピー』
- 381 ページの『strcspn() ― 最初に一致した文字のオフセットの検索』
- 394 ページの『strncat() ― ストリングの連結』
- 395 ページの『strncmp() ― ストリングの比較』
- 401 ページの『strpbrk() ― ストリング内の文字の検索』
- 406 ページの『strrchr() ― ストリング内で文字が最後に現れる位置の検出』
- 407 ページの『strspn() ―最初の不一致文字のオフセットの検索』
- 19 ページの『<string.h>』

strnicmp - 大/小文字の区別をしないサブストリングの比較

フォーマット

```
#include <string.h>
int strnicmp(const char *string1, const char *string2, int n);
```

注: strnset 関数と strset 関数は、 C++ プログラムで使用可能です。__cplusplus__strings__ マクロがプログラムで定義されている場合にのみ、C でも使用できます。

言語レベル: Extension

スレッド・セーフ: はい。

ロケール依存: この関数の振る舞いは、現行ロケールの LC_CTYPE カテゴリーの影響を受ける可能性があります。詳細については、 549 ページの『CCSID およびロケールの理解』を参照してください。

説明

`strnicmp` は、*string1* と *string2* の最初の最大 *n* 文字を大/小文字の区別なしで比較します。

この関数は、ヌル終了ストリング上で作動します。関数のストリング引数には、ストリングの終わりを示すマークであるヌル文字 (¥0) が含まれると想定されます。

戻り値

`strnicmp` は、サブストリング間の関係を示す次のような値を戻します。

値	意味
0 より小さい値	*substring1* は *substring2* より小さい
0	*substring1* は *substring2* と等しい
0 より大きい値	*substring1* は *substring2* より大きい

strnicmp の使用例

この例では `strnicmp` を使用して、2 つのストリングを比較します。

```
#include <stdio.h>
#include <string.h>
int main(void)
{
   char *str1 = "THIS IS THE FIRST STRING";
   char *str2 = "This is the second string";
   int numresult;
     /* Compare the first 11 characters of str1 and str2
        without regard to case                                       */
   numresult = strnicmp(str1, str2, 11);
   if (numresult < 0)
      printf("String 1 is less than string2.\n");
   else
      if (numresult > 0)
         printf("String 1 is greater than string2.\n");
      else
         printf("The two strings are equivalent.\n");
   return 0;
}
```

The output should be:

```
      The two strings are equivalent.
```

関連情報

- 376 ページの『strcmp() — ストリングの比較』
- 378 ページの『strcmpi() - 大/小文字を区別しないストリングの比較』
- 390 ページの『stricmp() - 大/小文字を区別しないストリングの比較』
- 395 ページの『strncmp() — ストリングの比較』
- 474 ページの『wcscmp() — ワイド文字ストリングの比較』
- 485 ページの『wcsncmp() — ワイド文字ストリングの比較』
- 19 ページの『<string.h>』

strnset - strset - ストリング内の文字の設定

フォーマット

```
#include <string.h>
char *strnset(char *string, int c, size_t n);
char *strset(char *string, int c);
```

注: strnset 関数と strset 関数は、 C++ プログラムで使用可能です。__cplusplus__strings__ マクロがプログラムで定義されている場合にのみ、C でも使用できます。

言語レベル: Extension

スレッド・セーフ: はい。

説明

strnset は、*string* の最初の最大 *n* 文字を (char に変換された) *c* に設定します。*n* がストリングの長さより大きい場合には、*n* の代わりにストリングの長さが使用されます。 strset は、終了ヌル文字 (¥0) を除き、*string* のすべての文字を (char に変換された) *c* に設定します。どちらの関数も、ストリングはヌル終了ストリングです。

戻り値

strset と strnset は両方とも、変更されたストリングへのポインターを戻します。エラーの戻り値はありません。

strnset() および strset() の使用例

この例では、strnset はストリングの 4 文字までを文字「x」に設定します。次に、strset 関数はストリングのヌル以外の文字を文字「k」に変更します。

```
#include <stdio.h>
#include <string.h>
int main(void)
{
   char str[] = "abcdefghi";
   printf("This is the string: %s\n", str);
   printf("This is the string after strnset: %s\n", strnset((char*)str, 'x', 4));
   printf("This is the string after strset: %s\n", strset((char*)str, 'k'));
   return 0;
}
```

The output should be:

```
      This is the string: abcdefghi
      This is the string after strnset: xxxxefghi
      This is the string after strset: kkkkkkkkk
```

関連情報

- 375 ページの『strchr() ― 文字の検索』
- 401 ページの『strpbrk() ― ストリング内の文字の検索』
- 472 ページの『wcschr() ― ワイド文字の検索』
- 490 ページの『wcspbrk() ― ストリング内のワイド文字の位置検出』
- 19 ページの『<string.h>』

strpbrk() ― ストリング内の文字の検索

フォーマット

```
#include <string.h>
char *strpbrk(const char *string1, const char *string2);
```

言語レベル: ANSI

スレッド・セーフ: はい。

ロケール依存: この関数の振る舞いは、現行ロケールの LC_CTYPE カテゴリーの影響を受ける可能性があります。詳細については、 549 ページの『CCSID およびロケールの理解』を参照してください。

説明

`strpbrk()` 関数は、*string2* が示すストリングの任意の文字が、*string1* が示すストリングで最初に現れる位置を見つけます。

戻り値

`strpbrk()` 関数は、その文字へのポインターを戻します。*string1* と *string2* に共通の文字がない場合には、NULL ポインターが戻されます。

`strpbrk()` の使用例

この例は、配列 *string* での a または b のいずれかが最初に現れる位置へのポインターを戻します。

```
#include <stdio.h>
#include <string.h>

int main(void)
{
   char *result, *string = "A Blue Danube";
   char *chars = "ab";

   result = strpbrk(string, chars);
   printf("The first occurrence of any of the characters ¥"%s¥" in "
          "¥"%s¥" is ¥"%s¥"¥n", chars, string, result);
}

/*****************  Output should be similar to:  *****************

The first occurrence of any of the characters "ab" in "The Blue Danube"
is "anube"
*/
```

関連情報

- 375 ページの『strchr() ― 文字の検索』
- 376 ページの『strcmp() ― ストリングの比較』
- 381 ページの『strcspn() ― 最初に一致した文字のオフセットの検索』
- 395 ページの『strncmp() ― ストリングの比較』
- 406 ページの『strrchr() ― ストリング内で文字が最後に現れる位置の検出』
- 407 ページの『strspn() ―最初の不一致文字のオフセットの検索』
- 472 ページの『wcschr() ― ワイド文字の検索』
- 477 ページの『wcscspn() ― 最初に一致したワイド文字のオフセットの検索』

- 490 ページの『wcspbrk() ― ストリング内のワイド文字の位置検出』
- 493 ページの『wcsrchr() ― ストリング内でワイド文字が最後に現れる位置の検出』
- 510 ページの『wcswcs() ― ワイド文字サブストリングの位置検出』
- 19 ページの『<string.h>』

strptime()― ストリングから日付/時刻への変換

フォーマット

```
#include <time.h>
char *strptime(const char *buf, const char *format, struct tm *tm);
```

言語レベル: XPG4

スレッド・セーフ: はい。

ロケール依存: この関数の振る舞いは、現行ロケールの LC_CTYPE、LC_TIME、および LC_TOD カテゴリーの影響を受ける可能性があります。この関数は、コンパイル・コマンドに対して LOCALETYPE (*CLD) が指定されている場合には使用できません。詳細については、 549 ページの『CCSID およびロケールの理解』を参照してください。

説明

`strptime()` 関数は、*format* で指定された形式を使用し、 *buf* により示される文字ストリングを、*tm* で示される *tm* 構造体に保管される値に変換します。

format には、ゼロ以上のディレクティブが含まれます。1 つのディレクティブには、普通文字 (% または空白文字以外) または変換指定が含まれます。それぞれの変換指定は、% 文字の後に変換文字が 1 つ以上続く形で構成されています。これによって、必要な置き換えを指定します。関数が予想どおりに振る舞うことを保証するには、 *buf* と *format* の両方に空白文字またはその他の区切り文字がなければなりません。2 つの string-to-number 変換の間には区切り文字がなければなりません。そうしないと、最初の数値変換が、2 番目の変換指定子に属する文字を変換する可能性があります。

ディレクティブが書式ストリングまたは入力ストリングのいずれかで走査される前に検出される空白文字 (`isspace()` で指定される) は、無視されます。普通文字のディレクティブは、入力ストリング内の次の走査済み文字と完全に一致しなければなりません。普通文字ディレクティブのマッチング時には、大/小文字が区別されます。書式ストリングの普通文字ディレクティブが入力ストリングの文字と一致しない場合、`strptime` は正常に行われていません。文字はこれ以上走査されません。

その他の変換指定のマッチングは、入力ストリング内の文字を走査して、その指定で使用可能な文字でない文字が検出されるまで、または文字を走査できなくなるまで行われます。指定が string-to-number であった場合、有効な文字範囲は `+`、`-`、または `isdigit()` で指定される文字です。数値指定子は、先行ゼロが必要ありません。指定が現行ロケールのフィールドと一致する必要がある場合は、一致が検出されるまで走査が繰り返されます。ロケールのフィールドをマッチングするとき、大/小文字の区別は無視されます。一致が検出されると、 *tm* により示される構造体が、対応するロケール情報で更新されます。一致が検出されない場合には、`strptime` は正常に行われていません。文字はこれ以上走査されません。

tm 構造体内の欠落したフィールドは、十分な情報が提供されると、 `strftime` によって充てんされる場合があります。例えば、日付が指定されると、 `tm_yday` が計算されます。

標準変換指定は、それぞれ該当する文字で置き換えられます。次の表を参照してください。

指定子	意味
%a	曜日の名前で、フルネームまたは省略形にできます。
%A	%a と同じ。
%b	月の名前で、フルネームまたは省略形にできます。
%B	%b と同じ。
%c	ロケールの形式の日付/時刻。
%C	世紀数 [00-99]。2 桁の年が使用される場合に年を計算します。
%d	日 [1-31]。
%D	日付形式で、%m/%d/%y と同じ。
%e	%d と同じ。
%g	ISO 日付の 2 桁の年の部分 [00-99]。
%G	ISO 日付の 4 桁の年の部分。負になる場合があります。
%h	%b と同じ。
%H	24 時間形式の時間 [0-23]。
%I	12 時間形式の時間 [1-12]。
%j	ユリウス日付 [1-366]。
%m	月 [1-12]。
%M	分 [0-59]。
%n	改行文字が検出されるまで、すべての空白文字をスキップします。
%p	12 時間形式が使用されている場合に時間の計算に使用される AM または PM ストリング。
%r	ロケールの AM/PM 形式の時刻。ロケールの時刻形式で使用できない場合、POSIX 時刻 AM/PM 形式である %I:%M:%S %p をデフォルトとします。
%R	秒を含まない 24 時間の時刻形式で、 %H:%M と同じ。
%S	秒 [00-61]。秒の範囲は、うるう秒および二重うるう秒を考慮に入れています。
%t	タブ文字が検出されるまで、すべての空白文字をスキップします。
%T	秒を含む 24 時間の時刻形式で、 %H:%M:%S と同じ。
%u	曜日 [1-7]。月曜が 1 で、日曜が 7。
%U	日曜を週の初日とした 1 年の週数 [0-53]。ユリウス日付を計算する際に使用します。
%V	1 年の ISO 週数 [1-53]。月曜が週の初日。1 月 1 日の週に新年の 4 日以上が含まれる場合、この週は週 1 と考慮されます。そうでない場合、この週は前年の最終週となり、次の週が新年の週 1 となります。ユリウス日付を計算する際に使用します。
%w	曜日 [0 -6]。日曜が 0。
%W	1 年の週数 [0-53]。月曜が週の初日。ユリウス日付を計算する際に使用します。
%x	ロケールの形式の日付。
%X	ロケールの形式の時刻。
%y	2 桁の年 [0-99]。
%Y	4 桁の年。負になる場合があります。
%z	UTC オフセット。出力は、+HHMM または -HHMM の形式のストリングで、+ は GMT の東、- は GMT の西、HH は GMT からの時間数、MM は GMT からの分数をそれぞれ表します。
%Z	時間帯名。
%%	% 文字。

変更された変換指定子

一部の変換指定子は、修飾子文字 E または O によって変更できます。これによって、代替の形式または指定を使用するよう指示できます。変換された変換指定子が現行ロケールで使用不可になっているフィールドを使用する場合、振る舞いは未変更変換指定が使用されたかのようになります。例えば、*era* ストリングが空ストリング "" (*era* は使用不可であることを意味します) である場合、 %EY が %Y のように振る舞います。

指定子	意味
%Ec	現在の時代の日付/時刻。
%EC	年代名。
%Ex	現在の時代の日付。
%EX	現在の時代の時刻。
%Ey	年代の年。これは基本の年からのオフセットです。
%EY	現在の時代の年。
%Od	代替数字を使用した日。
%Oe	%Od と同じ。
%OH	代替数字を使用した 24 時間形式の時間。
%OI	代替数字を使用した 12 時間形式の時間。
%Om	代替数字を使用した月。
%OM	代替数字を使用した分。
%OS	代替数字を使用した秒。
%Ou	代替数字を使用した曜日。月曜が 1 で、日曜が 7。
%OU	代替数字を使用した 1 年の週数。日曜が週の初日。
%OV	代替数字を使用した 1 年の ISO 週数。 ISO 週数の説明については、%V を参照してください。
%Ow	代替数字を使用した曜日。日曜が 0 で、土曜が 6。
%OW	代替数字を使用した 1 年の週数。月曜が週の初日。
%Oy	代替数字を使用した 2 桁の年。
%OZ	時間帯の省略名。

戻り値

正常終了の場合は、`strptime()` 関数は、解析済みの最終文字の次の文字へのポインターを戻します。それ以外の場合は、NULL ポインターが戻されます。 `errno` の値は **ECONVERT** (変換エラー) に設定される可能性があります。

strptime() の使用例

```
#include <stdio.h>
#include <locale.h>
#include <time.h>

int main(void)
{
    char buf[100];
    time_t t;
    struct tm *timeptr,result;

    setlocale(LC_ALL,"/QSYS.LIB/EN_US.LOCALE");
```

```
    t = time(NULL);
    timeptr = localtime(&t);
    strftime(buf,sizeof(buf), "%a %m/%d/%Y %r", timeptr);

    if(strptime(buf, "%a %m/%d/%Y %r",&result) == NULL)
          printf("¥nstrptime failed¥n");
   else
   {
          printf("tm_hour:  %d¥n",result.tm_hour);
          printf("tm_min:  %d¥n",result.tm_min);
          printf("tm_sec:  %d¥n",result.tm_sec);
          printf("tm_mon:  %d¥n",result.tm_mon);
          printf("tm_mday:  %d¥n",result.tm_mday);
          printf("tm_year:  %d¥n",result.tm_year);
          printf("tm_yday:  %d¥n",result.tm_yday);
          printf("tm_wday:  %d¥n",result.tm_wday);
   }

   return 0;
}

/************************************************************
     The output should be similar to:
     Tue 10/30/2001 10:59:10 AM
     tm_hour:  10
     tm_min:  59
     tm_sec:  10
     tm_mon:  9
     tm_mday:  30
     tm_year:  101
     tm_yday:  302
     tm_wday:  2
************************************************************/
```

関連情報

- 41 ページの『asctime() ― 時間から文字ストリングへの変換』
- 43 ページの『asctime_r() ― 時間から文字ストリングへの変換 (再始動可能)』
- 75 ページの『ctime() ― 時間から文字ストリングへの変換』
- 77 ページの『ctime64() ― 時間から文字ストリングへの変換』
- 80 ページの『ctime64_r() ― 時間から文字ストリングへの変換 (再始動可能)』
- 78 ページの『ctime_r() ― 時間から文字ストリングへの変換 (再始動可能)』
- 167 ページの『gmtime() ― 時間の変換』
- 169 ページの『gmtime64() ― 時間の変換』
- 173 ページの『gmtime64_r() ― 時間の変換 (再始動可能)』
- 171 ページの『gmtime_r() ― 時間の変換 (再始動可能)』
- 192 ページの『localtime() ― 時間の変換』
- 194 ページの『localtime64() ― 時間の変換』
- 197 ページの『localtime64_r() ― 時間の変換 (再始動可能)』
- 195 ページの『localtime_r() ― 時間の変換 (再始動可能)』
- 354 ページの『setlocale() ― ロケールの設定』
- 387 ページの『strftime() ― 日付/時刻からストリングへの変換』
- 429 ページの『time() ― 現在時刻の判別』
- 430 ページの『time64() ― 現在時刻の判別』

- 19 ページの『<time.h>』
- 491 ページの『wcsptime()— ワイド文字ストリングから日付/時刻への変換』

strrchr() — ストリング内で文字が最後に現れる位置の検出

フォーマット

```
#include <string.h>
char *strrchr(const char *string, int c);
```

言語レベル: ANSI

スレッド・セーフ: はい。

ロケール依存: この関数の振る舞いは、現行ロケールの LC_CTYPE カテゴリーの影響を受ける可能性があります。詳細については、 549 ページの『CCSID およびロケールの理解』を参照してください。

説明

strrchr() 関数は、*string* 内で (文字に変換された) *c* が最後に現れる位置を検索します。終了ヌル文字は、 *string* の一部と見なされます。

戻り値

strrchr() 関数は、*string* 内の *c* が最後に現れる位置へのポインターを戻します。指定文字が検出されない場合は、NULL ポインターが戻されます。

strrchr() の使用例

この例は、strchr() と strrchr() の使用法を比較します。ストリングを調べて、ストリング内で p が最初に現れる位置と最後に現れる位置を検索します。

```
#include <stdio.h>
#include <string.h>

#define SIZE 40

int main(void)
{
  char buf[SIZE] = "computer program";
  char * ptr;
  int    ch = 'p';

  /* This illustrates strchr */
  ptr = strchr( buf, ch );
  printf( "The first occurrence of %c in '%s' is '%s'¥n", ch, buf, ptr );

  /* This illustrates strrchr */
  ptr = strrchr( buf, ch );
  printf( "The last occurrence of %c in '%s' is '%s'¥n", ch, buf, ptr );
}

/*****************  Output should be similar to:  *****************

The first occurrence of p in 'computer program' is 'puter program'
The last occurrence of p in 'computer program' is 'program'
*/
```

関連情報

- 375 ページの『strchr() ― 文字の検索』
- 376 ページの『strcmp() ― ストリングの比較』
- 381 ページの『strcspn() ― 最初に一致した文字のオフセットの検索』
- 395 ページの『strncmp() ― ストリングの比較』
- 401 ページの『strpbrk() ― ストリング内の文字の検索』
- 『strspn() ―最初の不一致文字のオフセットの検索』
- 19 ページの『<string.h>』

strspn() ―最初の不一致文字のオフセットの検索

フォーマット

```
#include <string.h>
size_t strspn(const char *string1, const char *string2);
```

言語レベル: ANSI

スレッド・セーフ: はい。

ロケール依存: この関数の振る舞いは、現行ロケールの LC_CTYPE カテゴリーの影響を受ける可能性があります。詳細については、 549 ページの『CCSID およびロケールの理解』を参照してください。

説明

`strspn()` 関数は、*string2* で指定される文字のセットに含まれない、 *string1* 内文字が最初に現れる位置を検索します。 *string2* を終了するヌル文字 (¥0) は、一致プロセスでは考慮に入れられません。

戻り値

`strspn()` 関数は、検出された最初の文字のインデックスを戻します。この値は、*string2* からの文字のみで構成される *string1* の初期サブストリングの長さと等しいです。*string1* が *string2* 内にない文字で始まる場合、`strspn()` 関数は 0 を戻します。*string1* 内のすべての文字が *string2* で検出される場合、*string1* の長さが戻されます。

`strspn()` の使用例

この例は、配列 *string* で、a、b、または c 以外が最初に現れる位置を検索します。この例のストリングは、 cabbage であるため、`strspn()` 関数は 5 (a、b、または c 以外の文字の前の cabbage のセグメントの長さ) を戻します。

```
#include <stdio.h>
#include <string.h>

int main(void)
{
  char * string = "cabbage";
  char * source = "abc";
  int index;

  index = strspn( string, "abc" );
  printf( "The first %d characters of ¥"%s¥" are found in ¥"%s¥"¥n",
          index, string, source );
}

/*****************  Output should be similar to:  *****************

The first 5 characters of "cabbage" are found in "abc"
*/
```

関連情報

- 374 ページの『strcat() — ストリングの連結』
- 375 ページの『strchr() — 文字の検索』
- 376 ページの『strcmp() — ストリングの比較』
- 380 ページの『strcpy() — ストリングのコピー』
- 381 ページの『strcspn() — 最初に一致した文字のオフセットの検索』
- 401 ページの『strpbrk() — ストリング内の文字の検索』
- 406 ページの『strrchr() — ストリング内で文字が最後に現れる位置の検出』
- 472 ページの『wcschr() — ワイド文字の検索』
- 477 ページの『wcscspn() — 最初に一致したワイド文字のオフセットの検索』
- 490 ページの『wcspbrk() — ストリング内のワイド文字の位置検出』
- 496 ページの『wcsspn() — 最初の不一致ワイド文字のオフセットの検索』
- 510 ページの『wcswcs() — ワイド文字サブストリングの位置検出』
- 493 ページの『wcsrchr() — ストリング内でワイド文字が最後に現れる位置の検出』
- 19 ページの『<string.h>』

strstr() — サブストリングの位置検出

フォーマット

```
#include <string.h>
char *strstr(const char *string1, const char *string2);
```

言語レベル: ANSI

スレッド・セーフ: はい。

説明

strstr() 関数は、*string1* で *string2* が最初に現れる位置を検索します。この関数は、一致プロセスで *string2* を終了するヌル文字 (¥0) を無視します。

戻り値

strstr() 関数は、*string1* 内での *string2* が最初に現れる位置の先頭へのポインターを戻します。*string2* が *string1* に現れないと、strstr() 関数は NULL を戻します。*string2* がゼロ長のストリングを指す場合には、strstr() 関数は *string1* を戻します。

strstr() の使用例

この例では、ストリング "needle in a haystack" の中から、ストリング ″haystack″ を見つけます。

```
#include <string.h>
#include <stdio.h>

int main(void)
{
   char *string1 = "needle in a haystack";
   char *string2 = "haystack";
   char *result;

  result = strstr(string1,string2);
     /* Result = a pointer to "haystack" */
  printf("%s¥n", result);
}

/*****************  Output should be similar to: *****************

haystack
*/
```

関連情報

- 375 ページの『strchr() ― 文字の検索』
- 376 ページの『strcmp() ― ストリングの比較』
- 381 ページの『strcspn() ― 最初に一致した文字のオフセットの検索』
- 395 ページの『strncmp() ― ストリングの比較』
- 401 ページの『strpbrk() ― ストリング内の文字の検索』
- 406 ページの『strrchr() ― ストリング内で文字が最後に現れる位置の検出』
- 407 ページの『strspn() ―最初の不一致文字のオフセットの検索』
- 472 ページの『wcschr() ― ワイド文字の検索』
- 477 ページの『wcscspn() ― 最初に一致したワイド文字のオフセットの検索』
- 490 ページの『wcspbrk() ― ストリング内のワイド文字の位置検出』
- 493 ページの『wcsrchr() ― ストリング内でワイド文字が最後に現れる位置の検出』
- 496 ページの『wcsspn() ― 最初の不一致ワイド文字のオフセットの検索』
- 510 ページの『wcswcs() ― ワイド文字サブストリングの位置検出』
- 19 ページの『<string.h>』

strtod() ― strtof() ― strtold ― 文字ストリングから double、浮動、および long double への変換

フォーマット

```
#include <stdlib.h>
double strtod(const char *nptr, char **endptr);
float strtof(const char *nptr, char **endptr);
long double strtold(const char *nptr, char **endptr);
```

言語レベル: ANSI

スレッド・セーフ: はい。

ロケール依存: これらの関数の振る舞いは、現行ロケールの LC_CTYPE カテゴリーおよび LC_NUMERIC カテゴリーの影響を受ける可能性があります。詳細については、 549 ページの『CCSID およびロケールの理解』を参照してください。

説明

strtod()、strtof()、および strtold() 関数は、文字ストリングを double、浮動、または long double の値に変換します。パラメーター *nptr* は、数値バイナリー浮動小数点値として解釈できる文字のシーケンスを示します。これらの関数によって、数値の一部として認識できないストリングは、先頭文字のところで読み取りが停止されます。この文字は、ストリングの最後にあるヌル文字である場合もあります。

strtod()、strtof()、および strtold() 関数では、*nptr* が以下の形式でストリングを指すことを想定しています。

最初の文字がこの形式に適合しない場合、走査は停止されます。さらに、INFINITY または NAN のシーケンス (大/小文字を区別しない) が許可されます。

戻り値

strtod()、strtof()、および strtold() 関数は、表現がアンダーフローまたはオーバーフローを引き起こす場合を除き、浮動小数点数の値を戻します。オーバーフローの場合、strtof() は HUGE_VALF または -HUGE_VALF を戻し、strtod() および strtold() は HUGE_VAL または -HUGE_VAL を戻します。アンダーフローの場合、すべての関数は 0 を戻します。

どちらの場合も、errno は ERANGE に設定されます。*nptr* が指すストリングが想定する形式でない場合、変換は行われません。 *nptr* の値は、*endptr* が NULL ポインターでない場合に、 *endptr* が指すオブジェクトに保管されます。

数字以外の文字が、指数として読み取られる E または e に続く場合、strtod()、strtof()、および strtold() 関数は失敗しません。例えば、100elf は浮動小数点値 100.0 に変換されます。

INFINITY (大/小文字を区別しない) の文字シーケンスは、INFINITY の値をもたらします。NAN の文字値は、Quiet Not-A-Number (NAN) 値をもたらします。

strtod()、strtof()、および strtold() の使用例

この例では、ストリングを double、浮動、および long double 値に変換します。変換済みの値および変換を停止したサブストリングを出力します。

```
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
   char *string, *stopstring;
   double x;
   float f;
   long double ld;

   string = "3.1415926This stopped it";
   f = strtof(string, &stopstring);
  printf("string = %s¥n", string);
  printf("  strtof = %f¥n", f);
  printf("Stopped scan at ¥"%s¥"¥n¥n", stopstring);

  string = "3.1415926This stopped it";
   x = strtod(string, &stopstring);
   printf("string = %s¥n", string);
   printf("   strtod = %f¥n", x);
   printf("   Stopped scan at %s¥n¥n", stopstring);
   string = "100ergs";
   x = strtod(string, &stopstring);
   printf("string = ¥"%s¥"¥n", string);
   printf("   strtod = %f¥n", x);
   printf("   Stopped scan at ¥"%s¥"¥n¥n", stopstring);

  string = "3.1415926This stopped it";
   ld = strtold(string, &stopstring);
   printf("string = %s¥n", string);
   printf("   strtold = %lf¥n", ld);
   printf("   Stopped scan at %s¥n¥n", stopstring);
   string = "100ergs";
   ld = strtold(string, &stopstring);
   printf("string = ¥"%s¥"¥n", string);
   printf("   strtold = %lf¥n", ld);
   printf("   Stopped scan at ¥"%s¥"¥n¥n", stopstring);
}

/*****************  Output should be similar to:  *****************
string = 3.1415926This stopped it
   strtof = 3.141593
   Stopped scan at This stopped it

string = "100ergs
   strtof = 100.000000
   Stopped scan at "erg

string = 3.1415926This stopped it
   strtod = 3.141593
   Stopped scan at This stopped it

string = "100ergs
   strtod = 100.000000
   Stopped scan at "erg

string = 3.1415926This stopped it
   strtold = 3.141593
   Stopped scan at This stopped it

string = "100ergs
   strtold = 100.000000
   Stopped scan at "erg

*/
```

関連情報

- 49 ページの『atof() ― 文字ストリングから浮動小数点への変換』
- 50 ページの『atoi() ― 文字ストリングから整数への変換』
- 52 ページの『atol() ― atoll() ― 文字ストリングの long 型整数または long long 型整数への変換』
- 『strtod32() ― strtod64() ― strtod128() ― 文字ストリングから 10 進浮動小数点への変換』
- 418 ページの『strtol() ― strtoll() ― 文字ストリングから long 型および long long 型整数への変換』
- 420 ページの『strtoul() ― strtoull() ― 文字ストリングから符号なし long 型整数および符号なし long long 型整数への変換』
- 498 ページの『wcstod() ― ワイド文字ストリングから double 型への変換』
- 499 ページの『wcstod32() ― wcstod64() ― wcstod128()―ワイド文字ストリングから 10 進浮動小数点への変換』
- 18 ページの『<stdlib.h>』

strtod32() ― strtod64() ― strtod128() ― 文字ストリングから 10 進浮動小数点への変換

フォーマット

```
#include <stdlib.h>
_Decimal32 strtod32(const char *nptr, char **endptr);
_Decimal64 strtod64(const char *nptr, char **endptr);
_Decimal128 strtod132(const char *nptr, char **endptr);
```

言語レベル: ANSI

スレッド・セーフ: はい。

ロケール依存: これらの関数の振る舞いは、現行ロケールの LC_CTYPE カテゴリーおよび LC_NUMERIC カテゴリーの影響を受ける可能性があります。詳細については、 549 ページの『CCSID およびロケールの理解』を参照してください。

説明

strtod32()、strtod64()、および strtod128() 関数は、文字ストリングを単精度、倍精度、または 4 倍精度の 10 進浮動小数点値へ変換します。パラメーター *nptr* は、数値 10 進浮動小数点値として解釈できる文字のシーケンスを示します。これらの関数によって、数値の一部として認識できないストリングは、先頭文字のところで読み取りが停止されます。この文字は、ストリングの最後にあるヌル文字である場合もあります。 *endptr* パラメーターは、 *endptr* が NULL ポインターでない場合、この文字を指すように更新されます。

strtod32()、strtod64()、および strtod128() 関数は、*nptr* が以下の形式のストリングを指すことを想定します。

最初の文字がこの形式に適合しない場合、走査は停止されます。さらに、INFINITY または NAN のシーケンス (大/小文字を区別しない) が許可されます。

戻り値

`strtod32()`、`strtod64()`、および `strtod128()` 関数は、表現がアンダーフローまたはオーバーフローを引き起こす場合を除き、浮動小数点数の値を戻します。オーバーフローの場合、`strtod32()` は HUGE_VAL_D32 または -HUGE_VAL_D32 を、`strtod64()` は HUGE_VAL_D64 または -HUGE_VAL_D64 を、`strtod128()` は HUGE_VAL_D128 または -HUGE_VAL_D128 をそれぞれ戻します。アンダーフローの場合、すべての関数は +0.E0 を戻します。

オーバーフローの場合もアンダーフローの場合も、errno は ERANGE に設定されます。*nptr* が指すストリングが想定する形式ではない場合、 +0.E0 の値が戻されます。*nptr* の値は、 *endptr* が指すオブジェクトに保管されます (但し、*endptr* が NULL ポインターでない場合に限られます)。

数字以外の文字が、指数として読み取られる E または e に続く場合、`strtod32()`、 `strtod64()`、および `strtod128()` 関数は失敗しません。例えば、100elf は浮動小数点値 100.0 に変換されます。

`INFINITY` (大/小文字を区別しない) の文字シーケンスは、INFINITY の値をもたらします。`NAN` の文字値は、Quiet Not-A-Number (NaN) 値をもたらします。

戻り値は必要に応じて、丸めモード Round to Nearest、Ties to Even を使用して丸められます。

`strtod32()`、`strtod64()`、および `strtod128()` の使用例

この例では、ストリングを単精度、倍精度、および 4 倍精度の 10 進浮動小数点値に変換します。変換済みの値および変換を停止したサブストリングを出力します。

```
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
    char *string, *stopstring;
    _Decimal32 d32;
    _Decimal64 d64;
    _Decimal128 d128;

  string = "3.1415926This stopped it";
   d32 = strtod32(string, &stopstring);
   printf("string = %s¥n", string);
   printf(" strtod32 = %Hf¥n", d32);
   printf(" Stopped scan at %s¥n¥n", stopstring);
   string = "100ergs";
   d32 = strtod32(string, &stopstring);
   printf("string = ¥"%s¥"¥n", string);
   printf(" strtof = %Hf¥n", d32);
   printf(" Stopped scan at ¥"%s¥"¥n¥n", stopstring);

  string = "3.1415926This stopped it";
   d64 = strtod64(string, &stopstring);
   printf("string = %s¥n", string);
   printf(" strtod = %Df¥n", d64);
   printf(" Stopped scan at %s¥n¥n", stopstring);
   string = "100ergs";
   d64 = strtod64(string, &stopstring);
   printf("string = ¥"%s¥"¥n", string);
   printf(" strtod = %Df¥n", d64);
   printf(" Stopped scan at ¥"%s¥"¥n¥n", stopstring);

  string = "3.1415926This stopped it";
   d128 = strtod128(string, &stopstring);
   printf("string = %s¥n", string);
   printf(" strtold = %DDf¥n", d128);
   printf(" Stopped scan at %s¥n¥n", stopstring);
   string = "100ergs";
   d128 = strtod128(string, &stopstring);
   printf("string = ¥"%s¥"¥n", string);
   printf(" strtold = %DDf¥n", d128);
   printf(" Stopped scan at ¥"%s¥"¥n¥n", stopstring);
}

/***************** Output should be similar to: *****************

string = 3.1415926This stopped it
 strtof = 3.141593
 Stopped scan at This stopped it

string = "100ergs"
 strtof = 100.000000
 Stopped scan at "ergs"

string = 3.1415926This stopped it
 strtod= 3.141593
 Stopped scan at This stopped it

string = "100ergs"
 strtod = 100.000000
 Stopped scan at "ergs"
```

```
string = 3.1415926This stopped it
 strtold = 3.141593
 Stopped scan at This stopped it

string = "100ergs"
 strtold = 100.000000
 Stopped scan at "ergs"

*/
```

関連情報

- 49 ページの『atof() ― 文字ストリングから浮動小数点への変換』
- 50 ページの『atoi() ― 文字ストリングから整数への変換』
- 52 ページの『atol() ― atoll() ― 文字ストリングの long 型整数または long long 型整数への変換』
- 409 ページの『strtod() ― strtof() ― strtold ― 文字ストリングから double、浮動、および long double への変換』
- 418 ページの『strtol() ― strtoll() ― 文字ストリングから long 型および long long 型整数への変換』
- 420 ページの『strtoul() ― strtoull() ― 文字ストリングから符号なし long 型整数および符号なし long long 型整数への変換』
- 498 ページの『wcstod() ― ワイド文字ストリングから double 型への変換』
- 499 ページの『wcstod32() ― wcstod64() ― wcstod128()―ワイド文字ストリングから 10 進浮動小数点への変換』
- 18 ページの『<stdlib.h>』

strtok() ― ストリングのトークン化

フォーマット

```
#include <string.h>
char *strtok(char *string1, const char *string2);
```

言語レベル: ANSI

スレッド・セーフ: いいえ。代わりに `strtok_r()` を使用します。

ロケール依存: この関数の振る舞いは、現行ロケールの LC_CTYPE カテゴリーの影響を受ける可能性があります。詳細については、 549 ページの『CCSID およびロケールの理解』を参照してください。

説明

`strtok()` 関数は、*string1* をゼロ個以上のトークンのシリーズとして読み取り、 *string2* を *string1* 内のトークンの区切り文字として役立つ文字のセットとして読み取ります。*string1* 内のトークンは、 *string2* からの 1 つ以上の区切り文字によって分離することができます。*string1* 内のトークンは、`strtok()` 関数への一連の呼び出しによって見つかります。

指定された *string1* に関する `strtok()` 関数への最初の呼び出しで、 `strtok()` 関数は先行区切り文字をスキップして、*string1* 内の最初のトークンを検索します。最初のトークンへのポインターが戻されます。

`strtok()` 関数が NULL *string1* 引数で呼び出されると、次のトークンは最後のヌル以外の *string1* パラメーターの保管済みコピーから読み取られます。それぞれの区切り文字は、ヌル文字で置き換えられます。区切り文字のセットは呼び出しごとに異なるため、*string2* は任意の値を取ることができます。*string1* の初期値は、 `strtok()` 関数への呼び出し後は保存されないことに注意してください。

strtok() 関数は日付をバッファーに書き込むことに注意してください。この関数が受け渡されるバッファーは、strtok() 関数によって損傷を受けることになるため、トークン化されるストリングは、重要ではないバッファーに組み込まれるようにしてください。

戻り値

strtok() 関数の最初の呼び出し時に、 *string1* 内の最初のトークンへのポインターを戻します。同じトークン・ストリングでの以後の呼び出しで、strtok() 関数は、ストリング内の次のトークンへのポインターを戻します。トークンがなくなると、NULL ポインターが戻されます。すべてのトークンはヌル終了されます。

注: strtok() 関数は内部静的ポインターを使用して、トークン化されているストリング内の次のトークンを指します。 strtok() 関数の再入可能バージョン strtok_r() (内部静的ストレージを使用しない) を strtok() 関数の代わりに使用することができます。

strtok() の使用例

この例ではループを使用して、トークンが残らなくなるまでコンマで区切られたトークンをストリングから収集します。この例では、トークン a string、of、および tokens を出力します。

```
#include <stdio.h>
#include <string.h>

int main(void)
{
   char *token, *string = "a string, of, ,tokens¥0,after null terminator";

   /* the string pointed to by string is broken up into the tokens
      "a string", " of", " ", and "tokens" ; the null terminator (¥0)
      is encountered and execution stops after the token "tokens"     */
   token = strtok(string, ",");
   do
   {
      printf("token: %s¥n", token);
   }
   while (token = strtok(NULL, ","));
}

/*****************  Output should be similar to:  *****************

token: a string
token:  of
token:
token: tokens
*/
```

関連情報

- 374 ページの『strcat() — ストリングの連結』
- 375 ページの『strchr() — 文字の検索』
- 376 ページの『strcmp() — ストリングの比較』
- 380 ページの『strcpy() — ストリングのコピー』
- 381 ページの『strcspn() — 最初に一致した文字のオフセットの検索』
- 407 ページの『strspn() —最初の不一致文字のオフセットの検索』
- 417 ページの『strtok_r() — ストリングのトークン化 (再開可能)』
- 19 ページの『<string.h>』

strtok_r() ― ストリングのトークン化 (再開可能)

フォーマット

```
#include <string.h>
char *strtok_r(char *string, const char *seps,
               char **lasts);
```

言語レベル: XPG4

スレッド・セーフ: はい。

ロケール依存: この関数の振る舞いは、現行ロケールの LC_CTYPE カテゴリーの影響を受ける可能性があります。詳細については、 549 ページの『CCSID およびロケールの理解』を参照してください。

説明

この関数は、`strtok()` の再始動可能バージョンです。

`strtok_r()` 関数は、*string* をゼロ個以上のトークンのシリーズとして読み取り、 *seps* を *string* 内のトークンの区切り文字として役立つ文字のセットとして読み取ります。*string* 内のトークンは、 *seps* からの 1 つ以上の区切り文字によって分離することができます。引数 *lasts* は、`strtok_r()` 関数が同じストリングの走査を継続するために必要な格納情報を指す、ユーザー提供のポインターを指します。

指定されたヌル終了 *string* に関する `strtok_r()` 関数への最初の呼び出しで、先行区切り文字をスキップして、*string* 内の最初のトークンを検索します。最初のトークンの先頭文字へのポインターを戻し、戻されたトークンの直後の *string* にヌル文字を書き込み、 *lasts* が指すポインターを更新します。

| *string* からの次のトークンを読み取るには、NULL *string* 引数で `strtok_r()` 関数を呼び出します。これにより、`strtok_r()` 関数は前のトークン・ストリング内で次のトークンを検索します。元の *string* 内の各区切り文字は、ヌル文字に置き換えられ、*lasts* が指すポインターは更新されます。*seps* 内の区切り文字のセットは、呼び出しごとに異なりますが、*lasts* は前の呼び出しから変更しないで残す必要があります。*string* にトークンがなくなると、 NULL ポインターが戻されます。

戻り値

`strtok_r()` 関数の最初の呼び出し時に、 *string* の最初のトークンへのポインターを戻します。同じトークン・ストリングでの以後の呼び出しで、`strtok_r()` 関数は、ストリング内の次のトークンへのポインターを戻します。トークンがなくなると、NULL ポインターが戻されます。すべてのトークンはヌル終了されます。

関連情報

- 374 ページの『strcat() ― ストリングの連結』
- 375 ページの『strchr() ― 文字の検索』
- 376 ページの『strcmp() ― ストリングの比較』
- 380 ページの『strcpy() ― ストリングのコピー』
- 381 ページの『strcspn() ― 最初に一致した文字のオフセットの検索』
- 407 ページの『strspn() ―最初の不一致文字のオフセットの検索』
- 415 ページの『strtok() ― ストリングのトークン化』
- 19 ページの『<string.h>』

strtol() ― strtoll() ― 文字ストリングから long 型および long long 型整数への変換

フォーマット (strtol())

```
#include <stdlib.h>
long int strtol(const char *nptr, char **endptr, int base);
```

フォーマット (strtoll())

```
#include <stdlib.h>
long long int strtoll(char *string, char **endptr, int base);
```

言語レベル: ANSI

スレッド・セーフ: はい。

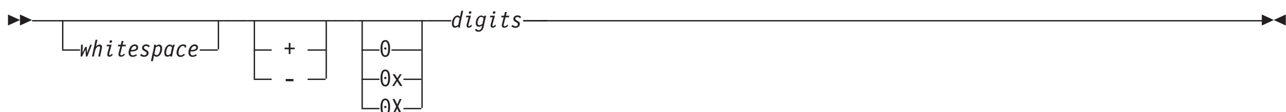
ロケール依存: これらの関数の振る舞いは、現行ロケールの LC_CTYPE カテゴリーの影響を受ける可能性があります。詳細については、 549 ページの『CCSID およびロケールの理解』を参照してください。

説明

strtol() 関数は、文字ストリングを long 型整数値に変換します。パラメーター *nptr* により、 long 型整数の数値として解釈できる文字のシーケンスが示されます。

strtoll() 関数は、文字ストリングを long long 型整数値に変換します。パラメーター *nptr* により、 long long 型整数の数値として解釈できる文字のシーケンスが示されます。

これらの関数を使用する場合は、*nptr* パラメーターによって、次の形式のストリングが指されている必要があります。



base パラメーターの値が 2 と 36 の間の場合には、サブジェクト・シーケンスの想定される形式は、基数が *base* パラメーターで指定される整数を表す文字と数字のシーケンスです。このシーケンスは、オプションで先頭に正符号 (+) または負符号 (-) が付きます。a から z までを含めた文字 (大文字または小文字) は、 10 から 35 までの値として見なされます。これらの値が base パラメーターの値より小さい文字だけが許可されます。base パラメーターの値が 16 の場合には、文字 0x または 0X はオプションで一連の文字および数字の前に付くことがありますが、正符号 (+) または負符号 (-) があるとその後に続きます。

base パラメーターの値が 0 である場合、ストリングは base を決定します。オプションの先行符号の後、先行 0 は 8 進変換を示し、先行 0x または 0X は 16 進変換を示し、その他のすべての先行文字は結果として 10 進変換になります。

これらの関数は、*base* パラメーターと不整合な最初の文字に達するまで、ストリングを走査します。この文字は、ストリングの最後にあるヌル文字 ('¥0') である場合もあります。先行空白文字は無視され、オプションの符号が数字の前に付く場合があります。

endptr パラメーターの値がヌル以外の場合、ポインター (走査を終了した文字へのポインター) は *endptr* が指す値に保管されます。値を形成できない場合、 *endptr* が指す値は *nptr* パラメーターに設定されます。

戻り値

base に無効値 (0 より小さい、1、または 36 より大きい) が含まれる場合、errno は EINVAL に設定され、0 が戻されます。*endptr* パラメーターが指す値は、 *nptr* パラメーターの値に設定されます。

この値が表示可能値の範囲外にある場合には、errno は ERANGE に設定されます。この値が正の場合、strtol() 関数は LONG_MAX を戻し、strtoll() 関数は LONGLONG_MAX を戻します。この値が負の場合、strtol() 関数は LONG_MIN を戻し、 strtoll() 関数は LONGLONG_MIN を戻します。

文字が変換されなかった場合、strtoll() 関数および strtol() 関数は errno を EINVAL に設定し、0 が戻されます。どちらの関数の場合も、 *endptr* が指す値は、 *nptr* パラメーターの値に設定されます。正常終了の場合は、両方の関数が変換済みの値を戻します。

strtol() の使用例

この例では、ストリングを long 値に変換します。変換済みの値、および変換を停止したサブストリングを出力します。

```
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
   char *string, *stopstring;
   long l;
   int bs;

   string = "10110134932";
   printf("string = %s¥n", string);
   for (bs = 2; bs <= 8; bs *= 2)
   {
      l = strtol(string, &stopstring, bs);
      printf("   strtol = %ld (base %d)¥n", l, bs);
      printf("   Stopped scan at %s¥n¥n", stopstring);
      }
}

/*****************  Output should be similar to:  *****************

string = 10110134932
   strtol = 45 (base 2)
   Stopped scan at 34932

   strtol = 4423 (base 4)
   Stopped scan at 4932
```

関連情報

- 49 ページの『atof() ― 文字ストリングから浮動小数点への変換』
- 50 ページの『atoi() ― 文字ストリングから整数への変換』
- 52 ページの『atol() ― atoll() ― 文字ストリングの long 型整数または long long 型整数への変換』
- 409 ページの『strtod() ― strtof() ― strtold ― 文字ストリングから double、浮動、および long double への変換』
- 412 ページの『strtod32() ― strtod64() ― strtod128() ― 文字ストリングから 10 進浮動小数点への変換』
- 420 ページの『strtoul() ― strtoull() ― 文字ストリングから符号なし long 型整数および符号なし long long 型整数への変換』

- 503 ページの『wcstol() ― wcstoll() ― ワイド文字ストリングから long 型および long long 型整数への変換』
- 18 ページの『<stdlib.h>』

strtoul() ― strtoull() ― 文字ストリングから符号なし long 型整数および符号なし long long 型整数への変換

フォーマット (strtoul())

```
#include <stdlib.h>
unsigned long int strtoul(const char *nptr, char **endptr, int base);
```

フォーマット (strtoull())

```
#include <stdlib.h>
unsigned long long int strtoull(char *string, char **endptr, int base);
```

言語レベル: ANSI

スレッド・セーフ: はい。

ロケール依存: これらの関数の振る舞いは、現行ロケールの LC_CTYPE カテゴリーの影響を受ける可能性があります。詳細については、 549 ページの『CCSID およびロケールの理解』を参照してください。

説明

`strtoul()` 関数は、文字ストリングを符号なし long 型整数値に変換します。パラメーター *nptr* により、符号なし long 型整数の数値として解釈できる文字のシーケンスが示されます。

`strtoull()` 関数は、文字ストリングを符号なし long long 型整数値に変換します。パラメーター *nptr* により、符号なし long long 型整数の数値として解釈できる文字のシーケンスが示されます。

これらの関数を使用する場合は、*nptr* パラメーターによって、次の形式のストリングが指されている必要があります。

base パラメーターの値が 2 と 36 の間の場合には、サブジェクト・シーケンスの想定される形式は、基数が base パラメーターで指定される整数を表す文字と数字のシーケンスです。このシーケンスは、オプションで先頭に正符号 (+) または負符号 (-) が付きます。a から z までを含めた文字 (大文字または小文字) は、 10 から 35 までの値と想定されます。想定された値が base パラメーターの値より小さい文字だけが許可されます。 *base* パラメーターの値が 16 の場合には、文字 `0x` または `0X` はオプションで一連の文字および数字の前に付くことがありますが、正符号 (+) または負符号 (-) があるとその後に続きます。

base パラメーターの値が 0 である場合、ストリングは *base* を決定します。オプションの先行符号の後、先行 0 は 8 進変換を示し、先行 `0x` または `0X` は 16 進変換を示し、その他のすべての先行文字は結果として 10 進変換になります。

これらの関数は、base パラメーターと不整合な最初の文字に達するまで、ストリングを走査します。この文字は、ストリングの最後にあるヌル文字 ('¥0') である場合もあります。先行空白文字は無視され、オプションの符号が数字の前に付く場合があります。

endptr パラメーターの値がヌル以外の場合、ポインター (走査を終了した文字へのポインター) は endptr が指す値に保管されます。値を形成できない場合、 *endptr* が指す値は *nptr* パラメーターに設定されます。

戻り値

base に無効値 (0 より小さい、1、または 36 より大きい) が含まれる場合、errno は EINVAL に設定され、0 が戻されます。 *endptr* パラメーターが指す値は、 *nptr* パラメーターの値に設定されます。

この値が表示可能値の範囲外にある場合には、errno は ERANGE に設定されます。strtoul() 関数は ULONG_MAX を戻し、 strtoull() 関数は ULONGLONG_MAX を戻します。

文字が変換されなかった場合、strtoull() 関数は errno を EINVAL に設定し、0 が戻されます。strtoul() 関数は 0 を戻しますが、errno を EINVAL に設定しません。どちらの場合も、 *endptr* が指す値は、 *nptr* パラメーターの値に設定されます。正常終了の場合は、両方の関数が変換済みの値を戻します。

strtoul() の使用例

この例では、ストリングを符号なし long 値に変換します。変換済みの値、および変換を停止したサブストリングを出力します。

```
#include <stdio.h>
#include <stdlib.h>

#define BASE 2

int main(void)
{
   char *string, *stopstring;
   unsigned long ul;

   string = "1000e13 e";
   printf("string = %s¥n", string);
   ul = strtoul(string, &stopstring, BASE);
   printf("   strtoul = %ld (base %d)¥n", ul, BASE);
   printf("   Stopped scan at %s¥n¥n", stopstring);
}

/*****************  Output should be similar to:  *****************

string = 1000e13 e
   strtoul = 8 (base 2)
   Stopped scan at e13 e
*/
```

関連情報

- 49 ページの『atof() ― 文字ストリングから浮動小数点への変換』
- 50 ページの『atoi() ― 文字ストリングから整数への変換』
- 52 ページの『atol() ― atoll() ― 文字ストリングの long 型整数または long long 型整数への変換』
- 409 ページの『strtod() ― strtof() ― strtold ― 文字ストリングから double、浮動、および long double への変換』

- 412 ページの『strtod32() ― strtod64() ― strtod128() ― 文字ストリングから 10 進浮動小数点への変換』
- 418 ページの『strtol() ― strtoll() ― 文字ストリングから long 型および long long 型整数への変換』
- 508 ページの『wcstoul() ― wcstoull() ― ワイド文字ストリングから符号なし long 型整数および符号なし long long 型整数への変換』
- 18 ページの『<stdlib.h>』

strxfrm() ― ストリングの変換

フォーマット

```
#include <string.h>
size_t strxfrm(char *string1, const char *string2, size_t count);
```

言語レベル: ANSI

スレッド・セーフ: はい。

ロケール依存: この関数の振る舞いは、現行ロケールの LC_CTYPE カテゴリーおよび LC_COLLATE カテゴリーの影響を受ける可能性があります。詳細については、 549 ページの『CCSID およびロケールの理解』を参照してください。

説明

strxfrm() 関数は、*string2* が指すストリングを変換し、その結果を *string1* が指すストリングに配置します。変換は、プログラムの現行ロケールにより判別されます。変換したストリングは必ずしも読み取り可能ではありませんが、strcmp() 関数または strncmp() 関数で使用できます。

戻り値

strxfrm() 関数は、終了ヌル文字以外の変換ストリングの長さを戻します。この戻り値が *count* 以上である場合、変換ストリングの内容は不確定となります。

strxfrm() が正常に実行されなかった場合、errno が変更されます。errno の値は EINVAL (*string1* または *string2* 引数に、現行ロケールで使用不可の文字が含まれている) に設定される可能性があります。

strxfrm() の使用例

この例では、文字のストリングを入力するようにユーザーにプロンプトを出し、strxfrm() を使用してストリングを変換し、その長さを戻します。

```
#include <stdio.h>
#include <string.h>

int main(void)
{
   char *string1, buffer[80];
   int length;

   printf("Type in a string of characters.¥n ");
   string1 = gets(buffer);
   length = strxfrm(NULL, string1, 0);
   printf("You would need a %d element array to hold the string¥n",length);
   printf("¥n¥n%s¥n¥n transformed according",string1);
   printf(" to this program's locale. ¥n");
}
```

関連情報

- 187 ページの『localeconv() ― 環境からの情報の取得』
- 354 ページの『setlocale() ― ロケールの設定』
- 376 ページの『strcmp() ― ストリングの比較』
- 379 ページの『strcoll() ― ストリングの比較』
- 395 ページの『strncmp() ― ストリングの比較』
- 19 ページの『<string.h>』

swprintf() ― ワイド文字のフォーマット設定とバッファーへの書き込み

フォーマット

```
#include <wchar.h>
int swprintf(wchar_t *wcsbuffer, size_t n,
             const wchar_t *format, argument-list);
```

言語レベル: ANSI

スレッド・セーフ: はい。

ロケール依存: この関数の振る舞いは、現行ロケールの LC_CTYPE カテゴリーおよび LC_NUMERIC カテゴリーの影響を受ける可能性があります。また、LOCALETYPE(*LOCALEUCS2) または LOCALETYPE(*LOCALEUTF) がコンパイル・コマンドで指定されている場合、この振る舞いは、現行ロケールの LC_UNI_CTYPE カテゴリーおよび LC_UNI_NUMERIC カテゴリーの影響も受ける可能性があります。この関数は、コンパイル・コマンドに対して LOCALETYPE(*CLD) が指定されている場合には使用できません。詳細については、 549 ページの『CCSID およびロケールの理解』を参照してください。

ワイド文字関数: 詳細については、 553 ページの『ワイド文字』 を参照してください。

説明

`swprintf()` 関数は、一連のワイド文字と値をフォーマット設定し、ワイド文字バッファー *wcsbuffer* に保管します。 `swprintf()` 関数は、ワイド文字で機能する点を除いて、`sprintf()` 関数と同等です。

値 *n* は、終了ヌル文字を含む、書き込まれるワイド文字の最大数を指定します。`swprintf()` 関数は、format 内の対応するワイド文字フォーマット指定子に応じて引数リスト内の各項目を変換します。 format には、`printf()` 関数の書式ストリングと同じ書式および関数がありますが、以下の例外があります。

- %c (l 接頭部なし) は、mbtowc() 関数を呼び出して変換したかのように、文字引数を wchar_t に変換します。
- %lc および %C は wchar_t を wchar_t へコピーします。%#lc は %lc と同等で、 %#C は %C と同等です。
- %s (l 接頭部なし) は、mbstowcs() 関数を呼び出して変換したかのように、マルチバイト文字の配列を wchar_t の配列に変換します。この配列は、終了ヌル文字に達するまで書き込まれます (終了ヌル文字自身は書き込まれません) 。ただし、より短い出力が精度に指定されている場合は除きます。
- %ls および %S は wchar_t の配列をコピーします (変換は行われません)。この配列は、終了ヌル文字に達するまで書き込まれます (終了ヌル文字自身は書き込まれません)。ただし、より短い出力が精度に指定されている場合は除きます。 %#ls は %ls と同等で、%#S は %S と同等です。

幅および精度は常にワイド文字です。

ヌル・ワイド文字は、書き込まれるワイド文字の末尾に追加されます。ヌル・ワイド文字は、戻り値の一部として数えられません。オーバーラップしたオブジェクト間でコピーが行われる場合には、振る舞いは予期できません。

戻り値

swprintf() 関数は、出力バッファーに書き込まれるワイド文字の数を戻し、エラーが発生した場合に終了ヌル・ワイド文字または負の値をカウントしません。 n 文字以上のワイド文字の書き込みが要求された場合、負の値が戻されます。

errno の値は **EINVAL** (無効な引数) に設定される可能性があります。

swprintf() の使用例

この例では、swprintf() 関数を使用して、いくつかの値をフォーマット設定し、バッファーへ出力します。

```
#include <wchar.h>
#include <stdio.h>

#define BUF_SIZE 100

int main(void)
{
   wchar_t wcsbuf[BUF_SIZE];
   wchar_t wstring[] = L"ABCDE";
   int     num;

   num = swprintf(wcsbuf, BUF_SIZE, L"%s", "xyz");
   num += swprintf(wcsbuf + num, BUF_SIZE - num, L"%ls", wstring);
   num += swprintf(wcsbuf + num, BUF_SIZE - num, L"%i", 100);
   printf("The array wcsbuf contains: ¥"%ls¥"¥n", wcsbuf);
   return 0;

   /***************************************************************
      The output should be similar to :

      The array wcsbuf contains: "xyzABCDE100"
   ***************************************************************/
}
```

関連情報

- 238 ページの『printf() ― 定様式の文字の出力』

- 368 ページの『sprintf() ― フォーマット設定データのバッファーへの出力』
- 459 ページの『vswprintf() ― ワイド文字のフォーマット設定とバッファーへの書き込み』
- 20 ページの『<wchar.h>』

swscanf() ― ワイド文字データの読み取り

フォーマット

```
#include <wchar.h>
int swscanf(const wchar_t *buffer, const wchar_t *format, argument-list);
```

言語レベル: ANSI

スレッド・セーフ: はい。

ロケール依存: この関数の振る舞いは、現行ロケールの LC_CTYPE カテゴリーおよび LC_NUMERIC カテゴリーの影響を受ける可能性があります。また、LOCALETYPE(*LOCALEUCS2) または LOCALETYPE(*LOCALEUTF) がコンパイル・コマンドで指定されている場合、この振る舞いは、現行ロケールの LC_UNI_CTYPE カテゴリーおよび LC_UNI_NUMERIC カテゴリーの影響も受ける可能性があります。この関数は、コンパイル・コマンドに対して LOCALETYPE(*CLD) が指定されている場合には使用できません。詳細については、 549 ページの『CCSID およびロケールの理解』を参照してください。

ワイド文字関数: 詳細については、 553 ページの『ワイド文字』 を参照してください。

説明

`swscanf()` 関数は、`fwscanf()` 関数と同等ですが、引数バッファーが入力の取得元をストリームではなくワイド・ストリングに指定する点が異なります。ワイド・ストリングの末尾に達することは、 `fwscanf()` 関数のファイルの終わりを検出することに相当します。

戻り値

`swscanf()` 関数は、正常に変換され、割り当てられたフィールドの数を戻します。戻り値には、読み取りは行われたが、割り当てられなかったフィールドは含まれません。変換が行われる前に、ストリングの末尾が検出される場合、戻り値は EOF です。

`errno` の値は、**EINVAL** (無効な引数) に設定される可能性があります。

`swscanf()` の使用例

この例では、`swscanf()` 関数を使用して、ストリング *ltokenstring* からさまざまなデータを読み取り、そのデータを表示します。

```
#include <wchar.h>
#include <stdio.h>

wchar_t *ltokenstring = L"15 12 14";
int i;
float fp;
char s[10];
char c;

int main(void)
{
   /* Input various data                                          */

   swscanf(ltokenstring, L"%s %c%d%f", s, &c, &i, &fp);

   /* If there were no space between %s and %c,        */
   /* swscanf would read the first character following */
   /* the string, which is a blank space.                    */

   printf("string = %s¥n",s);
   printf("character = %c¥n",c);
   printf("integer = %d¥n",i);
   printf("floating-point number = %f¥n",fp);

}
```

関連情報

- 138 ページの『fscanf() ― フォーマット済みデータの読み取り』
- 344 ページの『scanf() ― データの読み取り』
- 153 ページの『fwscanf() ― ワイド文字を使用したストリームからのデータの読み取り』
- 528 ページの『wscanf() ― ワイド文字書式ストリングを使用したデータの読み取り』
- 371 ページの『sscanf() ― データの読み取り』
- 368 ページの『sprintf() ― フォーマット設定データのバッファーへの出力』
- 20 ページの『<wchar.h>』

system() ― コマンドの実行

フォーマット

```
#include <stdlib.h>
int system(const char *string);
```

言語レベル: ANSI

スレッド・セーフ: はい。ただし、CL コマンド・プロセッサーおよびすべての CL コマンドは、スレッド・セーフではありません。この関数は注意して使用してください。

説明

system() 関数は、指定の *string* を処理するために CL コマンド・プロセッサーに受け渡します。

戻り値

ストリングへの非ヌル・ポインターが受け渡された場合、system() 関数は CL コマンド・プロセッサーに引数を受け渡します。コマンドが正常に行われると、system() 関数はゼロを戻します。ストリングへの NULL ポインターが受け渡された場合、system() は -1 を戻し、コマンド・プロセッサーは呼び出されません。コマンドが失敗した場合、system() は 1 を戻します。system() 関数が失敗した場合、<stddef.h>

内のグローバル変数 _EXCP_MSGID が例外メッセージ ID で設定されます。_EXCP_MSGID 変数内に設定された例外メッセージ ID は、ジョブ CCSID にあります。

system() の使用例

```
#include <stdlib.h>

 int main(void)
 {
   int result;

   /* A data area is created, displayed and deleted:  */

   result = system("CRTDTAARA QTEMP/TEST TYPE(*CHAR) VALUE('Test')");
   result = system("DSPDTAARA TEST");
   result = system("DLTDTAARA TEST");

 }
```

関連情報

- 92 ページの『exit() ― プログラムの終了』
- 18 ページの『<stdlib.h>』

tan() ― 正接の計算

フォーマット

```
#include <math.h>
double tan(double x);
```

言語レベル: ANSI

スレッド・セーフ: はい。

説明

tan() 関数は *x* の正接を計算します。ここで、*x* はラジアンで表されます。*x* が大きすぎると、結果の有効数字が部分的に消失することがあり、errno が ERANGE に設定されます。errno の値も、 EDOM に設定される可能性もあります。

戻り値

tan() 関数は、*x* の正接の値を戻します。

tan() の使用例

この例では、*x* を $\pi/4$ の正接として計算します。

```
#include <math.h>
#include <stdio.h>

int main(void)
{
   double pi, x;

   pi = 3.1415926;
   x = tan(pi/4.0);

   printf("tan( %lf ) is %lf¥n", pi/4, x);
}

/******************  Output should be similar to:  ****************

tan( 0.785398 ) is 1.000000
*/
```

関連情報

- 40 ページの『acos() ― 逆余弦の計算』
- 45 ページの『asin() ― 逆正弦の計算』
- 47 ページの『atan() - atan2() ― 逆正接の計算』
- 68 ページの『cos() ― 余弦の計算』
- 69 ページの『cosh() ― 双曲線余弦の計算』
- 365 ページの『sin() ― 正弦の計算』
- 366 ページの『sinh() ― 双曲線正弦の計算』
- 『tanh() ― 双曲線正接の計算』
- 9 ページの『<math.h>』

tanh() ― 双曲線正接の計算

フォーマット

```
#include <math.h>
double tanh(double x);
```

言語レベル: ANSI

スレッド・セーフ: はい。

説明

`tanh()` 関数は x の双曲線正接を計算します。ここで、x はラジアンで表されます。

戻り値

`tanh()` 関数は x の双曲線正接の値を戻します。`tanh()` の結果は、範囲エラーを含むことができません。

`tanh()` の使用例

この例では、x を $\pi/4$ の双曲線正接として計算します。

```
#include <math.h>
#include <stdio.h>

int main(void)
{
   double pi, x;

   pi = 3.1415926;
   x = tanh(pi/4);

   printf("tanh( %lf ) = %lf¥n", pi/4, x);
}

/******************  Output should be similar to:  ****************

tanh( 0.785398 ) = 0.655794
*/
```

関連情報

- 40 ページの『acos() ― 逆余弦の計算』
- 45 ページの『asin() ― 逆正弦の計算』
- 47 ページの『atan() - atan2() ― 逆正接の計算』
- 68 ページの『cos() ― 余弦の計算』
- 69 ページの『cosh() ― 双曲線余弦の計算』
- 365 ページの『sin() ― 正弦の計算』
- 366 ページの『sinh() ― 双曲線正弦の計算』
- 427 ページの『tan() ― 正接の計算』
- 9 ページの『<math.h>』

time() ― 現在時刻の判別

フォーマット

```
#include <time.h>
time_t time(time_t *timeptr);
```

言語レベル: ANSI

スレッド・セーフ: はい。

説明

`time()` 関数は、現在のカレンダー時間を秒単位で判別します。

注: カレンダー時間とは、エポックである UTC (1970 年 1 月 1 日 00:00:00) から数えた秒数です。

戻り値

`time()` 関数は、現在のカレンダー時間を戻します。また、戻り値は *timeptr* によって提供される場所にも保管されます。*timeptr* が NULL の場合には、戻り値は保管されません。カレンダー時間が使用できない場合には、値 (time_t)(-1) が戻されます。

`time()` の使用例

この例では、時刻を取得して、それを *ltime* に割り当てます。その後 `ctime()` 関数は、秒数を現在の日時に変換します。この例では、その後で現在時刻を表示するメッセージを出力します。

```
#include <time.h>
#include <stdio.h>

int main(void)
{
   time_t ltime;
   if(time(&ltime) == -1)
{
   printf("Calendar time not available.¥n");
   exit(1);
}
   printf("The time is %s¥n", ctime(&ltime));
}

/******************  Output should be similar to:  ****************

The time is Mon Mar 22 19:01:41 2004
*/
```

関連情報

- 41 ページの『asctime() — 時間から文字ストリングへの変換』
- 43 ページの『asctime_r() — 時間から文字ストリングへの変換 (再始動可能)』
- 75 ページの『ctime() — 時間から文字ストリングへの変換』
- 77 ページの『ctime64() — 時間から文字ストリングへの変換』
- 80 ページの『ctime64_r() — 時間から文字ストリングへの変換 (再始動可能)』
- 78 ページの『ctime_r() — 時間から文字ストリングへの変換 (再始動可能)』
- 167 ページの『gmtime() — 時間の変換』
- 169 ページの『gmtime64() — 時間の変換』
- 173 ページの『gmtime64_r() — 時間の変換 (再始動可能)』
- 171 ページの『gmtime_r() — 時間の変換 (再始動可能)』
- 192 ページの『localtime() — 時間の変換』
- 194 ページの『localtime64() — 時間の変換』
- 197 ページの『localtime64_r() — 時間の変換 (再始動可能)』
- 195 ページの『localtime_r() — 時間の変換 (再始動可能)』
- 228 ページの『mktime() — 地方時の変換』
- 230 ページの『mktime64() — 地方時の変換』
- 『time64() — 現在時刻の判別』
- 19 ページの『<time.h>』

time64() — 現在時刻の判別

フォーマット

```
#include <time.h>
time64_t time64(time64_t *timeptr);
```

言語レベル: ILE C Extension

スレッド・セーフ: はい。

説明

`time64()` 関数は、現在のカレンダー時間を秒単位で判別します。

注: カレンダー時間とは、エポックである UTC (1970 年 1 月 1 日 00:00:00) から数えた秒数です。

戻り値

`time64()` 関数は、現在のカレンダー時間を戻します。また、戻り値は *timeptr* によって提供される場所にも保管されます。*timeptr* が NULL の場合には、戻り値は保管されません。カレンダー時間が使用できない場合には、値 (time_t)(-1) が戻されます。

`time64()` の使用例

この例では、時刻を取得して、それを *ltime* に割り当てます。その後 `ctime64()` 関数は、秒数を現在の日時に変換します。この例では、その後で現在時刻を表示するメッセージを出力します。

```
#include <time.h>
#include <stdio.h>

int main(void)
{
   time64_t ltime;

   if(time64(&ltime) == -1)
{
   printf("Calendar time not available.¥n");
   exit(1);
}
   printf("The time is %s", ctime64(&ltime));
}

/******************  Output should be similar to:  ****************

The time is Mon Mar 22 19:01:41 2004
*/
```

関連情報

- 41 ページの『asctime() — 時間から文字ストリングへの変換』
- 43 ページの『asctime_r() — 時間から文字ストリングへの変換 (再始動可能)』
- 75 ページの『ctime() — 時間から文字ストリングへの変換』
- 77 ページの『ctime64() — 時間から文字ストリングへの変換』
- 78 ページの『ctime_r() — 時間から文字ストリングへの変換 (再始動可能)』
- 75 ページの『ctime() — 時間から文字ストリングへの変換』
- 167 ページの『gmtime() — 時間の変換』
- 169 ページの『gmtime64() — 時間の変換』
- 173 ページの『gmtime64_r() — 時間の変換 (再始動可能)』
- 171 ページの『gmtime_r() — 時間の変換 (再始動可能)』
- 192 ページの『localtime() — 時間の変換』
- 194 ページの『localtime64() — 時間の変換』
- 197 ページの『localtime64_r() — 時間の変換 (再始動可能)』

- 195 ページの『localtime_r() ― 時間の変換 (再始動可能)』
- 228 ページの『mktime() ― 地方時の変換』
- 230 ページの『mktime64() ― 地方時の変換』
- 429 ページの『time() ― 現在時刻の判別』
- 19 ページの『<time.h>』

tmpfile() ― 一時ファイルの作成

フォーマット

```
#include <stdio.h>
FILE *tmpfile(void);
```

言語レベル: ANSI

スレッド・セーフ: はい。

説明

tmpfile() 関数は一時バイナリー・ファイルを作成します。このファイルは、クローズしたとき、またはプログラムが終了したとき自動的に削除されます。

tmpfile() 関数は、wb+ モードで一時ファイルをオープンします。

戻り値

正常に行われると、tmpfile() 関数はストリーム・ポインターを戻します。ファイルをオープンできない場合は、NULL ポインターを戻します。正常終了時 (exit()) に、これらの一時ファイルは除去されます。

i5/OS データ管理機能システムでは、tmpfile() 関数は QTEMP/QACXxxxx という名前の新規ファイルを作成します。コンパイル・コマンドで SYSIFCOPT(*IFSIO) オプションを指定すると、tmpfile() 関数は /tmp/QACXaaaaaaa という名前の新規ファイルを作成します。 tmpfile() 関数からのファイル名で作成されるファイルは、ジョブの終わりに廃棄されます。remove() 関数を使用して、ファイルを除去することができます。

tmpfile() の使用例

この例では一時ファイルを作成し、正常に作成できたら、そのファイルに tmpstring を書き込みます。このファイルはプログラムの最後に除去されます。

```
#include <stdio.h>

FILE *stream;
char tmpstring[ ] = "This is the string to be temporarily written";

int main(void)

{
   if((stream = tmpfile( )) == NULL)
      perror("Cannot make a temporary file");
   else
      fprintf(stream, "%s", tmpstring);
}
```

関連情報

- 114 ページの『fopen() ― ファイルのオープン』
- 17 ページの『<stdio.h>』

tmpnam() ― 一時ファイル名の作成

フォーマット

```
#include <stdio.h>
char *tmpnam(char *string);
```

言語レベル: ANSI

スレッド・セーフ: はい。ただし、tmpnam(NULL) の使用はスレッド・セーフではありません。

説明

`tmpnam()` 関数は、既存のファイルの名前と異なる有効なファイル名を作成します。これは、この名前を *string* に保管します。*string* が NULL の場合には、`tmpnam()` 関数は内部静的バッファーの結果を終了します。以降の呼び出しは、この値を破棄します。*string* が NULL ではないときには、これは少なくとも L_tmpnam バイトの配列を指している必要があります。L_tmpnam の値は <stdio.h> で定義されます。

`tmpnam()` 関数は、少なくとも最大 TMP_MAX 名の活動化グループ内で呼び出されるたびに、異なる名前を作成します。ILE C の場合、TMP_MAX は 32 767 です。これは理論上の限度です。同時にオープンできる実際のファイル数は、システムで使用可能なスペースによって異なります。

戻り値

`tmpnam()` 関数は、その名前へのポインターを戻します。固有の名前を作成できない場合は、NULL を戻します。

tmpnam() の使用例

この例では `tmpnam()` を呼び出して、有効なファイル名を作成します。

```
#include <stdio.h>

int main(void)
{
   char *name1;
   if ((name1 = tmpnam(NULL)) !=NULL)
      printf("%s can be used as a file name.¥n", name1);
   else printf("Cannot create a unique file name¥n");

}
```

関連情報

- 114 ページの『fopen() ― ファイルのオープン』
- 286 ページの『remove() ― ファイルの削除』
- 17 ページの『<stdio.h>』

toascii() ― 文字から ASCII で表現可能な文字への変換

フォーマット

```
#include <ctype.h>
int toascii(int c);
```

スレッド・セーフ: はい。

ロケール依存: この関数の振る舞いは、現行ロケールの LC_CTYPE カテゴリーの影響を受ける可能性があります。この関数は、コンパイル・コマンドに対して LOCALETYPE(*CLD) が指定されている場合には使用できません。詳細については、 549 ページの『CCSID およびロケールの理解』を参照してください。

説明

`toascii()` 関数は、文字 *c* が 7 ビット US-ASCII ロケールで何にマップされるかを判別し、現行ロケール内の対応する文字エンコードを戻します。

戻り値

`toascii()` 関数は、文字 *c* を 7 ビット US-ASCII ロケールに応じてマップし、現行ロケール内の対応する文字エンコードを戻します。

`toascii()` の使用例

この例は、`toascii()` によって 0x7c から 0x82 がマップされる 7 ビット US-ASCII 文字のエンコードを出力します。

```
#include <stdio.h>
#include <ctype.h>

 void main(void)
 {
   int ch;

   for (ch=0x7c; ch<=0x82; ch++) {
     printf("toascii(%#04x) = %c¥n", ch, toascii(ch));
   }

 }

/*****************And the output should be:********************************
toascii(0x7c) = @
toascii(0x7d) = '
toascii(0x7e) = =
toascii(0x7f) = "
toascii(0x80) = X
toascii(0x81) = a
toascii(0x82) = b
**********************************************************************/
```

関連情報

- 177 ページの『isascii() ― 表示可能文字の ASCII 値としてのテスト』
- 3 ページの『<ctype.h>』

tolower() - toupper() ― 英大/小文字の変換

フォーマット

```
#include <ctype.h>
int tolower(int C);
int toupper(int c);
```

言語レベル: ANSI

スレッド・セーフ: はい。

ロケール依存: これらの関数の振る舞いは、現行ロケールの LC_CTYPE カテゴリーの影響を受ける可能性があります。詳細については、 549 ページの『CCSID およびロケールの理解』を参照してください。

説明

`tolower()` 関数は、大文字 *C* を対応する英小文字に変換します。

`toupper()` 関数は、小文字 *c* を対応する英大文字に変換します。

戻り値

どちらの関数も、変換された文字を戻します。文字 *c* に対応する英小文字または英大文字がない場合、関数は *c* を変更しないで戻します。

`toupper()` および `tolower()` の使用例

この例では、`toupper()` 関数と `tolower()` 関数を使用して、コード 0 とコード 7f の間で文字を変更します。

```
#include <stdio.h>
#include <ctype.h>

int main(void)
{
   int ch;

   for (ch = 0; ch <= 0x7f; ch++)
   {
      printf("toupper=%#04x¥n", toupper(ch));
      printf("tolower=%#04x¥n", tolower(ch));
      putchar('¥n');
      }
}
```

関連情報

- 176 ページの『isalnum() - isxdigit() ― 整数値のテスト』
- 436 ページの『towlower() -towupper() ― ワイド文字の英大/小文字の変換』
- 3 ページの『<ctype.h>』

towctrans() ― ワイド文字の変換

フォーマット

```
#include <wctype.h>
wint_t towctrans(wint_t wc, wctrans_t desc);
```

言語レベル: ANSI

スレッド・セーフ: はい。

ロケール依存: コンパイル・コマンドで LOCALETYPE(*LOCALE) が指定される場合、この関数の振る舞いは、現行ロケールの LC_CTYPE カテゴリーの影響を受ける可能性があります。また、コンパイル・コマ

ンドで LOCALETYPE(*LOCALEUCS2) または LOCALETYPE(*LOCALEUTF) オプションのいずれかが指定される場合、現行ロケールの LC_UNI_CTYPE カテゴリーの影響も受ける可能性があります。この関数は、コンパイル・コマンドに対して LOCALETYPE(*CLD) が指定されている場合には使用できません。詳細については、 549 ページの『CCSID およびロケールの理解』を参照してください。

ワイド文字関数: 詳細については、 553 ページの『ワイド文字』 を参照してください。

説明

`towctrans()` 関数は、*desc* によって示されるマッピングを使用して、ワイド文字 *wc* をマップします。

`towctrans(wc, wctrans("tolower"))` はワイド文字に対して、文字ケースのマッピング関数である `towlower()` の呼び出しと同じように振る舞います。

`towctrans(wc, wctrans("toupper"))` はワイド文字に対して、文字ケースのマッピング関数である `towupper()` の呼び出しと同じように振る舞います。

戻り値

`towctrans()` 関数は、*desc* によって示されるマッピングを使用して、 *wc* のマップされた値を戻します。

`towctrans()` の使用例

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <wchar.h>
#include <wctype.h>

int main()
{
   char *alpha = "abcdefghijklmnopqrstuvwxyz";
   char *tocase[2] = {"toupper", "tolower"};
   wchar_t *wcalpha;
   int i, j;
   size_t alphalen;

   alphalen = strlen(alpha)+1;
   wcalpha = (wchar_t *)malloc(sizeof(wchar_t)*alphalen);

   mbstowcs(wcalpha, alpha, 2*alphalen);

   for (i=0; i<2; ++i) {
      printf("Input string: %ls¥n", wcalpha);
      for (j=0; j
      for (j=0; j
```

関連情報

- 516 ページの『wctrans() 一文字マッピングのハンドルの取得』
- 20 ページの『<wchar.h>』

towlower() -towupper() ― ワイド文字の英大/小文字の変換

フォーマット

```
#include <wctype.h>
wint_t towlower(wint_t wc);
wint_t towupper(wint_t wc);
```

言語レベル: ANSI

スレッド・セーフ: はい。

ロケール依存: コンパイル・コマンドで LOCALETYPE(*LOCALE) が指定される場合、これらの関数の振る舞いは、現行ロケールの LC_CTYPE カテゴリーの影響を受ける可能性があります。また、コンパイル・コマンドで LOCALETYPE(*LOCALEUCS2) または LOCALETYPE(*LOCALEUTF) オプションのいずれかが指定される場合、これらの関数の振る舞いは、現行ロケールの LC_UNI_CTYPE カテゴリーの影響も受ける可能性があります。これらの関数は、コンパイル・コマンドで LOCALETYPE(*CLD) が指定される場合には使用できません。詳細については、 549 ページの『CCSID およびロケールの理解』を参照してください。

ワイド文字関数: 詳細については、 553 ページの『ワイド文字』を参照してください。

説明

`towupper()` 関数は、小文字 *wc* を対応する英大文字に変換します。`towlower()` 関数は、大文字 *wc* を対応する英小文字に変換します。

戻り値

wc がワイド文字 (`iswupper()` (または `iswlower()`) が真) である場合に、対応するワイド文字 (`iswlower()` (または `iswupper()`) が真) が存在すると、`towlower()` (または `towupper()`) は対応するワイド文字を戻します。そうでない場合、引数は変更されずに戻されます。

`towlower()` および `towupper()` **の使用例**

この例では `towlower()` および `towupper()` を使用して、 0 と 0x7f の間で文字を変換します。

```
#include <wctype.h>
#include <stdio.h>

int main(void)
{
      wint_t w_ch;

      for (w_ch = 0; w_ch <= 0xff; w_ch++) {
          printf ("towupper : %#04x %#04x, ", w_ch, towupper(w_ch));
          printf ("towlower : %#04x %#04x¥n", w_ch, towlower(w_ch));
      }
      return 0;
/************************************************************************
The output should be similar to:

:
towupper : 0xc1 0xc1, towlower : 0xc1 0x81
towupper : 0xc2 0xc2, towlower : 0xc2 0x82
towupper : 0xc3 0xc3, towlower : 0xc3 0x83
towupper : 0xc4 0xc4, towlower : 0xc4 0x84
towupper : 0xc5 0xc5, towlower : 0xc5 0x85
:
towupper : 0x81 0xc1, towlower : 0x81 0x81
towupper : 0x82 0xc2, towlower : 0x82 0x82
towupper : 0x83 0xc3, towlower : 0x83 0x83
towupper : 0x84 0xc4, towlower : 0x84 0x84
towupper : 0x85 0xc5, towlower : 0x85 0x85
:
**********************************************************************/
}
```

関連情報

- 180 ページの『iswalnum() から iswxdigit() ― ワイド整数値のテスト』
- 434 ページの『tolower() - toupper() ― 英大/小文字の変換』
- 20 ページの『<wctype.h>』

_ultoa - 符号なし long 型整数からストリングへの変換

フォーマット

```
#include <stdlib.h>
char *_ultoa(unsigned long value, char *string, int radix);
```

注: `_ultoa` 関数は、C++ でのみサポートされており、 C ではサポートされていません。

言語レベル: Extension

スレッド・セーフ: はい。

説明

`_ultoa` は、指定された符号なし long 値 *value* の数字を、ヌル文字で終了する文字ストリングに変換し、その結果を *string* に保管します。*radix* 引数は、*value* の基数を指定します。2 から 36 の範囲でなければなりません。

注: *string* 用に割り振られたスペースは、戻されるストリングを保持するために十分な大きさでなければなりません。この関数は、ヌル文字 (¥0) を含めて最大 33 バイトまで戻すことができます。

戻り値

_ultoa は *string* へのポインターを戻します。エラーの戻り値はありません。

string 引数が NULL、または *radix* が 2 から 36 の範囲外にある場合、errno は EINVAL に設定されます。

_ultoa() の使用例

この例では、整数値 255 を 10 進数、2 進数、および 16 進数の表現に変換します。

```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
   char buffer[35];
   char *p;
   p = _ultoa(255UL, buffer, 10);
   printf("The result of _ultoa(255) with radix of 10 is %s\n", p);
   p = _ultoa(255UL, buffer, 2);
   printf("The result of _ultoa(255) with radix of 2\n    is %s\n", p);
   p = _ultoa(255UL, buffer, 16);
   printf("The result of _ultoa(255) with radix of 16 is %s\n", p);
   return 0;
}
```

The output should be:

```
      The result of _ultoa(255) with radix of 10 is 255
      The result of _ultoa(255) with radix of 2
          is 11111111
      The result of _ultoa(255) with radix of 16 is ff
```

関連情報

- 157 ページの『_gcvt - 浮動小数点からストリングへの変換』
- 183 ページの『_itoa - 整数からストリングへの変換』
- 200 ページの『_ltoa - long 型整数からストリングへの変換』
- 18 ページの『<stdlib.h>』

ungetc() — 入力ストリームへの文字のプッシュ

フォーマット

```
#include <stdio.h>
int ungetc(int c, FILE *stream);
```

言語レベル: ANSI

スレッド・セーフ: はい。

説明

ungetc() 関数は、符号なしの文字 *c* を指定された入力 *stream* にプッシュ・バックします。ただし、連続して ungetc() を呼び出す場合は、入力ストリームへのプッシュ・バックを保証できる連続した文字は 1 つのみです。 *stream* は、読み取りのためにオープンしていなければなりません。*stream* での以降の読み取り操作は、 *c* から開始します。文字 *c* を EOF 文字にすることはできません。

ungetc() でストリームに配置された文字は、*stream* から読み取られる前に、 fseek()、fsetpos()、rewind()、または fflush() が呼び出された場合は消去されます。

戻り値

ungetc() 関数は、*c* をプッシュ・バックできなかった場合に、 unsigned char、または EOF に変換された整数引数 *c* を戻します。

errno の値は、次のいずれかに設定されます。

値　　意味

ENOTREAD
　ファイルは読み取り操作用にオープンされていません。

EIOERROR
　リカバリー不能な入出力エラーが発生しました。

EIORECERR
　リカバリー可能な入出力エラーが発生しました。

ungetc() 関数は、type=record を指定してオープンしたファイルについてはサポートされません。

ungetc() の使用例

この例では、while ステートメントが算術ステートメントを使用して入力データ・ストリームから 10 進数を読み取り、読み取り時の数字の数値を構成します。ファイルの終わりまでに非数字が検出される場合、ungetc() は、後の入力関数が処理を実行できるように、入力ストリーム内でそれを置き換えます。

```
#include <stdio.h>
#include <ctype.h>

int main(void)
{
   FILE *stream;
   int ch;
   unsigned int result = 0;
   while ((ch = getc(stream)) != EOF && isdigit(ch))
      result = result * 10 + ch - '0';
   if (ch != EOF)
      ungetc(ch,stream);
         /* Put the nondigit character back */
   printf("The result is: %d¥n", result);
   if ((ch = getc(stream)) != EOF)
      printf("The character is: %c¥n", ch);
}
```

関連情報

- 158 ページの『getc() - getchar() ― 文字の読み取り』
- 101 ページの『fflush() ― ファイルへのバッファーの書き込み』
- 140 ページの『fseek() ― fseeko() ― ファイル位置の位置変更』
- 142 ページの『fsetpos() ― ファイル位置の設定』
- 249 ページの『putc() - putchar() ― 文字の書き込み』
- 288 ページの『rewind() ― 現在のファイル位置の調整』
- 17 ページの『<stdio.h>』

ungetwc() ― 入力ストリームへのワイド文字のプッシュ

フォーマット

```
#include <wchar.h>
#include <stdio.h>
wint_t ungetwc(wint_t wc, FILE *stream);
```

言語レベル: ANSI

スレッド・セーフ: はい。

ロケール依存: この関数の振る舞いは、現行ロケールの LC_CTYPE カテゴリーの影響を受ける可能性があります。また、この振る舞いは、LOCALETYPE(*LOCALEUCS2) または LOCALETYPE(*LOCALEUTF) がコンパイル・コマンドに対して指定されている場合は、現行ロケールの LC_UNI_CTYPE カテゴリーの影響を受ける可能性もあります。この関数は、コンパイル・コマンドに対して LOCALETYPE(*CLD) が指定されている場合には使用できません。詳細については、 549 ページの『CCSID およびロケールの理解』を参照してください。

統合ファイル・システム・インターフェース: この関数は、コンパイル・コマンドに対して SYSIFCOPT(*NOIFSIO) が指定されている場合には使用できません。

ワイド文字関数: 詳細については、 553 ページの『ワイド文字』を参照してください。

説明

ungetwc() 関数は、ワイド文字 *wc* を入力ストリームにプッシュ・バックします。プッシュ・バックされたワイド文字は、プッシュの逆順で以降の読み取りによってそのストリームに戻されます。ファイル位置決め関数 (fseek()、fsetpos()、または rewind()) に対する (ストリーム上の) 介入呼び出しが正常に行われると、ストリームのプッシュ・バックされたワイド文字が廃棄されます。ストリームに対応する外部ストレージは変更されません。プッシュ・バックされるワイド文字は、少なくとも常に 1 文字は存在します。*wc* の値が WEOF である場合には、操作は失敗し、入力ストリームは変更されません。

ungetwc() 関数に対する呼び出しが正常に行われると、ストリームの EOF 標識がクリアされます。すべてのプッシュ・バックされたワイド文字が読み込まれたか、廃棄された後のストリームのファイル位置標識の値は、ワイド文字がプッシュ・バックされる前のものと同じです。ただし、連続して ungetwc() を呼び出す場合に、入力ストリームへのプッシュ・バックを保証できる連続したワイド文字は 1 つのみです。

テキスト・ストリームの場合には、ファイル位置標識は 1 つのワイド文字によってバックアップされます。これは ftell()、fflush()、fseek() (SEEK_CUR と共に)、および fgetpos() 関数に影響を与えます。 バイナリー・ストリームの場合には、最後の文字がプッシュ・バックされない限り、すべての文字が読み取られるか、廃棄されるまで、位置標識は指定されません。この場合には、ファイル位置標識は 1 つのワイド文字によってバックアップされます。これは ftell()、fseek() (SEEK_CUR と共に)、fgetpos()、および fflush() 関数に影響を与えます。

戻り値

ungetwc() 関数は、変換後にプッシュ・バックされたワイド文字を戻します。操作が失敗した場合は、WEOF を戻します。

ungetwc() の使用例

```
#include <wchar.h>
#include <wctype.h>
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
```

```
   FILE         *stream;
   wint_t       wc;
   wint_t       wc2;
   unsigned int result = 0;

   if (NULL == (stream = fopen("ungetwc.dat", "r+"))) {
      printf("Unable to open file.¥n");
      exit(EXIT_FAILURE);
   }

   while (WEOF != (wc = fgetwc(stream)) && iswdigit(wc))
      result = result * 10 + wc - L'0';

   if (WEOF != wc)
      ungetwc(wc, stream);    /* Push the nondigit wide character back */

   /* get the pushed back character */
   if (WEOF != (wc2 = fgetwc(stream))) {
      if (wc != wc2) {
         printf("Subsequent fgetwc does not get the pushed back character.¥n");
         exit(EXIT_FAILURE);
      }
      printf("The digits read are '%i'¥n"
             "The character being pushed back is '%lc'", result, wc2);
   }
   return 0;

   /****************************************************************************
      Assuming the file ungetwc.dat contains:

      12345ABCDE67890XYZ

      The output should be similar to :

      The digits read are '12345'
      The character being pushed back is 'A'
   ****************************************************************************/
}
```

関連情報

- 101 ページの『fflush() ― ファイルへのバッファーの書き込み』
- 140 ページの『fseek() ― fseeko() ― ファイル位置の位置変更』
- 142 ページの『fsetpos() ― ファイル位置の設定』
- 163 ページの『getwc() ― ストリームからのワイド文字の読み取り』
- 252 ページの『putwc() ― ワイド文字の書き込み』
- 439 ページの『ungetc() ― 入力ストリームへの文字のプッシュ』
- 20 ページの『<wchar.h>』

va_arg() - va_end() - va_start() ― 関数引数のアクセス

フォーマット

```
#include <stdarg.h>
var_type va_arg(va_list arg_ptr, var_type);
void va_end(va_list arg_ptr);
void va_start(va_list arg_ptr, variable_name);
```

言語レベル: ANSI

スレッド・セーフ: はい。

説明

va_arg()、va_end()、および va_start() の各関数は、必須引数が定数でオプション引数が変数である場合に、この関数の引数にアクセスします。必須引数を通常のパラメーターとして、関数に対して宣言し、パラメーター名を介して引数にアクセスします。

va_start() は、後続の va_arg() と va_end() の呼び出しに対応する、*arg_ptr* ポインターを初期化します。

引数 *variable_name* は、(, ... の前にある) パラメーター・リストの右端の名前付きパラメーターの ID です。va_arg() の前に va_start() を使用します。対応する va_start() マクロと va_end() マクロは、同じ関数に入っていなければなりません。

va_arg() 関数は、*arg_ptr* によって指定されたロケーションから指定された *var_type* の値を検索し、*arg_ptr* を増やして、リストの次の引数を指します。va_arg() 関数は、関数内で任意の回数だけリストから引数を検索できます。 *var_type* 引数は、int、long、decimal、double、struct、union、または pointer のいずれか、またはこれらのいずれかの型の typedef でなければなりません。

va_end() 関数は、パラメーター走査の終了を指示するために必要です。

呼び出されたルーチンは引数の数を常に判別できるわけではないため、呼び出しルーチンが呼び出されたルーチンに引数の数を伝える必要があります。引数の数を判別するために、ルーチンは、NULL ポインターを使用してリストの最後をシグナル通知するか、オプションの引数のカウントを必須引数の 1 つとして受け渡すことができます。例えば、printf() 関数は、*format-string* 引数を介して引数の数を通知することができます。

戻り値

va_arg() 関数は、現行の引数を戻します。va_end 関数と va_start() 関数は値を戻しません。

va_arg() – va_end() – va_start() の使用例

この例では、引数の可変値を関数に受け渡し、各引数を配列に保管し、各引数を出力します。

```
#include <stdio.h>
#include <stdarg.h>

int vout(int max, ...);

int main(void)
{
   vout(3, "Sat", "Sun", "Mon");
   printf("¥n");
   vout(5, "Mon", "Tues", "Wed", "Thurs", "Fri");
}

int vout(int max, ...)
{
   va_list arg_ptr;
   int args = 0;
   char *days[7];

   va_start(arg_ptr, max);
   while(args < max)
   {
      days[args] = va_arg(arg_ptr, char *);
      printf("Day:  %s  ¥n", days[args++]);
      }
   va_end(arg_ptr);
}

/******************  Output should be similar to:  ****************

Day:  Sat
Day:  Sun
Day:  Mon

Day:  Mon
Day:  Tues
Day:  Wed
Day:  Thurs
Day:  Fri
*/
```

関連情報

- 『vfprintf() — ストリームへの引数データの出力』
- 452 ページの『vprintf() — 引数データの出力』
- 447 ページの『vfwprintf() — 引数データのワイド文字としてのフォーマット設定とストリームへの書き込み』
- 456 ページの『vsprintf() — 引数データのバッファーへの出力』
- 15 ページの『<stdarg.h>』

vfprintf() — ストリームへの引数データの出力

フォーマット

```
#include <stdarg.h>
#include <stdio.h>
int vfprintf(FILE *stream, const char *format, va_list arg_ptr);
```

言語レベル: ANSI

スレッド・セーフ: はい。

ロケール依存: この関数の振る舞いは、現行ロケールの LC_CTYPE カテゴリーおよび LC_NUMERIC カテゴリーの影響を受ける可能性があります。また、この振る舞いは、LOCALETYPE(*LOCALEUCS2) または LOCALETYPE(*LOCALEUTF) がコンパイル・コマンドに対して指定されている場合は、現行ロケールの LC_UNI_CTYPE カテゴリーの影響を受ける可能性もあります。この関数は、コンパイル・コマンドに対して LOCALETYPE(*CLD) が指定されている場合には使用できません。詳細については、549 ページの『CCSID およびロケールの理解』を参照してください。

説明

`vfprintf()` 関数は、一連の文字および値をフォーマット設定し、出力 *stream* に書き込みます。`vfprintf()` 関数は、 `fprintf()` 関数に似ていますが、*arg_ptr* は、プログラムの呼び出しによって個数が異なることがある引数のリストを指しているという点が異なります。これらの引数は、各呼び出しごとに va_start で初期化する必要があります。反対に、`fprintf()` 関数は引数のリストを持てますが、そのリストの引数の数はプログラムをコンパイルしたときに決定されます。

`vfprintf()` 関数は、*format* 内の対応するフォーマット指定子に応じて、引数リスト内の各項目を変換します。*format* には、 `printf()` 関数の書式ストリングと同じ書式および関数があります。

戻り値

正常に実行された場合、`vfprintf()` は *stream* に書き込まれたバイト数を戻します。エラーが発生した場合、関数は負の値を戻します。

`vfprintf()` の使用例

この例は、ストリングの可変値をファイル myfile に出力します。

```
#include <stdarg.h>
#include <stdio.h>

void vout(FILE *stream, char *fmt, ...);
char fmt1 [] = "%s  %s  %s¥n";

int main(void)
{
   FILE *stream;
   stream = fopen("mylib/myfile", "w");

   vout(stream, fmt1, "Sat", "Sun", "Mon");
}

void vout(FILE *stream, char *fmt, ...)

{
   va_list arg_ptr;

   va_start(arg_ptr, fmt);
   vfprintf(stream, fmt, arg_ptr);
   va_end(arg_ptr);
}

/******************  Output should be similar to:  ****************

Sat  Sun  Mon
*/
```

関連情報

- 122 ページの『fprintf() — フォーマット済みデータのストリームへの書き込み』

- 238 ページの『printf() ― 定様式の文字の出力』
- 442 ページの『va_arg() - va_end() - va_start() ― 関数引数のアクセス』
- 452 ページの『vprintf() ― 引数データの出力』
- 456 ページの『vsprintf() ― 引数データのバッファーへの出力』
- 463 ページの『vwprintf() ― 引数データのワイド文字としてのフォーマット設定と出力』
- 15 ページの『<stdarg.h>』
- 17 ページの『<stdio.h>』

vfscanf() ― フォーマット済みデータの読み取り

フォーマット

```
#include <stdarg.h>
#include <stdio.h>
int vfscanf(FILE *stream, const char *format, va_list arg_ptr);
```

言語レベル: ANSI

スレッド・セーフ: はい。

ロケール依存: この関数の振る舞いは、現行ロケールの LC_CTYPE カテゴリーおよび LC_NUMERIC カテゴリーの影響を受ける可能性があります。また、この振る舞いは、LOCALETYPE(*LOCALEUCS2) または LOCALETYPE(*LOCALEUTF) がコンパイル・コマンドに対して指定されている場合は、現行ロケールの LC_UNI_CTYPE カテゴリーの影響を受ける可能性もあります。この関数は、コンパイル・コマンドに対して LOCALETYPE(*CLD) が指定されている場合には使用できません。詳細については、 549 ページの『CCSID およびロケールの理解』を参照してください。

説明

vfscanf() 関数は、ストリームから引数の可変値で指定されたロケーションへデータを読み取ります。vfscanf() 関数は、fscanf() 関数と同様に機能しますが、*arg_ptr* はプログラムの呼び出しによって個数が異なることがある引数のリストを指しているという点が異なります。これらの引数は、各呼び出しごとに va_start で初期化する必要があります。反対に、fscanf() 関数は引数のリストを持てますが、そのリストの引数の数はプログラムをコンパイルしたときに決定されます。

各引数は、format-string の型指定子に対応する型を持つ変数へのポインターでなければなりません。*format* には、 scanf() 関数の書式ストリングと同じ書式および関数があります。

戻り値

vfscanf() 関数は、正常に変換され、割り当てられたフィールドの数を戻します。戻り値には、読み取りは行われたが、割り当てられなかったフィールドは含まれません。変換が行われていない場合に、ファイルの終わりを読み取ろうとすると、戻り値は EOF になります。戻り値 0 は、フィールドが割り当てられなかったことを意味します。

vfscanf() の使用例

この例では、入力のためにファイル *myfile* をオープンし、ストリング、long 型整数値、および浮動小数点値について、このファイルを走査します。

```
#include <stdio.h>
#include <stdarg.h>

int vread(FILE *stream, char *fmt, ...)
{
 int rc;
 va_list arg_ptr;
 va_start(arg_ptr, fmt);
 rc = vfscanf(stream, fmt, arg_ptr);
 va_end(arg_ptr);
 return(rc);
}

#define MAX_LEN 80
int main(void)
{
 FILE *stream;
 long l;
 float fp;
 char s[MAX_LEN + 1];
 char c;
 stream = fopen("mylib/myfile", "r");
 /* Put in various data. */
 vread(stream, "%s", &s[0]);
 vread(stream, "%ld", &l);
 vread(stream, "%c", &c);
 vread(stream, "%f", &fp);
 printf("string = %s¥n", s);
 printf("long double = %ld¥n", l);
 printf("char = %c¥n", c);
 printf("float = %f¥n", fp);
}
/*************** If myfile contains *************************
**************** abcdefghijklmnopqrstuvwxyz 343.2 ***********
********************** expected output is: *********************
string = abcdefghijklmnopqrstuvwxyz
long double = 343
char = .
float = 2.000000
*/
```

関連情報

- 122 ページの『fprintf() ― フォーマット済みデータのストリームへの書き込み』
- 138 ページの『fscanf() ― フォーマット済みデータの読み取り』
- 153 ページの『fwscanf() ― ワイド文字を使用したストリームからのデータの読み取り』
- 344 ページの『scanf() ― データの読み取り』
- 371 ページの『sscanf() ― データの読み取り』
- 425 ページの『swscanf() ― ワイド文字データの読み取り』
- 528 ページの『wscanf() ― ワイド文字書式ストリングを使用したデータの読み取り』
- 17 ページの『<stdio.h>』

vfwprintf() ― 引数データのワイド文字としてのフォーマット設定とストリームへの書き込み

フォーマット

```
#include <stdarg.h>
#include <stdio.h>
#include <wchar.h>
int vfwprintf(FILE *stream, const wchar_t *format, va_list arg);
```

言語レベル: ANSI

スレッド・セーフ: はい。

ロケール依存: この関数の振る舞いは、現行ロケールの LC_CTYPE カテゴリーおよび LC_NUMERIC カテゴリーの影響を受ける可能性があります。また、LOCALETYPE(*LOCALEUCS2) または LOCALETYPE(*LOCALEUTF) がコンパイル・コマンドで指定されている場合、この振る舞いは、現行ロケールの LC_UNI_CTYPE カテゴリーおよび LC_UNI_NUMERIC カテゴリーの影響も受ける可能性があります。この関数は、コンパイル・コマンドに対して LOCALETYPE(*CLD) が指定されている場合には使用できません。詳細については、 549 ページの『CCSID およびロケールの理解』を参照してください。

統合ファイル・システム・インターフェース: この関数は、コンパイル・コマンドに対して SYSIFCOPT(*NOIFSIO) が指定されている場合には使用できません。

ワイド文字関数: 詳細については、 553 ページの『ワイド文字』を参照してください。

説明

vfwprintf() 関数は、fwprintf() 関数と同等ですが、変数引数リストが、va_start マクロ (および、場合によっては後続の va_arg 呼び出し) によって初期化される *arg* によって置き換えられるという点が異なります。 vfwprintf() 関数は、va_end マクロを呼び出しません。

関数 vfwprintf()、vswprintf()、および vwprintf() は va_arg マクロを呼び出すため、戻り後の *arg* の値は指定されていません。

戻り値

vfwprintf() 関数は、出力バッファーに書き込まれるワイド文字の数を戻し、エラーが発生した場合に終了ヌル・ワイド文字または負の値をカウントしません。 n 文字以上のワイド文字の書き込みが要求された場合、負の値が戻されます。

vfwprintf() の使用例

この例では、ワイド文字 *a* をファイルに出力します。この出力は vout() 関数から行われ、引数の可変値を取り、 vfwprintf() を使用してそれらをファイルに出力します。

```
#include <wchar.h>
#include <stdarg.h>
#include <locale.h>

void vout (FILE *stream, wchar_t *fmt, ...);

const char ifs_path [] = "tmp/myfile";

 int main(void)  {

  FILE *stream;
  wchar_t format [] = L"%lc";

  setlocale(LC_ALL, "POSIX");
  if ((stream = fopen (ifs_path, "w")) == NULL) {
    printf("Could not open file.¥n");
    return (-1);
```

```
  }
  vout (stream, format, L'a');
  fclose (stream);

 /**********************************************
    The contents of output file tmp/myfile.dat should
    be a wide char 'a' which in the "POSIX" locale
    is '0081'x.
 */

  return (0);


 }

 void vout (FILE *stream, wchar_t *fmt, ...)
{
  va_list arg_ptr;
   va_start (arg_ptr, fmt);
   vfwprintf (stream, fmt, arg_ptr);
  va_end (arg_ptr);
}
```

関連情報

- 238 ページの『printf() ― 定様式の文字の出力』
- 122 ページの『fprintf() ― フォーマット済みデータのストリームへの書き込み』
- 444 ページの『vfprintf() ― ストリームへの引数データの出力』
- 452 ページの『vprintf() ― 引数データの出力』
- 56 ページの『btowc() ― 1 バイト文字のワイド文字への変換』
- 210 ページの『mbrtowc() ― マルチバイト文字からワイド文字への変換 (再始動可能)』
- 149 ページの『fwprintf() ― ワイド文字としてのデータのフォーマット設定とストリームへの書き込み』
- 459 ページの『vswprintf() ― ワイド文字のフォーマット設定とバッファーへの書き込み』
- 463 ページの『vwprintf() ― 引数データのワイド文字としてのフォーマット設定と出力』
- 15 ページの『<stdarg.h>』
- 17 ページの『<stdio.h>』
- 20 ページの『<wchar.h>』

vfwscanf() ― フォーマット済みワイド文字データの読み取り

フォーマット

```
#include <stdarg.h>
#include <stdio.h>
int vfwscanf(FILE *stream, const wchar_t *format, va_list arg_ptr);
```

言語レベル: ANSI

スレッド・セーフ: はい。

ロケール依存: この関数の振る舞いは、現行ロケールの LC_CTYPE カテゴリーおよび LC_NUMERIC カテゴリーの影響を受ける可能性があります。また、LOCALETYPE(*LOCALEUCS2) または LOCALETYPE(*LOCALEUTF) がコンパイル・コマンドで指定されている場合、この振る舞いは、現行ロケールの LC_UNI_CTYPE カテゴリーおよび LC_UNI_NUMERIC カテゴリーの影響も受ける可能性がありま

す。この関数は、コンパイル・コマンドに対して LOCALETYPE(*CLD) が指定されている場合には使用できません。詳細については、 549 ページの『CCSID およびロケールの理解』を参照してください。

統合ファイル・システム・インターフェース: 詳細については、 553 ページの『ワイド文字』を参照してください。

ワイド文字関数: この関数は、コンパイル・コマンドに対して SYSIFCOPT(*NOIFSIO) が指定されている場合には使用できません。

説明

`vfwscanf()` 関数は、ストリームから引数の可変値で指定されたロケーションへワイド・データを読み取ります。`vfwscanf()` 関数は、`fwscanf()` 関数と同様に機能しますが、*arg_ptr* はプログラムの呼び出しによって個数が異なることがある引数のリストを指しているという点が異なります。これらの引数は、各呼び出しごとに va_start で初期化する必要があります。反対に、 `fwscanf()` 関数は引数のリストを持てますが、そのリストの引数の数はプログラムをコンパイルしたときに決定されます。

各引数は、format-string の型指定子に対応する型を持つ変数へのポインターでなければなりません。*format* には、 `fwscanf()` 関数の書式ストリングと同じ書式および関数があります。

戻り値

`vfwscanf()` 関数は、正常に変換され、割り当てられたフィールドの数を戻します。戻り値には、読み取りは行われたが、割り当てられなかったフィールドは含まれません。変換が行われていない場合に、ファイルの終わりを読み取ろうとすると、戻り値は EOF になります。戻り値 0 は、フィールドが割り当てられなかったことを意味します。

`vfwscanf` の使用例

この例では、入力のためにファイル *myfile* をオープンし、ストリング、long 型整数値、および浮動小数点値について、このファイルを走査します。

```
#include <stdio.h>
#include <stdarg.h>
#include <wchar.h>

int vread(FILE *stream, wchar_t *fmt, ...)
{
 int rc;
 va_list arg_ptr;
 va_start(arg_ptr, fmt);
 rc = vfwscanf(stream, fmt, arg_ptr);
 va_end(arg_ptr);
 return(rc);
}

#define MAX_LEN 80
int main(void)
{
 FILE *stream;
 long l;
 float fp;
 char s[MAX_LEN + 1];
 char c;
 stream = fopen("mylib/myfile", "r");
 /* Put in various data. */
 vread(stream, L"%s", &s [0]);
 vread(stream, L"%ld", &l);
 vread(stream, L"%c", &c);
 vread(stream, L"%f", &fp);
 printf("string = %s¥n", s);
 printf("long double = %ld¥n", l);
 printf("char = %c¥n", c);
 printf("float = %f¥n", fp);
}
/*************** If myfile contains ************************
**************** abcdefghijklmnopqrstuvwxyz 343.2 ***********
********************** expected output is: *********************
string = abcdefghijklmnopqrstuvwxyz
long double = 343
char = .
float = 2.000000
*/
```

関連情報

- 138 ページの『fscanf() ― フォーマット済みデータの読み取り』
- 149 ページの『fwprintf() ― ワイド文字としてのデータのフォーマット設定とストリームへの書き込み』
- 153 ページの『fwscanf() ― ワイド文字を使用したストリームからのデータの読み取り』
- 344 ページの『scanf() ― データの読み取り』
- 371 ページの『sscanf() ― データの読み取り』
- 423 ページの『swprintf() ― ワイド文字のフォーマット設定とバッファーへの書き込み』
- 425 ページの『swscanf() ― ワイド文字データの読み取り』
- 446 ページの『vfscanf() ― フォーマット済みデータの読み取り』
- 449 ページの『vfwscanf() ― フォーマット済みワイド文字データの読み取り』
- 453 ページの『vscanf() ― フォーマット済みデータの読み取り』
- 458 ページの『vsscanf() ― フォーマット済みデータの読み取り』
- 461 ページの『vswscanf() ― フォーマット済みワイド文字データの読み取り』
- 465 ページの『vwscanf() ― フォーマット済みワイド文字データの読み取り』

- 527 ページの『wprintf() ― データのワイド文字としてのフォーマット設定と出力』
- 528 ページの『wscanf() ― ワイド文字書式ストリングを使用したデータの読み取り』
- 20 ページの『<wchar.h>』

vprintf() ― 引数データの出力

フォーマット

```
#include <stdarg.h>
#include <stdio.h>
int vprintf(const char *format, va_list arg_ptr);
```

言語レベル: ANSI

スレッド・セーフ: はい。

ロケール依存: この関数の振る舞いは、現行ロケールの LC_CTYPE カテゴリーおよび LC_NUMERIC カテゴリーの影響を受ける可能性があります。また、この振る舞いは、LOCALETYPE(*LOCALEUCS2) または LOCALETYPE(*LOCALEUTF) がコンパイル・コマンドに対して指定されている場合は、現行ロケールの LC_UNI_CTYPE カテゴリーの影響を受ける可能性もあります。詳細については、 549 ページの『CCSID およびロケールの理解』を参照してください。

説明

`vprintf()` 関数は、一連の文字および値をフォーマット設定し、 `stdout` に出力します。`vprintf()` 関数は、`printf()` 関数と同様に機能しますが、*arg_ptr* はプログラムの呼び出しによって個数が異なることがある引数のリストを指しているという点が異なります。これらの引数は、各呼び出しごとに va_start で初期化する必要があります。反対に、`printf()` 関数は引数のリストを持てますが、そのリストの引数の数はプログラムをコンパイルしたときに決定されます。

`vprintf()` 関数は、*format* 内の対応するフォーマット指定子に応じて、引数リスト内の各項目を変換します。*format* には、 `printf()` 関数の書式ストリングと同じ書式および関数があります。

戻り値

正常に実行された場合、`vprintf()` 関数は `stdout` に書き込まれたバイト数を戻します。エラーが発生した場合、`vprintf()` 関数は負の値を戻します。errno の値は、ETRUNC に設定することができます。

`vprintf()` の使用例

この例では、ストリングの可変値を `stdout` に出力します。

```
#include <stdarg.h>
#include <stdio.h>

void vout(char *fmt, ...);
char fmt1 [] = "%s  %s  %s   %s   %s ¥n";

int main(void)
{
   FILE *stream;
   stream = fopen("mylib/myfile", "w");

   vout(fmt1, "Mon", "Tues", "Wed", "Thurs", "Fri");
}

void vout(char *fmt, ...)
{
   va_list arg_ptr;

   va_start(arg_ptr, fmt);
   vprintf(fmt, arg_ptr);
   va_end(arg_ptr);
}

/******************  Output should be similar to:  ****************

Mon  Tues  Wed   Thurs   Fri
*/
```

関連情報

- 238 ページの『printf() ― 定様式の文字の出力』
- 442 ページの『va_arg() - va_end() - va_start() ― 関数引数のアクセス』
- 444 ページの『vfprintf() ― ストリームへの引数データの出力』
- 456 ページの『vsprintf() ― 引数データのバッファーへの出力』
- 15 ページの『<stdarg.h>』
- 17 ページの『<stdio.h>』

vscanf() ― フォーマット済みデータの読み取り

フォーマット

```
#include <stdarg.h>
#include <stdio.h>
int vscanf(const char *format, va_list arg_ptr);
```

言語レベル: ANSI

スレッド・セーフ: はい。

ロケール依存: この関数の振る舞いは、現行ロケールの LC_CTYPE カテゴリーおよび LC_NUMERIC カテゴリーの影響を受ける可能性があります。また、この振る舞いは、LOCALETYPE(*LOCALEUCS2) または LOCALETYPE(*LOCALEUTF) がコンパイル・コマンドに対して指定されている場合は、現行ロケールの LC_UNI_CTYPE カテゴリーの影響を受ける可能性もあります。詳細については、 549 ページの『CCSID およびロケールの理解』を参照してください。

説明

vscanf() 関数は、stdin から引数の可変値で指定されたロケーションへデータを読み取ります。vscanf 関数は、scanf() 関数と同様に機能しますが、*arg_ptr* はプログラムの呼び出しによって個数が異なることがある引数のリストを指しているという点が異なります。これらの引数は、各呼び出しごとに va_start で初期化する必要があります。反対に、scanf() 関数は引数のリストを持てますが、そのリストの引数の数はプログラムをコンパイルしたときに決定されます。

各引数は、format-string の型指定子に対応する型を持つ変数へのポインターでなければなりません。*format* には、 scanf() 関数の書式ストリングと同じ書式および関数があります。

戻り値

vscanf() 関数は、正常に変換され、割り当てられたフィールドの数を戻します。戻り値には、読み取りは行われたが、割り当てられなかったフィールドは含まれません。変換が行われていない場合に、ファイルの終わりを読み取ろうとすると、戻り値は EOF になります。戻り値 0 は、フィールドが割り当てられなかったことを意味します。

vscanf() の使用例

この例では、vscanf() 関数を使用して、stdin から整数、浮動小数点値、文字、およびストリングを読み取り、それらの値を表示します。

```
#include <stdio.h>
#include <stdarg.h>
int vread(char *fmt, ...)
{
 int rc;
 va_list arg_ptr;
 va_start(arg_ptr, fmt);
 rc = vscanf(fmt, arg_ptr);
 va_end(arg_ptr);
 return(rc);
}

int main(void)
{
 int i, rc;
 float fp;
 char c, s[81];
 printf("Enter an integer, a real number, a character "
 "and a string : ¥n");
 rc = vread("%d %f %c %s", &i, &fp, &c, s);
 if (rc != 4)
  printf("Not all fields are assigned¥n");
 else
 {
  printf("integer = %d¥n", i);
  printf("real number = %f¥n", fp);
  printf("character = %c¥n", c);
  printf("string = %s¥n",s);
 }
}
/***************** If input is: 12 2.5 a yes, *******************
************** then output should be similar to: ****************
Enter an integer, a real number, a character and a string :
integer = 12
real number = 2.500000
character = a
string = yes
*/
```

関連情報

- 138 ページの『fscanf() ― フォーマット済みデータの読み取り』
- 149 ページの『fwprintf() ― ワイド文字としてのデータのフォーマット設定とストリームへの書き込み』
- 153 ページの『fwscanf() ― ワイド文字を使用したストリームからのデータの読み取り』
- 344 ページの『scanf() ― データの読み取り』
- 371 ページの『sscanf() ― データの読み取り』
- 423 ページの『swprintf() ― ワイド文字のフォーマット設定とバッファーへの書き込み』
- 425 ページの『swscanf() ― ワイド文字データの読み取り』
- 446 ページの『vfscanf() ― フォーマット済みデータの読み取り』
- 449 ページの『vfwscanf() ― フォーマット済みワイド文字データの読み取り』
- 453 ページの『vscanf() ― フォーマット済みデータの読み取り』
- 458 ページの『vsscanf() ― フォーマット済みデータの読み取り』
- 461 ページの『vswscanf() ― フォーマット済みワイド文字データの読み取り』
- 465 ページの『vwscanf() ― フォーマット済みワイド文字データの読み取り』
- 527 ページの『wprintf() ― データのワイド文字としてのフォーマット設定と出力』
- 528 ページの『wscanf() ― ワイド文字書式ストリングを使用したデータの読み取り』
- 20 ページの『<wchar.h>』

vsnprintf() ― 引数データのバッファーへの出力

フォーマット

```
#include <stdarg.h>
#include <stdio.h>
int vsnprintf(char *target-string, size_t n, const char *format, va_list arg_ptr);
```

言語レベル: ANSI

スレッド・セーフ: はい。

ロケール依存: この関数の振る舞いは、現行ロケールの LC_CTYPE カテゴリーおよび LC_NUMERIC カテゴリーの影響を受ける可能性があります。また、この振る舞いは、LOCALETYPE(*LOCALEUCS2) または LOCALETYPE(*LOCALEUTF) がコンパイル・コマンドに対して指定されている場合は、現行ロケールの LC_UNI_CTYPE カテゴリーの影響を受ける可能性もあります。詳細については、 549 ページの『CCSID およびロケールの理解』を参照してください。

説明

vsnprintf() 関数は、一連の文字と値をフォーマット設定し、バッファー target-string に保管します。vsnprintf() 関数は、 snprintf() 関数に似ていますが、*arg_ptr* はプログラムの呼び出しによって個数が異なることがある引数のリストを指しているという点が異なります。これらの引数は、各呼び出しごとに va_start 関数で初期化する必要があります。反対に、snprintf() 関数は引数のリストを持てますが、そのリストの引数の数はプログラムをコンパイルしたときに決定されます。

vsnprintf() 関数は、*format* 内の対応するフォーマット指定子に応じて、引数リスト内の各項目を変換します。*format* には、 printf() 関数の書式ストリングと同じ書式および関数があります。

戻り値

vsnprintf() 関数は、配列に書き込まれるバイト数を戻し、終了ヌル文字はカウントしません。

vsnprintf() の使用例

この例では、可変数のストリングを *string* に割り当て、結果のストリングを出力します。

```
#include <stdarg.h>
#include <stdio.h>

void vout(char *string, char *fmt, ...);
char fmt1 [] = "%s  %s  %s¥n";

int main(void)
{
   char string[100];

   vout(string, fmt1, "Sat", "Sun", "Mon");
   printf("The string is:  %s¥n", string);
}

void vout(char *string, char *fmt, ...)
{
   va_list arg_ptr;

   va_start(arg_ptr, fmt);
   vsnprintf(string, 8, fmt, arg_ptr);
   va_end(arg_ptr);
}

/******************  Output should be similar to: ****************

The string is:  Sat  Su
*/
```

関連情報

- 238 ページの『printf() ― 定様式の文字の出力』
- 368 ページの『sprintf() ― フォーマット設定データのバッファーへの出力』
- 367 ページの『snprintf() ― フォーマット設定データのバッファーへの出力』
- 442 ページの『va_arg() - va_end() - va_start() ― 関数引数のアクセス』
- 444 ページの『vfprintf() ― ストリームへの引数データの出力』
- 『vsprintf() ― 引数データのバッファーへの出力』
- 15 ページの『<stdarg.h>』
- 17 ページの『<stdio.h>』

vsprintf() ― 引数データのバッファーへの出力

フォーマット

```
#include <stdarg.h>
#include <stdio.h>
int vsprintf(char *target-string, const char *format, va_list arg_ptr);
```

言語レベル: ANSI

スレッド・セーフ: はい。

ロケール依存: この関数の振る舞いは、現行ロケールの LC_CTYPE カテゴリーおよび LC_NUMERIC カテゴリーの影響を受ける可能性があります。また、この振る舞いは、LOCALETYPE(*LOCALEUCS2) また

は LOCALETYPE(*LOCALEUTF) がコンパイル・コマンドに対して指定されている場合は、現行ロケールの LC_UNI_CTYPE カテゴリーの影響を受ける可能性もあります。詳細については、 549 ページの『CCSID およびロケールの理解』を参照してください。

説明

`vsprintf()` 関数は、一連の文字および値をフォーマット設定し、バッファー *target-string* に保管します。`vsprintf()` 関数は、`sprintf()` 関数と同様に機能しますが、*arg_ptr* はプログラムの呼び出しによって個数が異なることがある引数のリストを指しているという点が異なります。これらの引数は、各呼び出しごとに va_start 関数で初期化する必要があります。反対に、`sprintf()` 関数は引数のリストを持てますが、そのリストの引数の数はプログラムをコンパイルしたときに決定されます。

`vsprintf()` 関数は、*format* 内の対応するフォーマット指定子に応じて、引数リスト内の各項目を変換します。*format* には、 `printf()` 関数の書式ストリングと同じ書式および関数があります。

戻り値

正常に実行された場合、`vsprintf()` 関数は *target-string* に書き込まれたバイト数を戻します。エラーが発生した場合、`vsprintf()` 関数は負の値を戻します。

`vsprintf()` の使用例

この例では、ストリングの可変値を string に割り当て、結果のストリングを出力します。

```
#include <stdarg.h>
#include <stdio.h>

void vout(char *string, char *fmt, ...);
char fmt1 [] = "%s  %s  %s¥n";

int main(void)
{
   char string[100];

   vout(string, fmt1, "Sat", "Sun", "Mon");
   printf("The string is:  %s¥n", string);
}

void vout(char *string, char *fmt, ...)
{
   va_list arg_ptr;

   va_start(arg_ptr, fmt);
   vsprintf(string, fmt, arg_ptr);
   va_end(arg_ptr);
}

/******************  Output should be similar to: ****************

The string is:  Sat  Sun  Mon
*/
```

関連情報

- 238 ページの『printf() ― 定様式の文字の出力』
- 368 ページの『sprintf() ― フォーマット設定データのバッファーへの出力』
- 442 ページの『va_arg() - va_end() - va_start() ― 関数引数のアクセス』
- 444 ページの『vfprintf() ― ストリームへの引数データの出力』

- 452 ページの『vprintf() ― 引数データの出力』
- 459 ページの『vswprintf() ― ワイド文字のフォーマット設定とバッファーへの書き込み』
- 15 ページの『<stdarg.h>』
- 17 ページの『<stdio.h>』

vsscanf() ― フォーマット済みデータの読み取り

フォーマット

```
#include <stdarg.h>
#include <stdio.h>
int vsscanf(const char *buffer, const char *format, va_list arg_ptr);
```

言語レベル: ANSI

スレッド・セーフ: はい。

ロケール依存: この関数の振る舞いは、現行ロケールの LC_CTYPE カテゴリーおよび LC_NUMERIC カテゴリーの影響を受ける可能性があります。また、この振る舞いは、LOCALETYPE(*LOCALEUCS2) または LOCALETYPE(*LOCALEUTF) がコンパイル・コマンドに対して指定されている場合は、現行ロケールの LC_UNI_CTYPE カテゴリーの影響を受ける可能性もあります。詳細については、 549 ページの『CCSID およびロケールの理解』を参照してください。

説明

vsscanf() 関数は、バッファーから引数の可変値で指定されたロケーションへデータを読み取ります。vsscanf() 関数は、sscanf() 関数と同様に機能しますが、*arg_ptr* はプログラムの呼び出しによって個数が異なることがある引数のリストを指しているという点が異なります。これらの引数は、各呼び出しごとに va_start で初期化する必要があります。反対に、sscanf() 関数は引数のリストを持てますが、そのリストの引数の数はプログラムをコンパイルしたときに決定されます。

各引数は、format-string の型指定子に対応する型を持つ変数へのポインターでなければなりません。*format* には、 scanf() 関数の書式ストリングと同じ書式および関数があります。

戻り値

vsscanf() 関数は、正常に変換され、割り当てられたフィールドの数を戻します。戻り値には、読み取りは行われたが、割り当てられなかったフィールドは含まれません。変換が行われていない場合に、ファイルの終わりを読み取ろうとすると、戻り値は EOF になります。戻り値 0 は、フィールドが割り当てられなかったことを意味します。

vsscanf() の使用例

この例では vsscanf() を使用して、ストリング *tokenstring* からさまざまなデータを読み取り、そのデータを表示します。

```
#include <stdio.h>
#include <stdarg.h>
#include <stddef.h>

int vread(const char *buffer, char *fmt, ...)
{
 int rc;
 va_list arg_ptr;
 va_start(arg_ptr, fmt);
 rc = vsscanf(buffer, fmt, arg_ptr);
 va_end(arg_ptr);
 return(rc);
}
int main(void)
{
 char *tokenstring = "15 12 14";
 wchar_t * idestring = L"ABC Z";
 wchar_t ws[81];
 wchar_t wc;
 int i;
 float fp;
 char s[81];
 char c;
 /* Input various data */
 /* In the first invocation of vsscanf, the format string is */
 /* "%s %c%d%f". If there were no space between %s and %c, */
 /* vsscanf would read the first character following the */
 /* string, which is a blank space. */
 vread(tokenstring, "%s %c%d%f", s, &c, &i, &fp);
 vread((char *) idestring, "%S %C", ws,&wc);
 /* Display the data */
 printf("¥nstring = %s¥n",s);
 printf("character = %c¥n",c);
 printf("integer = %d¥n",i);
 printf("floating-point number = %f¥n",fp);
 printf("wide-character string = %S¥n", ws);
 printf("wide-character = %C¥n", wc);
}
/***************** Output should be similar to: *****************
string = 15
character = 1
integer = 2
floating-point number = 14.000000
wide-character string = ABC
wide-character = Z
*******************************************************************/
```

関連情報

- 138 ページの『fscanf() ― フォーマット済みデータの読み取り』
- 153 ページの『fwscanf() ― ワイド文字を使用したストリームからのデータの読み取り』
- 344 ページの『scanf() ― データの読み取り』
- 371 ページの『sscanf() ― データの読み取り』
- 368 ページの『sprintf() ― フォーマット設定データのバッファーへの出力』
- 17 ページの『<stdio.h>』
- 425 ページの『swscanf() ― ワイド文字データの読み取り』
- 528 ページの『wscanf() ― ワイド文字書式ストリングを使用したデータの読み取り』

vswprintf() ― ワイド文字のフォーマット設定とバッファーへの書き込み

フォーマット

```
#include <stdarg.h>
#include <wchar.h>
int vswprintf(wchar_t *wcsbuffer, size_t n, const wchar_t
              *format, va_list argptr);
```

言語レベル: ANSI

スレッド・セーフ: はい。

ロケール依存: この関数の振る舞いは、現行ロケールの LC_CTYPE カテゴリーおよび LC_NUMERIC カテゴリーの影響を受ける可能性があります。また、LOCALETYPE(*LOCALEUCS2) または LOCALETYPE(*LOCALEUTF) がコンパイル・コマンドに対して指定されている場合は、現行ロケールの LC_UNI_CTYPE カテゴリーおよび LC_UNI_NUMERIC カテゴリーにも影響を受ける可能性があります。この関数は、コンパイル・コマンドに対して LOCALETYPE(*CLD) が指定されている場合には使用できません。詳細については、 549 ページの『CCSID およびロケールの理解』を参照してください。

ワイド文字関数: 詳細については、 553 ページの『ワイド文字』を参照してください。

説明

`vswprintf()` 関数は、一連のワイド文字と値をフォーマット設定し、バッファー *wcsbuffer* に保管します。`vswprintf()` 関数は、 `swprintf()` 関数に似ていますが、*argptr* は番号が呼び出しによって異なることがあるワイド文字引数のリストを指しているという点が異なります。これらの引数は、各呼び出しごとに va_start で初期化する必要があります。これとは対照的に、`swprintf()` 関数は引数のリストを持てますが、そのリストの引数の数はプログラムをコンパイルしたときに決定されます。

値 *n* は、終了ヌル文字を含む、書き込まれるワイド文字の最大数を指定します。`vswprintf()` 関数は、format 内の対応するワイド文字フォーマット指定子に応じて引数リスト内の各項目を変換します。 format には、`printf()` 関数の書式ストリングと同じ書式および関数がありますが、以下の例外があります。

- %c (l 接頭部なし) は、`mbtowc()` 関数を呼び出して変換したかのように、整数引数を wchar_t に変換します。
- %lc は wint_t を wchar_t に変換します。
- %s (l 接頭部なし) は、`mbrtowc()` 関数を呼び出して変換したかのように、マルチバイト文字の配列を wchar_t の配列に変換します。この配列は、終了ヌル文字に達するまで書き込まれます (終了ヌル文字自身は書き込まれません) 。ただし、より短い出力が精度に指定されている場合は除きます。
- %ls は wchar_t の配列を書き込みます。この配列は、終了ヌル文字に達するまで書き込まれます (終了ヌル文字自身は書き込まれません) 。ただし、より短い出力が精度に指定されている場合は除きます。

ヌル・ワイド文字は、書き込まれるワイド文字の末尾に追加されます。ヌル・ワイド文字は、戻り値の一部として数えられません。オーバーラップしたオブジェクト間でコピーが行われる場合には、振る舞いは予期できません。

戻り値

`vswprintf()` 関数は、配列に書き込まれるバイト数を戻し、終了ヌル・ワイド文字はカウントしません。

`vswprintf()` の使用例

この例では、ワイド文字引数の可変値を取り、 `vswprintf()` を使用してそれらを *wcstr* に出力する関数 `vout()` を作成します。

```
#include <stdio.h>
#include <stdarg.h>
#include <wchar.h>

wchar_t *format3 = L"%ls  %d  %ls";
wchar_t *format5 = L"%ls  %d  %ls  %d  %ls";

void vout(wchar_t *wcs, size_t n, wchar_t *fmt, ...)
{
   va_list arg_ptr;

   va_start(arg_ptr, fmt);
   vswprintf(wcs, n, fmt, arg_ptr);
   va_end(arg_ptr);
   return;
}

int main(void)
{
   wchar_t wcstr[100];

   vout(wcstr, 100, format3, L"ONE", 2L, L"THREE");
   printf("%ls¥n", wcstr);
   vout(wcstr, 100, format5, L"ONE", 2L, L"THREE", 4L, L"FIVE");
   printf("%ls¥n", wcstr);
   return 0;

   /************************************************************
      The output should be similar to:

      ONE  2  THREE
      ONE  2  THREE  4  FIVE
   ************************************************************/
}
```

関連情報

- 423 ページの『swprintf() ― ワイド文字のフォーマット設定とバッファーへの書き込み』
- 444 ページの『vfprintf() ― ストリームへの引数データの出力』
- 452 ページの『vprintf() ― 引数データの出力』
- 456 ページの『vsprintf() ― 引数データのバッファーへの出力』
- 15 ページの『<stdarg.h>』
- 20 ページの『<wchar.h>』

vswscanf() ― フォーマット済みワイド文字データの読み取り

フォーマット

```
#include <stdarg.h>
#include <wchar.h>

int vswscanf(const wchar_t *buffer, const wchar_t *format, va_list arg_ptr);
```

言語レベル: ANSI

スレッド・セーフ: はい。

ロケール依存: この関数の振る舞いは、現行ロケールの LC_CTYPE カテゴリーおよび LC_NUMERIC カテゴリーの影響を受ける可能性があります。また、LOCALETYPE(*LOCALEUCS2) または LOCALETYPE(*LOCALEUTF) がコンパイル・コマンドに対して指定されている場合は、現行ロケールの

LC_UNI_CTYPE カテゴリーおよび LC_UNI_NUMERIC カテゴリーにも影響を受ける可能性があります。この関数は、コンパイル・コマンドに対して LOCALETYPE(*CLD) が指定されている場合には使用できません。詳細については、 549 ページの『CCSID およびロケールの理解』を参照してください。

ワイド文字関数: 詳細については、 553 ページの『ワイド文字』を参照してください。

説明

`vswscanf()` 関数は、バッファーから引数の引数番号で指定されたロケーションへワイド・データを読み取ります。`vswscanf()` 関数は、`swscanf()` 関数と同様に機能しますが、*arg_ptr* はプログラムの呼び出しによって個数が異なることがある引数のリストを指しているという点が異なります。これらの引数は、各呼び出しごとに va_start で初期化する必要があります。反対に、 `swscanf()` 関数は引数のリストを持てますが、そのリストの引数の数はプログラムをコンパイルしたときに決定されます。

各引数は、format-string の型指定子に対応する型を持つ変数へのポインターでなければなりません。*format* には、 `swscanf()` 関数の書式ストリングと同じ書式および関数があります。

戻り値

`vswscanf()` 関数は、正常に変換され、割り当てられたフィールドの数を戻します。戻り値には、読み取りは行われたが、割り当てられなかったフィールドは含まれません。変換が行われていない場合に、ファイルの終わりを読み取ろうとすると、戻り値は EOF になります。戻り値 0 は、フィールドが割り当てられなかったことを意味します。

`vswscanf()` の使用例

この例では `vswscanf()` 関数を使用して、ストリング *tokenstring* からさまざまなデータを読み取り、そのデータを表示します。

```
#include <stdio.h>
#include <stdarg.h>
#include <wchar.h>
int vread(const wchar_t *buffer, wchar_t *fmt, ...)
{
 int rc;
 va_list arg_ptr;
 va_start(arg_ptr, fmt);
 rc = vswscanf(buffer, fmt, arg_ptr);
 va_end(arg_ptr);
 return(rc);
}
int main(void)
{
 wchar_t *tokenstring = L"15 12 14";
 char s[81];
 char c;
 int i;
 float fp;

 /* Input various data */

 vread(tokenstring, L"%s %c%d%f", s, &c, &i, &fp);

 /* Display the data */
 printf("¥nstring = %s¥n",s);
 printf("character = %c¥n",c);
 printf("integer = %d¥n",i);
 printf("floating-point number = %f¥n",fp);

}
/***************** Output should be similar to: *****************
string = 15
character = 1
integer = 2
floating-point number = 14.000000

*******************************************************************/
```

関連情報

- 138 ページの『fscanf() ― フォーマット済みデータの読み取り』
- 344 ページの『scanf() ― データの読み取り』
- 153 ページの『fwscanf() ― ワイド文字を使用したストリームからのデータの読み取り』
- 528 ページの『wscanf() ― ワイド文字書式ストリングを使用したデータの読み取り』
- 371 ページの『sscanf() ― データの読み取り』
- 368 ページの『sprintf() ― フォーマット設定データのバッファーへの出力』
- 425 ページの『swscanf() ― ワイド文字データの読み取り』
- 20 ページの『<wchar.h>』

---

vwprintf() ― 引数データのワイド文字としてのフォーマット設定と出力

フォーマット

```
#include <stdarg.h>
#include <wchar.h>
int vwprintf(const wchar_t *format, va_list arg);
```

言語レベル: ANSI

スレッド・セーフ: はい。

ロケール依存: この関数の振る舞いは、現行ロケールの LC_CTYPE カテゴリーおよび LC_NUMERIC カテゴリーの影響を受ける可能性があります。また、LOCALETYPE(*LOCALEUCS2) または LOCALETYPE(*LOCALEUTF) がコンパイル・コマンドに対して指定されている場合は、現行ロケールの LC_UNI_CTYPE カテゴリーおよび LC_UNI_NUMERIC カテゴリーにも影響を受ける可能性があります。この関数は、コンパイル・コマンドに対して LOCALETYPE(*CLD) が指定されている場合には使用できません。詳細については、 549 ページの『CCSID およびロケールの理解』を参照してください。

統合ファイル・システム・インターフェース: この関数は、コンパイル・コマンドに対して SYSIFCOPT(*NOIFSIO) が指定されている場合には使用できません。

ワイド文字関数: 詳細については、 553 ページの『ワイド文字』を参照してください。

説明

`vwprintf()` 関数は、`wprintf()` 関数と同等ですが、変数引数リストが、va_start マクロ (および、場合によっては後続の va_arg 呼び出し) によって初期化される *arg* によって置き換えられるという点が異なります。 `vwprintf()` 関数は、va_end マクロを呼び出しません。

戻り値

`vwprintf()` 関数は、送信されたワイド文字の数を戻します。出力エラーが発生した場合、`vwprintf()` は負の値を戻します。

`vwprintf()` の使用例

この例では、ワイド文字 *a* を出力します。この出力は `vout()` 関数から行われ、引数の可変値を取り、`vwprintf()` 関数を使用してそれらを stdout に出力します。

```
#include <wchar.h>
#include <stdarg.h>
#include <locale.h>

void vout (wchar_t *fmt, ...);

const char ifs_path[] = "tmp/mytest";

int main(void)     {
 FILE *stream;
 wchar_t format[] = L"%lc";
 setlocale(LC_ALL, "POSIX");
 vout (format, L'a');
return(0);

/* A long a is written to stdout, if stdout is written to the screen
   it may get converted back to a single byte 'a'.  */
}

 void vout (wchar_t *fmt, ...) {

 va_list arg_ptr;
 va_start (arg_ptr, fmt);
 vwprintf (fmt, arg_ptr);
 va_end (arg_ptr);
}
```

関連情報

- 238 ページの『printf() ― 定様式の文字の出力』
- 444 ページの『vfprintf() ― ストリームへの引数データの出力』
- 452 ページの『vprintf() ― 引数データの出力』
- 56 ページの『btowc() ― 1 バイト文字のワイド文字への変換』
- 210 ページの『mbrtowc() ― マルチバイト文字からワイド文字への変換 (再始動可能)』
- 149 ページの『fwprintf() ― ワイド文字としてのデータのフォーマット設定とストリームへの書き込み』
- 459 ページの『vswprintf() ― ワイド文字のフォーマット設定とバッファーへの書き込み』
- 447 ページの『vfwprintf() ― 引数データのワイド文字としてのフォーマット設定とストリームへの書き込み』
- 15 ページの『<stdarg.h>』
- 20 ページの『<wchar.h>』

vwscanf() ― フォーマット済みワイド文字データの読み取り

フォーマット

```
#include <stdarg.h>
#include <stdio.h>

int vwscanf(const wchar_t *format, va_list arg_ptr);
```

言語レベル: ANSI

スレッド・セーフ: はい。

ロケール依存: この関数の振る舞いは、現行ロケールの LC_CTYPE カテゴリーおよび LC_NUMERIC カテゴリーの影響を受ける可能性があります。また、LOCALETYPE(*LOCALEUCS2) または LOCALETYPE(*LOCALEUTF) がコンパイル・コマンドに対して指定されている場合は、現行ロケールの LC_UNI_CTYPE カテゴリーおよび LC_UNI_NUMERIC カテゴリーにも影響を受ける可能性があります。この関数は、コンパイル・コマンドに対して LOCALETYPE(*CLD) が指定されている場合には使用できません。詳細については、 549 ページの『CCSID およびロケールの理解』を参照してください。

統合ファイル・システム・インターフェース: この関数は、コンパイル・コマンドに対して SYSIFCOPT(*NOIFSIO) が指定されている場合には使用できません。

ワイド文字関数: 詳細については、 553 ページの『ワイド文字』を参照してください。

説明

`vwscanf()` 関数は、`stdin` から引数の可変値で指定されたロケーションへデータを読み取ります。`vwscanf()` 関数は、`wscanf()` 関数と同様に機能しますが、*arg_ptr* はプログラムの呼び出しによって個数が異なることがある引数のリストを指しているという点が異なります。これらの引数は、各呼び出しごとに va_start で初期化する必要があります。反対に、 `wscanf()` 関数は引数のリストを持てますが、そのリストの引数の数はプログラムをコンパイルしたときに決定されます。

各引数は、format-string の型指定子に対応する型を持つ変数へのポインターでなければなりません。*format* には、 `wscanf()` 関数の書式ストリングと同じ書式および関数があります。

戻り値

vwscanf() 関数は、正常に変換され、割り当てられたフィールドの数を戻します。戻り値には、読み取りは行われたが、割り当てられなかったフィールドは含まれません。変換が行われていない場合に、ファイルの終わりを読み取ろうとすると、戻り値は EOF になります。戻り値 0 は、フィールドが割り当てられなかったことを意味します。

vwscanf() の使用例

この例では、stdin からのさまざまなタイプのデータを走査します。

```
#include <stdio.h
#include <stdarg.h

int vread(wchar_t *fmt, ...)
{
 int rc;
 va_list arg_ptr;
 va_start(arg_ptr, fmt);
 rc = vwscanf(fmt, arg_ptr);
 va_end(arg_ptr);
 return(rc);
}

int main(void)
{
 int i, rc;
 float fp;
 char c, s[81];
 printf("Enter an integer, a real number, a character "
 "and a string : ¥n");
 rc = vread(L"%d %f %c %s",&i,&fp,&c, s);
 if (rc != 4)
  printf("Not all fields are assigned¥n");
 else
 {
  printf("integer = %d¥n", i);
  printf("real number = %f¥n", fp);
  printf("character = %c¥n", c);
  printf("string = %s¥n",s);
 }
}
/***************** If input is: 12 2.5 a yes, *******************
************** then output should be similar to: ****************
Enter an integer, a real number, a character and a string :
integer = 12
real number = 2.500000
character = a
string = yes
*/
```

関連情報

- 138 ページの『fscanf() ― フォーマット済みデータの読み取り』
- 344 ページの『scanf() ― データの読み取り』
- 371 ページの『sscanf() ― データの読み取り』
- 425 ページの『swscanf() ― ワイド文字データの読み取り』
- 153 ページの『fwscanf() ― ワイド文字を使用したストリームからのデータの読み取り』
- 528 ページの『wscanf() ― ワイド文字書式ストリングを使用したデータの読み取り』
- 368 ページの『sprintf() ― フォーマット設定データのバッファーへの出力』
- 17 ページの『<stdio.h>』

wcrtomb() ― ワイド文字からマルチバイト文字への変換 (再開可能)

フォーマット

```
#include <wchar.h>
size_t wcrtomb (char *s, wchar_t wc, mbstate_t *ps);
```

言語レベル: ANSI

スレッド・セーフ: *ps* が NULL の場合を除いて、対応しています。

ロケール依存: この関数の振る舞いは、現行ロケールの LC_CTYPE カテゴリーの影響を受ける可能性があります。また、この振る舞いは、LOCALETYPE(*LOCALEUCS2) または LOCALETYPE(*LOCALEUTF) がコンパイル・コマンドに対して指定されている場合は、現行ロケールの LC_UNI_CTYPE カテゴリーの影響を受ける可能性もあります。この関数は、コンパイル・コマンドに対して LOCALETYPE(*CLD) が指定されている場合には使用できません。詳細については、 549 ページの『CCSID およびロケールの理解』を参照してください。

ワイド文字関数: 詳細については、 553 ページの『ワイド文字』を参照してください。

説明

この関数は `wctomb()` 関数の再始動可能バージョンです。

`wcrtomb()` 関数は、ワイド文字をマルチバイト文字へ変換します。

s が NULL ポインターの場合には、`wcrtomb()` 関数は初期シフト状態 (エンコードが状態依存でないか、または初期変換状態が記述されている場合はゼロ) を入力するのに必要なバイト数を判別します。記述されている結果の状態は、初期変換状態です。

s が NULL ポインターでない場合は、`wcrtomb()` 関数は *wc* によって指定されたワイド文字に対応するマルチバイト文字 (シフト・シーケンスを含む) を表示するのに必要なバイト数を判別し、結果のバイトを最初のエレメントが *s* によって指定される配列に保管します。最大で MB_CUR_MAX バイトが保管されます。*wc* がヌル・ワイド文字の場合には、記述されている結果の状態は初期変換状態です。

この関数は、対応する内部状態のマルチバイト文字関数とは以下の点で異なります。この関数には追加のパラメーターとして mbstate_t の型を指す *ps* ポインターがあり、このポインターは、関連するマルチバイト文字シーケンスの現在の変換状態を完全に表すことができるオブジェクトを指します。 ps が NULL の場合、変換状態の追跡を継続するために内部静的変数が使用されます。内部静的変数の使用はスレッド・セーフではありません。

戻り値

s が NULL ポインターの場合には、`wcrtomb()` 関数は、初期シフト状態を入力するのに必要なバイト数を戻します。戻された値は、 MB_CUR_MAX マクロの値より大きくなりません。

s が NULL ポインターでない場合、`wcrtomb()` 関数は、 *wc* が有効なワイド文字のときに、配列オブジェクトに保管されているバイト数 (シフト・シーケンスを含む) を戻します。そうでない場合には (*wc* が有効なワイド文字でないときに)、エンコード・エラーが起こり、マクロ EILSEQ の値が errno に保管され、-1 が戻されますが変換状態は変更されないまま残ります。

変換エラーが発生した場合、`errno` は **ECONVERT** に設定される可能性があります。

`wcrtomb()` の使用例

このプログラムは LOCALETYPE(*LOCALE) および SYSIFCOPT(*IFSIO) でコンパイルされています。

```
#include <stdio.h>
#include <locale.h>
#include <wchar.h>
#include <errno.h>

#define  STRLENGTH   10
#define  LOCNAME     "qsys.lib/JA_JP.locale"
#define  LOCNAME_EN  "qsys.lib/EN_US.locale"

int main(void)
{
    char      string[STRLENGTH];
    int length, sl = 0;
    wchar_t   wc = 0x4171;
    wchar_t   wc2 = 0x00C1;
    wchar_t  wc_string[10];
    mbstate_t ps = 0;
    memset(string, '\0', STRLENGTH);
    wc_string[0] = 0x00C1;
    wc_string[1] = 0x4171;
    wc_string[2] = 0x4172;
    wc_string[3] = 0x00C2;
    wc_string[4] = 0x0000;
    /* In this first example we will convert a wide character */
    /* to a single byte character.  We first set the locale   */
    /* to a single byte locale.  We choose a locale with      */
    /* CCSID 37.  For single byte cases the state will always */
    /* remain in the initial state  0       */

    if (setlocale(LC_ALL, LOCNAME_EN) == NULL)
        printf("setlocale failed.\n");

    length = wcrtomb(string, wc, &ps);

    /* In this case since wc > 256 hex, lenth is -1 and  */
    /* errno is set to EILSEQ (3492)  */
    printf("errno = %d, length = %d\n\n", errno, length);

    length = wcrtomb(string, wc2, &ps);

    /* In this case wc2 00C1 is converted to C1  */

    printf("string = %s\n\n", string);

    /* Now lets try a multibyte example.  We first must set the */
    /* locale to a multibyte locale.  We choose a locale with     */
    /* CCSID 5026  */

    if (setlocale(LC_ALL, LOCNAME) == NULL)
        printf("setlocale failed.\n");

    length = wcrtomb(string, wc_string[0], &ps);

    /* The first character is < 256 hex so is converted to   */
    /* single byte and the state is still the initial state 0  */

    printf("length = %d, state = %d\n\n", length, ps);

    sl += length;

    length = wcrtomb(&string[sl], wc_string[1], &ps);

    /* The next character is > 256 hex so we get a shift out   */
    /* 0x0e followed by the double byte character.  State is   */
    /* changed to double byte state.  Length is 3.             */
```

```
    printf("length = %d, state = %d\n\n", length, ps);

    sl += length;

    length = wcrtomb(&string[sl], wc_string[2], &ps);

    /* The next character is > 256 hex so we get another       */
    /* double byte character.   The state is left in           */
    /* double byte state.  Length is 2.                        */

    printf("length = %d, state = %d\n\n", length, ps);

    sl += length;

    length = wcrtomb(&string[sl], wc_string[3], &ps);

    /* The next character is < 256 hex so we close off the     */
    /* double byte characters with a shift in 0x0f and then    */
    /* get a single byte character.  Length is 2.              */
    /* The hex look at string would now be:                    */
    /* C10E417141720FC2                                        */
    /* You would need a device capable of displaying multibyte */
    /* characters to see this string.                          */

    printf("length = %d, state = %d\n\n", length, ps);

    /* In the last example we will show what happens if NULL   */
    /* is passed in for the state.                             */
    memset(string, '\0', STRLENGTH);

    length = wcrtomb(string, wc_string[1], NULL);

    /* The second character is > 256 hex so a shift out       */
    /* followed by the double character is produced but since  */
    /* the state is NULL, the double byte character is closed  */
    /* off with a shift in right away.  So string we look      */
    /* like this:  0E41710F  and length is 4 and the state is  */
    /* left in the initial state.                              */

    printf("length = %d, state = %d\n\n", length, ps);

}
/*  The output should look like this:

errno = 3492, length = -1

string = A

length = 1, state = 0

length = 3, state = 2

length = 2, state = 2

length = 2, state = 0

length = 4, state = 0
                                   */
```

このプログラムは LOCALETYPE(*LOCALEUCS2) および SYSIFCOPT(*IFSIO) でコンパイルされています。

```
#include <stdio.h>
#include <locale.h>
#include <wchar.h>
#include <errno.h>
```

```
#define  STRLENGTH   10
#define  LOCNAME     "qsys.lib/JA_JP.locale"
#define  LOCNAME_EN  "qsys.lib/EN_US.locale"

int main(void)
{
    char      string[STRLENGTH];
    int length, sl = 0;
    wchar_t   wc = 0x4171;
    wchar_t   wc2 = 0x0041;
    wchar_t  wc_string[10];
    mbstate_t ps = 0;
    memset(string, '\0', STRLENGTH);
    wc_string[0] = 0x0041;
    wc_string[1] = 0xFF31;
    wc_string[2] = 0xFF32;
    wc_string[3] = 0x0042;
    wc_string[4] = 0x0000;
    /* In this first example we will convert a UNICODE character */
    /* to a single byte character.  We first set the locale   */
    /* to a single byte locale.  We choose a locale with      */
    /* CCSID 37.  For single byte cases the state will always */
    /* remain in the initial state  0       */

    if (setlocale(LC_ALL, LOCNAME_EN) == NULL)
        printf("setlocale failed.\n");

    length = wcrtomb(string, wc2, &ps);

    /* In this case wc2 0041 is converted to C1  */
    /* 0041 is UNICODE A, C1 is CCSID 37 A       */

    printf("string = %s\n\n", string);

    /* Now lets try a multibyte example.  We first must set the */
    /* locale to a multibyte locale.  We choose a locale with     */
    /* CCSID 5026  */

    if (setlocale(LC_ALL, LOCNAME) == NULL)
        printf("setlocale failed.\n");

    length = wcrtomb(string, wc_string[0], &ps);

    /* The first character UNICODE character is converted to a */
    /* single byte and the state is still the initial state 0  */

    printf("length = %d, state = %d\n\n", length, ps);

    sl += length;

    length = wcrtomb(&string[sl], wc_string[1], &ps);

    /* The next UNICODE character is converted to a shift out   */
    /* 0x0e followed by the double byte character.  State is   */
    /* changed to double byte state.  Length is 3.             */

    printf("length = %d, state = %d\n\n", length, ps);

    sl += length;

    length = wcrtomb(&string[sl], wc_string[2], &ps);

    /* The UNICODE character is converted to another       */
    /* double byte character.   The state is left in           */
    /* double byte state.  Length is 2.                        */
```

```
    printf("length = %d, state = %d\n\n", length, ps);

    sl += length;

    length = wcrtomb(&string[sl], wc_string[3], &ps);

    /* The next UNICODE character converts to single byte so   */
    /* we close off the                                        */
    /* double byte characters with a shiftin 0x0f and then     */
    /* get a single byte character.  Length is 2.              */
    /* The hex look at string would now be:                    */
    /* C10E42D842D90FC2                                        */
    /* You would need a device capable of displaying multibyte */
    /* characters to see this string.                          */

    printf("length = %d, state = %d\n\n", length, ps);


}
/*  The output should look like this:

string = A

length = 1, state = 0

length = 3, state = 2

length = 2, state = 2

length = 2, state = 0
                                    */
```

関連情報

- 205 ページの『mblen() ― マルチバイト文字の長さの計算』
- 207 ページの『mbrlen() ― マルチバイト文字の長さの計算 (再始動可能)』
- 210 ページの『mbrtowc() ― マルチバイト文字からワイド文字への変換 (再始動可能)』
- 214 ページの『mbsrtowcs() ― マルチバイト・ストリングからワイド文字ストリングへの変換 (再始動可能)』
- 494 ページの『wcsrtombs() ― ワイド文字ストリングからマルチバイト・ストリングへの変換 (再開可能)』
- 515 ページの『wctomb() ― ワイド文字からマルチバイト文字への変換』
- 20 ページの『<wchar.h>』

wcscat() ― ワイド文字ストリングの連結

フォーマット

```
#include <wchar.h>
wchar_t *wcscat(wchar_t *string1, const wchar_t *string2);
```

言語レベル: XPG4

スレッド・セーフ: はい。

ワイド文字関数: 詳細については、 553 ページの『ワイド文字』 を参照してください。

説明

wcscat() 関数は、*string2* が指すストリングのコピーを *string1* が指すストリングの終わりに追加します。

wcscat() 関数は、ヌル終了 wchar_t ストリング上で作動します。この関数のストリング引数には、ストリングの終わりを示す wchar_t ヌル文字が入っていなければなりません。境界検査は実行されません。

戻り値

wcscat() 関数は、連結された *string1* へのポインターを戻します。

wcscat() の使用例

この例では、wcscat() 関数を使用して、ワイド文字ストリング ″computer program″ を作成します。

```
#include <stdio.h>
#include <wchar.h>

#define SIZE 40


int main(void)
{
  wchar_t buffer1[SIZE] = L"computer";
  wchar_t * string      = L" program";
  wchar_t * ptr;

  ptr = wcscat( buffer1, string );
  printf( "buffer1 = %ls¥n", buffer1 );

}

/****************  Output should be similar to:  ******************

buffer1 = computer program
******************************************************************/
```

関連情報

- 374 ページの『strcat() ― ストリングの連結』
- 394 ページの『strncat() ― ストリングの連結』
- 『wcschr() ― ワイド文字の検索』
- 474 ページの『wcscmp() ― ワイド文字ストリングの比較』
- 476 ページの『wcscpy() ― ワイド文字ストリングのコピー』
- 477 ページの『wcscspn() ― 最初に一致したワイド文字のオフセットの検索』
- 482 ページの『wcslen() ― ワイド文字ストリング長の計算』
- 484 ページの『wcsncat() ― ワイド文字ストリングの連結』
- 20 ページの『<wchar.h>』

wcschr() ― ワイド文字の検索

フォーマット

```
#include <wchar.h>
wchar_t *wcschr(const wchar_t *string, wchar_t character);
```

言語レベル: XPG4

スレッド・セーフ: はい。

ワイド文字関数: 詳細については、 553 ページの『ワイド文字』 を参照してください。

説明

`wcschr()` 関数は、ワイド文字 *string* 内の *character* のオカレンスを検索します。*character* は、wchar_t ヌル文字 (¥0) にすることができます。 *string* の終わりにある wchar_t ヌル文字は検索に含まれます。

`wcschr()` 関数は、ヌル終了 wchar_t ストリング上で作動します。この関数のストリング引数には、ストリングの終わりを示す wchar_t ヌル文字が入っていなければなりません。

戻り値

`wcschr()` 関数は、*string* 内の *character* が最初に現れる位置へのポインターを戻します。文字が検出されない場合は、NULL ポインターが戻されます。

`wcschr()` の使用例

この例では、ワイド文字ストリング ″computer program″ 内の文字 ″p″ が最初に現れる位置を検索します。

```
#include <stdio.h>
#include <wchar.h>

#define SIZE 40

int main(void)
{
  wchar_t buffer1[SIZE] = L"computer program";
  wchar_t * ptr;
  wchar_t ch = L'p';

  ptr = wcschr( buffer1, ch );
  printf( "The first occurrence of %lc in '%ls' is '%ls'¥n",
                        ch, buffer1, ptr );

}

/****************  Output should be similar to: ******************

The first occurrence of p in 'computer program' is 'puter program'
*/
```

関連情報

- 375 ページの『strchr() ― 文字の検索』
- 381 ページの『strcspn() ― 最初に一致した文字のオフセットの検索』
- 401 ページの『strpbrk() ― ストリング内の文字の検索』
- 406 ページの『strrchr() ― ストリング内で文字が最後に現れる位置の検出』
- 407 ページの『strspn() ―最初の不一致文字のオフセットの検索』
- 471 ページの『wcscat() ― ワイド文字ストリングの連結』
- 474 ページの『wcscmp() ― ワイド文字ストリングの比較』
- 476 ページの『wcscpy() ― ワイド文字ストリングのコピー』
- 477 ページの『wcscspn() ― 最初に一致したワイド文字のオフセットの検索』
- 482 ページの『wcslen() ― ワイド文字ストリング長の計算』
- 485 ページの『wcsncmp() ― ワイド文字ストリングの比較』
- 490 ページの『wcspbrk() ― ストリング内のワイド文字の位置検出』

- 493 ページの『wcsrchr() ― ストリング内でワイド文字が最後に現れる位置の検出』
- 496 ページの『wcsspn() ― 最初の不一致ワイド文字のオフセットの検索』
- 510 ページの『wcswcs() ― ワイド文字サブストリングの位置検出』
- 20 ページの『<wchar.h>』

wcscmp() ― ワイド文字ストリングの比較

フォーマット

```
#include <wchar.h>
int wcscmp(const wchar_t *string1, const wchar_t *string2);
```

言語レベル: ANSI

スレッド・セーフ: はい。

ワイド文字関数: 詳細については、 553 ページの『ワイド文字』 を参照してください。

説明

wcscmp() 関数は、2 つのワイド文字ストリングを比較します。wcscmp() 関数は、ヌル終了 wchar_t ストリング上で作動します。この関数のストリング引数には、ストリングの終わりを示す wchar_t ヌル文字が入っていなければなりません。ストリングを追加またはコピーするとき、境界検査は行われません。

戻り値

wcscmp() 関数は、2 つのストリング間の関係を示す次のような値を戻します。

値　意味

0 より小さい値
string1 は *string2* より小さい

0
string1 は *string2* と等しい

0 より大きい値
string1 は *string2* より大きい。

wcscmp() の使用例

この例では、wcscmp() を使用して、ワイド文字ストリング *string1* と *string2* を比較します。

```
#include <stdio.h>
#include <wchar.h>

int main(void)
{
  int  result;
  wchar_t string1[] = L"abcdef";
  wchar_t string2[] = L"abcdefg";

  result = wcscmp( string1, string2 );

  if ( result == 0 )
    printf( "¥"%ls¥" is identical to ¥"%ls¥"¥n", string1, string2);
  else if ( result < 0 )
    printf( "¥"%ls¥" is less than ¥"%ls¥"¥n", string1, string2 );
  else
    printf( "¥"%ls¥" is greater than ¥"%ls¥"¥n", string1, string2);
```

```
}

/****************  Output should be similar to:  ******************

"abcdef" is less than "abcdefg"
*/
```

関連情報

- 376 ページの『strcmp() ― ストリングの比較』
- 395 ページの『strncmp() ― ストリングの比較』
- 471 ページの『wcscat() ― ワイド文字ストリングの連結』
- 472 ページの『wcschr() ― ワイド文字の検索』
- 476 ページの『wcscpy() ― ワイド文字ストリングのコピー』
- 477 ページの『wcscspn() ― 最初に一致したワイド文字のオフセットの検索』
- 482 ページの『wcslen() ― ワイド文字ストリング長の計算』
- 485 ページの『wcsncmp() ― ワイド文字ストリングの比較』
- 480 ページの『__wcsicmp() ― 大/小文字の区別をしないワイド文字ストリングの比較』
- 488 ページの『__wcsnicmp() ― 大/小文字の区別をしないワイド文字ストリングの比較』
- 20 ページの『<wchar.h>』

wcscoll() ―言語照合ストリングの比較

フォーマット

```
#include <wchar.h>
int wcscoll (const wchar_t *wcs1, const wchar_t *wcs2);
```

言語レベル: XPG4

スレッド・セーフ: はい。

ロケール依存: コンパイル・コマンドで LOCALETYPE(*LOCALE) が指定される場合、この関数の振る舞いは、現行ロケールの LC_COLLATE カテゴリーの影響を受ける可能性があります。また、コンパイル・コマンドで LOCALETYPE(*LOCALEUCS2) または LOCALETYPE(*LOCALEUTF) が指定される場合、この関数の振る舞いは、現行ロケールの LC_UNI_COLLATE カテゴリーの影響も受ける可能性があります。この関数は、コンパイル・コマンドに対して LOCALETYPE(*CLD) が指定されている場合には使用できません。詳細については、 549 ページの『CCSID およびロケールの理解』を参照してください。

ワイド文字関数: 詳細については、 553 ページの『ワイド文字』 を参照してください。

説明

wcscoll() 関数は、 *wcs1* および *wcs2* が指すワイド文字ストリングを比較します。これらは両方とも現行ロケールの LC_COLLATE カテゴリー (または、UNICODE LOCALETYPE が指定されている場合は LC_UNI_COLLATE カテゴリー) に対して適切なものと解釈されます。

戻り値

wcscoll() 関数は、ストリング間の関係を示す次のような整数値を戻します。

値　　意味

0 より小さい値
wcs1 は *wcs2* より小さい

0
wcs1 は *wcs2* と等しい

0 より大きい値
wcs1 は *wcs2* より大きい

wcs1 または *wcs2* に、照合シーケンスのドメイン外の文字が含まれる場合、wcscoll() 関数は errno を EINVAL に設定します。エラーが発生した場合、wcscoll() 関数は errno を非ゼロ値に設定します。エラーの戻り値はありません。

wcscoll() の使用例

この例はデフォルト・ロケールを使用しています。

```
#include <stdio.h>
#include <wchar.h>

int main(void)
{
   int result;
   wchar_t *wcs1 = L"first_wide_string";
   wchar_t *wcs2 = L"second_wide_string";

   result = wcscoll(wcs1, wcs2);

   if ( result == 0)
      printf("¥"%S¥" is identical to ¥"%S¥"¥n", wcs1, wcs2);
   else if ( result < 0)
      printf("¥"%S¥" is less than ¥"%S¥"¥n", wcs1, wcs2);
   else
      printf("¥"%S¥" is greater than ¥"%S¥"¥n", wcs1, wcs2);
}
```

関連情報

- 379 ページの『strcoll() ― ストリングの比較』
- 354 ページの『setlocale() ― ロケールの設定』
- 20 ページの『<wchar.h>』

wcscpy() ― ワイド文字ストリングのコピー

フォーマット

```
#include <wchar.h>
wchar_t *wcscpy(wchar_t *string1, const wchar_t *string2);
```

言語レベル: XPG4

スレッド・セーフ: はい。

ワイド文字関数: 詳細については、 553 ページの『ワイド文字』 を参照してください。

説明

wcscpy() 関数は、(終了 wchar_t ヌル文字を含む) *string2* の内容を *string1* にコピーします。

wcscpy() 関数は、ヌル終了 wchar_t ストリング上で作動します。この関数のストリング引数には、ストリングの終わりを示す wchar_t ヌル文字が入っていなければなりません。ヌル文字を含む必要があるのは *string2* のみです。境界検査は実行されません。

戻り値

wcscpy() 関数は *string1* へのポインターを戻します。

wcscpy() の使用例

この例では、source の内容を destination にコピーします。

```
#include <stdio.h>
#include <wchar.h>

#define SIZE    40

int main(void)
{
  wchar_t source[ SIZE ] = L"This is the source string";
  wchar_t destination[ SIZE ] = L"And this is the destination string";
  wchar_t * return_string;

  printf( "destination is originally = ¥"%ls¥"¥n", destination );
  return_string = wcscpy( destination, source );
  printf( "After wcscpy, destination becomes ¥"%ls¥"¥n", destination );
}

/****************  Output should be similar to:  ******************

destination is originally = "And this is the destination string"
After wcscpy, destination becomes "This is the source string"
*/
```

関連情報

- 380 ページの『strcpy() ― ストリングのコピー』
- 397 ページの『strncpy() ― ストリングのコピー』
- 471 ページの『wcscat() ― ワイド文字ストリングの連結』
- 472 ページの『wcschr() ― ワイド文字の検索』
- 474 ページの『wcscmp() ― ワイド文字ストリングの比較』
- 『wcscspn() ― 最初に一致したワイド文字のオフセットの検索』
- 482 ページの『wcslen() ― ワイド文字ストリング長の計算』
- 487 ページの『wcsncpy() ― ワイド文字ストリングのコピー』
- 20 ページの『<wchar.h>』

wcscspn() ― 最初に一致したワイド文字のオフセットの検索

フォーマット

```
#include <wchar.h>
size_t wcscspn(const wchar_t *string1, const wchar_t *string2);
```

言語レベル: XPG4

スレッド・セーフ: はい。

ワイド文字関数: 詳細については、 553 ページの『ワイド文字』 を参照してください。

説明

`wcscspn()` 関数は、 *string1* が指すストリングの初期セグメントにあって、 *string2* が指すストリングに現れない wchar_t 文字数を判別します。

`wcscspn()` 関数は、ヌル終了 wchar_t ストリング上で作動します。この関数のストリング引数には、ストリングの終わりを示す wchar_t ヌル文字が入っていなければなりません。

戻り値

`wcscspn()` 関数は、wchar_t 文字数をセグメントに戻します。

`wcscspn()` の使用例

この例では `wcscspn()` を使用して、string で a、x、l、または e のいずれかの文字が最初に現れる位置を検索します。

```
#include <stdio.h>
#include <wchar.h>

#define SIZE 40

int main(void)
{
  wchar_t string[SIZE] = L"This is the source string";
  wchar_t * substring = L"axle";

  printf( "The first %i characters in the string ¥"%ls¥" are not in the "
          "string ¥"%ls¥" ¥n", wcscspn( string, substring),
          string, substring );
}

/****************  Output should be similar to:  ******************

The first 10 characters in the string "This is the source string" are not
in the string "axle"
*/
```

関連情報

- 381 ページの『strcspn() — 最初に一致した文字のオフセットの検索』
- 407 ページの『strspn() —最初の不一致文字のオフセットの検索』
- 471 ページの『wcscat() — ワイド文字ストリングの連結』
- 472 ページの『wcschr() — ワイド文字の検索』
- 474 ページの『wcscmp() — ワイド文字ストリングの比較』
- 476 ページの『wcscpy() — ワイド文字ストリングのコピー』
- 482 ページの『wcslen() — ワイド文字ストリング長の計算』
- 496 ページの『wcsspn() — 最初の不一致ワイド文字のオフセットの検索』
- 510 ページの『wcswcs() — ワイド文字サブストリングの位置検出』
- 20 ページの『<wchar.h>』

wcsftime() ー フォーマット済み日時への変換

フォーマット

```
#include <wchar.h>
size_t wcsftime(wchar_t *wdest, size_t maxsize,
                const wchar_t *format, const struct tm *timeptr);
```

言語レベル: ANSI

スレッド・セーフ: はい。

ロケール依存: コンパイル・コマンドで LOCALETYPE(*LOCALE) が指定される場合、この関数の振る舞いは、現行ロケールの LC_CTYPE、LC_TIME、および LC_TOD カテゴリーの影響を受ける可能性があります。また、コンパイル・コマンドで LOCALETYPE(*LOCALEUCS2) または LOCALETYPE (*LOCALEUTF) が指定される場合、この関数の振る舞いは、現行ロケールの LC_UNI_CTYPE、LC_UNI_TIME、および LC_UNI_TOD カテゴリーの影響も受ける可能性があります。この関数は、コンパイル・コマンドに対して LOCALETYPE(*CLD) が指定されている場合には使用できません。詳細については、 549 ページの『CCSID およびロケールの理解』を参照してください。

ワイド文字関数: 詳細については、 553 ページの『ワイド文字』 を参照してください。

説明

`wcsftime()` 関数は、*timeptr* 構造体の日時仕様をワイド文字ストリングに変換します。フォーマットで示される書式ストリングに応じて、*wdest* で示される配列にヌル終了ストリングを保管します。*maxsize* 値によって、配列にコピーできるワイド文字の最大数が指定できます。この関数は、`strftime()` と同等ですが、ワイド文字を使用する点が異なります。

`wcsftime()` 関数は `strftime()` 関数に似ていますが、ワイド文字を使用する点が異なります。書式ストリングは、以下を含むワイド文字の文字ストリングです。

- 変換指定文字。
- 変換されずに配列にコピーされる通常のワイド文字。

この関数は、*timeptr* が示す時間構造体を使用し、指定子がロケール依存である場合には、現行ロケールの LC_TIME カテゴリーも使用して、有効な各指定子の適切な置換値を判別します。*timeptr* が示す時間構造体は、通常は `gmtime()` または `localtime()` 関数の呼び出しにより取得されます。

戻り値

終了ヌル・ワイド文字を含む結果ストリングのワイド文字の合計数が *maxsize* を超えない場合には、`wcsftime()` は、*wdest* に配置されたワイド文字数 (終了ヌル・ワイド文字を含まない) を戻します。それ以外の場合は、`wcsftime()` 関数は 0 を戻し、配列の内容は不確定となります。

変換エラーが発生した場合、`errno` は **ECONVERT** に設定される可能性があります。

`wcsftime()` の使用例

この例では、`localtime()` を使用して日付と時刻を取得し、 `wcsftime()` でその情報をフォーマット設定し、日付と時刻を出力します。

```
#include <stdio.h>
#include <time.h>
#include <wchar.h>
```

```
int main(void)
{
   struct tm *timeptr;
   wchar_t   dest[100];
   time_t    temp;
   size_t    rc;

   temp = time(NULL);
   timeptr = localtime(&temp);
   rc = wcsftime(dest, sizeof(dest), L" Today is %A,"
                 L" %b %d.¥n Time: %I:%M %p", timeptr);
   printf("%d characters placed in string to make:¥n¥n%ls¥n", rc, dest);
   return 0;

   /********************************************************************
      The output should be similar to:

      43 characters placed in string to make:

       Today is Thursday, Nov 10.
       Time: 04:56 PM
   ********************************************************************/
}
```

関連情報

- 75 ページの『ctime() — 時間から文字ストリングへの変換』
- 77 ページの『ctime64() — 時間から文字ストリングへの変換』
- 80 ページの『ctime64_r() — 時間から文字ストリングへの変換 (再始動可能)』
- 78 ページの『ctime_r() — 時間から文字ストリングへの変換 (再始動可能)』
- 167 ページの『gmtime() — 時間の変換』
- 169 ページの『gmtime64() — 時間の変換』
- 173 ページの『gmtime64_r() — 時間の変換 (再始動可能)』
- 171 ページの『gmtime_r() — 時間の変換 (再始動可能)』
- 192 ページの『localtime() — 時間の変換』
- 194 ページの『localtime64() — 時間の変換』
- 197 ページの『localtime64_r() — 時間の変換 (再始動可能)』
- 195 ページの『localtime_r() — 時間の変換 (再始動可能)』
- 387 ページの『strftime() — 日付/時刻からストリングへの変換』
- 402 ページの『strptime()— ストリングから日付/時刻への変換』
- 429 ページの『time() — 現在時刻の判別』
- 430 ページの『time64() — 現在時刻の判別』
- 20 ページの『<wchar.h>』

__wcsicmp() — 大/小文字の区別をしないワイド文字ストリングの比較

フォーマット

```
#include <wchar.h>
int __wcsicmp(const wchar_t *string1, const wchar_t *string2);
```

言語レベル: Extension

スレッド・セーフ: はい。

ロケール依存: コンパイル・コマンドで LOCALETYPE(*LOCALE) が指定される場合、この関数の振る舞いは、現行ロケールの LC_CTYPE カテゴリーの影響を受ける可能性があります。また、コンパイル・コマンドで LOCALETYPE(*LOCALEUCS2) または LOCALETYPE(*LOCALEUTF) が指定される場合、この関数の振る舞いは、現行ロケールの LC_UNI_CTYPE カテゴリーの影響も受ける可能性があります。この関数は、コンパイル・コマンドに対して LOCALETYPE(*CLD) が指定されている場合には使用できません。詳細については、 549 ページの『CCSID およびロケールの理解』を参照してください。

ワイド文字関数: 詳細については、 553 ページの『ワイド文字』 を参照してください。

説明

__wcsicmp() 関数は、*string1* と *string2* を大/小文字の区別なしで比較します。*string1* と *string2* のすべての英字ワイド文字は、比較の前に小文字に変換されます。この関数は、ヌル終了ワイド文字ストリング上で作動します。この関数のストリング引数には、ストリングの終わりを示す wchar_t ヌル文字 (L'¥0') が含まれていなければなりません。

戻り値

__wcsicmp() 関数は、2 つのストリング間の関係を示す次のような値を戻します。

表 10. __wcsicmp() の戻り値

値	意味
0 より小さい値	*string1* は *string2* より小さい
0	*string1* は *string2* と等しい
0 より大きい値	*string1* は *string2* より大きい

.

__wcsicmp() の使用例

この例では __wcsicmp() を使用して、2 つのワイド文字ストリングを比較します。

```
#include <stdio.h>
#include <wchar.h>

int main(void)
{
  wchar_t *str1 = L"STRING";
  wchar_t *str2 = L"string";
  int result;

  result = __wcsicmp(str1, str2);

  if (result == 0)
    printf("Strings compared equal.¥n");
  else if (result < 0)
    printf("¥"%ls¥" is less than ¥"%ls¥".¥n", str1, str2);
  else
    printf("¥"%ls¥" is greater than ¥"%ls¥".¥n", str1, str2);

  return 0;
}

/********  The output should be similar to: ***************

Strings compared equal.

***********************************/
```

関連情報

- 376 ページの『strcmp() ― ストリングの比較』
- 395 ページの『strncmp() ― ストリングの比較』
- 471 ページの『wcscat() ― ワイド文字ストリングの連結』
- 472 ページの『wcschr() ― ワイド文字の検索』
- 477 ページの『wcscspn() ― 最初に一致したワイド文字のオフセットの検索』
- 『wcslen() ― ワイド文字ストリング長の計算』
- 485 ページの『wcsncmp() ― ワイド文字ストリングの比較』
- 488 ページの『__wcsnicmp() ― 大/小文字の区別をしないワイド文字ストリングの比較』
- 20 ページの『<wchar.h>』

wcslen() ― ワイド文字ストリング長の計算

フォーマット

```
#include <wchar.h>
size_t wcslen(const wchar_t *string);
```

言語レベル: XPG4

スレッド・セーフ: はい。

ワイド文字関数: 詳細については、 553 ページの『ワイド文字』 を参照してください。

説明

wcslen() 関数は、 *string* が指すストリング内のワイド文字数を計算します。

戻り値

wcslen() 関数は、終了 wchar_t ヌル文字以外の *string* 内のワイド文字数を戻します。

wcslen() の使用例

この例では、ワイド文字ストリング string の長さを計算します。

```
#include <stdio.h>
#include <wchar.h>

int main(void)
{
  wchar_t * string = L"abcdef";

  printf( "Length of ¥"%ls¥" is %i¥n", string, wcslen( string ));
}

/****************  Output should be similar to:  ******************

Length of "abcdef" is 6
*/
```

関連情報

- 205 ページの『mblen() ― マルチバイト文字の長さの計算』
- 392 ページの『strlen() ― ストリング長の判別』

- 484 ページの『wcsncat() ― ワイド文字ストリングの連結』
- 485 ページの『wcsncmp() ― ワイド文字ストリングの比較』
- 487 ページの『wcsncpy() ― ワイド文字ストリングのコピー』
- 20 ページの『<wchar.h>』

wcslocaleconv() ― ワイド・ロケール情報の検索

フォーマット

```
#include <locale.h>
struct wcslconv *wcslocaleconv(void);
```

言語レベル: Extended

スレッド・セーフ: はい。

ロケール依存: この関数の振る舞いは、現行ロケールの LC_UNI_NUMERIC カテゴリーおよび LC_UNI_MONETARY カテゴリーの影響を受ける可能性があります。この関数は、コンパイル・モードで LOCALETYPE(*LOCALEUCS2) または LOCALETYPE(*LOCALEUTF) が指定される場合のみ使用可能です。詳細については、 549 ページの『CCSID およびロケールの理解』を参照してください。

ワイド文字関数: 詳細については、 553 ページの『ワイド文字』 を参照してください。

説明

`wcslocaleconv()` 関数は `localeconv` 関数と同じですが、 *lconv* 構造体のワイド・バージョンである *wcslconv* 構造体へのポインターを戻す点が異なります。これらのエレメントは、現行ロケールの LC_UNI_MONETARY カテゴリーおよび LC_UNI_NUMERIC カテゴリーによって判別されます。

戻り値

`wcslocaleconv()` 関数は、*wcslconv* 構造体へのポインターを戻します。

`wcslocaleconv()` の使用例

この例では、フランス語ロケールの Unicode 通貨記号を出力します。

```
/***********************************************************************
This example prints out the Unicode currency symbol for a French
locale.  You first must create a Unicode French locale.  You can do
this with this command:
CRTLOCALE LOCALE('QSYS.LIB/MYLIB.LIB/LC_UNI_FR.LOCALE') +
SRCFILE('QSYS.LIB/QSYSLOCALE.LIB/QLOCALESRC.FILE/ +
FR_FR.MBR') CCSID(13488)

Then you must compile your c program with LOCALETYPE(*LOCALEUCS2)
**********************************************************************/
#include <stdio.h>
#include <locale.h>
int main(void) {
char * string;
struct wcslconv * mylocale;
if (NULL != (string = setlocale(LC_UNI_ALL,
"QSYS.LIB/MYLIB.LIB/LC_UNI_FR.LOCALE"))) {
mylocale = wcslocaleconv();
/* Display the Unicode currency symbol in a French locale */
printf("French Unicode currency symbol is a %ls¥n", mylocale->currency_symbol);
} else {
printf("setlocale(LC_UNI_ALL, ¥"QSYS.LIB/MYLIB.LIB/LC_UNI_FR.LOCALE¥") ¥
returned <NULL>¥n");
}

return 0;
}
```

関連情報

- 354 ページの『setlocale() ― ロケールの設定』
- 8 ページの『<locale.h>』
- 187 ページの『localeconv() ― 環境からの情報の取得』

wcsncat() ― ワイド文字ストリングの連結

フォーマット

```
#include <wchar.h>
wchar_t *wcsncat(wchar_t *string1, const wchar_t *string2, size_t count);
```

言語レベル: XPG4

スレッド・セーフ: はい。

ワイド文字関数: 詳細については、 553 ページの『ワイド文字』 を参照してください。

説明

wcsncat() 関数は、*string2* から最大 *count* ワイド文字を *string1* の終わりに追加し、wchar_t ヌル文字を結果に追加します。

wcsncat() 関数は、ヌル終了ワイド文字ストリング上で作動します。この関数のストリング引数には、ストリングの終わりを示す wchar_t ヌル文字が入っていなければなりません。

戻り値

wcsncat() 関数は *string1* を戻します。

wcsncat() の使用例

この例では、wcscat() 関数と wcsncat() 関数の違いを説明します。 wcscat() 関数は、2 番目のストリング全体を最初のストリングに追加します。wcsncat() 関数は、 2 番目のストリング内の指定された文字数だけを最初のストリングに追加します。

```
#include <stdio.h>
#include <wchar.h>
#include <string.h>

#define SIZE 40

int main(void)
{
  wchar_t buffer1[SIZE] = L"computer";
  wchar_t * ptr;

  /* Call wcscat with buffer1 and " program" */

  ptr = wcscat( buffer1, L" program" );
  printf( "wcscat : buffer1 = ¥"%ls¥"¥n", buffer1 );

  /* Reset buffer1 to contain just the string "computer" again */

  memset( buffer1, L'¥0', sizeof( buffer1 ));
  ptr = wcscpy( buffer1, L"computer" );

  /* Call wcsncat with buffer1 and " program" */
  ptr = wcsncat( buffer1, L" program", 3 );
  printf( "wcsncat: buffer1 = ¥"%ls¥"¥n", buffer1 );
}
/****************  Output should be similar to: ******************

wcscat : buffer1 = "computer program"
wcsncat: buffer1 = "computer pr"
*/
```

関連情報

- 374 ページの『strcat() ― ストリングの連結』
- 394 ページの『strncat() ― ストリングの連結』
- 471 ページの『wcscat() ― ワイド文字ストリングの連結』
- 『wcsncmp() ― ワイド文字ストリングの比較』
- 487 ページの『wcsncpy() ― ワイド文字ストリングのコピー』
- 20 ページの『<wchar.h>』

---

wcsncmp() ― ワイド文字ストリングの比較

フォーマット

```
#include <wchar.h>
int wcsncmp(const wchar_t *string1, const wchar_t *string2, size_t count);
```

言語レベル: XPG4

スレッド・セーフ: はい。

ワイド文字関数: 詳細については、 553 ページの『ワイド文字』 を参照してください。

説明

wcsncmp() 関数は、*string1* から *string2* 間の最大の *count* ワイド文字までを比較します。

wcsncmp() 関数は、ヌル終了ワイド文字ストリング上で作動します。この関数のストリング引数には、ストリングの終わりを示す wchar_t ヌル文字が入っていなければなりません。

戻り値

wcsncmp() 関数は、2 つのストリング間の関係を示す次のような値を戻します。

値　　意味

0 より小さい値
string1 は *string2* より小さい

0　*string1* は *string2* と等しい

0 より大きい値
string1 は *string2* より大きい。

wcsncmp() の使用例

この例では、ストリング全体を比較する wcscmp() 関数と、ストリング内の指定された数のワイド文字のみを比較する wcsncmp() 関数の違いについて説明します。

```
#include <stdio.h>
#include <wchar.h>

#define SIZE 10

int main(void)
{
  int  result;
  int  index = 3;
  wchar_t buffer1[SIZE] = L"abcdefg";
  wchar_t buffer2[SIZE] = L"abcfg";
  void print_result( int, wchar_t *, wchar_t * );

  result = wcscmp( buffer1, buffer2 );
  printf( "Comparison of each character¥n" );
  printf( "  wcscmp: " );
  print_result( result, buffer1, buffer2 );

  result = wcsncmp( buffer1, buffer2, index);
  printf( "¥nComparison of only the first %i characters¥n", index );
  printf( "  wcsncmp: " );
  print_result( result, buffer1, buffer2 );
}

void print_result( int res, wchar_t * p_buffer1, wchar_t * p_buffer2 )
{
  if ( res == 0 )
    printf( "¥"%ls¥" is identical to ¥"%ls¥"¥n", p_buffer1, p_buffer2);
  else if ( res < 0 )
    printf( "¥"%ls¥" is less than ¥"%ls¥"¥n", p_buffer1, p_buffer2 );
  else
    printf( "¥"%ls¥" is greater than ¥"%ls¥"¥n", p_buffer1, p_buffer2 );
}
/****************  Output should be similar to: ******************

Comparison of each character
  wcscmp: "abcdefg" is less than "abcfg"

Comparison of only the first 3 characters
  wcsncmp: "abcdefg" is identical to "abcfg"
*/
```

関連情報

- 376 ページの『strcmp() ― ストリングの比較』
- 379 ページの『strcoll() ― ストリングの比較』
- 395 ページの『strncmp() ― ストリングの比較』
- 474 ページの『wcscmp() ― ワイド文字ストリングの比較』
- 484 ページの『wcsncat() ― ワイド文字ストリングの連結』
- 『wcsncpy() ― ワイド文字ストリングのコピー』
- 20 ページの『<wchar.h>』

wcsncpy() ― ワイド文字ストリングのコピー

フォーマット

```
#include <wchar.h>
wchar_t *wcsncpy(wchar_t *string1, const wchar_t *string2, size_t count);
```

言語レベル: XPG4

スレッド・セーフ: はい

ワイド文字関数: 詳細については、 553 ページの『ワイド文字』 を参照してください。

説明

`wcsncpy()` 関数は、最大 *count* までのワイド文字を、*string2* から *string1* にコピーします。*string2* が *count* 文字より短い場合には、*string1* は wchar_t ヌル文字を使用して *count* 文字まで引き伸ばされます。

`wcsncpy()` 関数は、ヌル終了ワイド文字ストリング上で作動します。この関数のストリング引数には、ストリングの終わりを示す wchar_t ヌル文字が入っていなければなりません。ヌル文字を含む必要があるのは *string2* のみです。

戻り値

`wcsncpy()` は *string1* へのポインターを戻します。

関連情報

- 380 ページの『strcpy() ― ストリングのコピー』
- 397 ページの『strncpy() ― ストリングのコピー』
- 476 ページの『wcscpy() ― ワイド文字ストリングのコピー』
- 484 ページの『wcsncat() ― ワイド文字ストリングの連結』
- 485 ページの『wcsncmp() ― ワイド文字ストリングの比較』
- 20 ページの『<wchar.h>』

__wcsnicmp() ― 大/小文字の区別をしないワイド文字ストリングの比較

フォーマット

```
#include <wchar.h>;
int __wcsnicmp(const wchar_t *string1, const wchar_t *string2, size_t count);
```

言語レベル: Extension

スレッド・セーフ: はい。

ロケール依存: コンパイル・コマンドで LOCALETYPE(*LOCALE) が指定される場合、この関数の振る舞いは、現行ロケールの LC_CTYPE カテゴリーの影響を受ける可能性があります。また、コンパイル・コマンドで LOCALETYPE(*LOCALEUCS2) または LOCALETYPE(*LOCALEUTF) が指定される場合、この関数の振る舞いは、現行ロケールの LC_UNI_CTYPE カテゴリーの影響も受ける可能性があります。この関数は、コンパイル・コマンドに対して LOCALETYPE(*CLD) が指定されている場合には使用できません。詳細については、 549 ページの『CCSID およびロケールの理解』を参照してください。

ワイド文字関数: 詳細については、 553 ページの『ワイド文字』 を参照してください。

説明

`__wcsnicmp()` 関数は、*string1* と *string2* の最大 count 文字数を大/小文字の区別なしで比較します。*string1* と *string2* のすべての英字ワイド文字は、比較の前に小文字に変換されます。

`__wcsnicmp()` 関数は、ヌル終了ワイド文字ストリング上で作動します。この関数のストリング引数には、ストリングの終わりを示す wchar_t ヌル文字 `(L'¥0')` が含まれていなければなりません。

戻り値

__wcsnicmp() 関数は、2 つのストリング間の関係を示す次のような値を戻します。

表 11. __wcsicmp() の戻り値

値	意味
0 より小さい値	*string1* は *string2* より小さい
0	*string1* は *string2* と等しい
0 より大きい値	*string1* は *string2* より大きい

.

__wcsnicmp() の使用例

この例では __wcsnicmp() を使用して、2 つのワイド文字ストリングを比較します。

```
#include <stdio.h>
#include <wchar.h>

int main(void)
{
  wchar_t *str1 = L"STRING ONE";
  wchar_t *str2 = L"string TWO";
  int result;

  result = __wcsnicmp(str1, str2, 6);

  if (result == 0)
    printf("Strings compared equal.¥n");
  else if (result < 0)
    printf("¥"%ls¥" is less than ¥"%ls¥".¥n", str1, str2);
  else
    printf("¥"%ls¥" is greater than ¥"%ls¥".¥n", str1, str2);

  return 0;
}
/********  The output should be similar to: ***************

Strings compared equal.

***********************************/
```

関連情報

- 376 ページの『strcmp() ― ストリングの比較』
- 395 ページの『strncmp() ― ストリングの比較』
- 471 ページの『wcscat() ― ワイド文字ストリングの連結』
- 472 ページの『wcschr() ― ワイド文字の検索』
- 477 ページの『wcscspn() ― 最初に一致したワイド文字のオフセットの検索』
- 482 ページの『wcslen() ― ワイド文字ストリング長の計算』
- 485 ページの『wcsncmp() ― ワイド文字ストリングの比較』
- 480 ページの『__wcsicmp() ― 大/小文字の区別をしないワイド文字ストリングの比較』
- 20 ページの『<wchar.h>』

wcspbrk() ― ストリング内のワイド文字の位置検出

フォーマット

```
#include <wchar.h>
wchar_t *wcspbrk(const wchar_t *string1, const wchar_t *string2);
```

言語レベル: XPG4

スレッド・セーフ: はい。

ワイド文字関数: 詳細については、 553 ページの『ワイド文字』 を参照してください。

説明

wcspbrk() 関数は、*string2* が示すストリングの任意のワイド文字が、*string1* が示すストリングで最初に現れる位置を見つけます。

戻り値

wcspbrk() 関数は、その文字へのポインターを戻します。*string1* と *string2* に共通のワイド文字がない場合には、wcspbrk() 関数は NULL を戻します。

wcspbrk() の使用例

この例では wcspbrk() を使用して、配列 *string* で ″a″ または ″b″ のいずれかが最初に現れる位置を検索します。

```
#include <stdio.h>
#include <wchar.h>

int main(void)
{
  wchar_t * result;
  wchar_t * string = L"The Blue Danube";
  wchar_t *chars = L"ab";

  result = wcspbrk( string, chars);
   printf("The first occurrence of any of the characters ¥"%ls¥" in "
          "¥"%ls¥" is ¥"%ls¥"¥n", chars, string, result);

}

/****************  Output should be similar to:  ******************

The first occurrence of any of the characters "ab" in "The Blue Danube"
is "anube"
******************************************************************/
```

関連情報

- 375 ページの『strchr() ― 文字の検索』
- 381 ページの『strcspn() ― 最初に一致した文字のオフセットの検索』
- 401 ページの『strpbrk() ― ストリング内の文字の検索』
- 407 ページの『strspn() ―最初の不一致文字のオフセットの検索』
- 472 ページの『wcschr() ― ワイド文字の検索』
- 474 ページの『wcscmp() ― ワイド文字ストリングの比較』

- 477 ページの『wcscspn() — 最初に一致したワイド文字のオフセットの検索』
- 485 ページの『wcsncmp() — ワイド文字ストリングの比較』
- 493 ページの『wcsrchr() — ストリング内でワイド文字が最後に現れる位置の検出』
- 510 ページの『wcswcs() — ワイド文字サブストリングの位置検出』
- 20 ページの『<wchar.h>』

wcsptime()— ワイド文字ストリングから日付/時刻への変換

フォーマット

```
#include <wchar.h>
wchar_t *wcsptime(const wchar_t *buf, const wchar_t *format, struct tm *tm);
```

言語レベル: Extended

スレッド・セーフ: はい。

ロケール依存: この関数の振る舞いは、現行ロケールの LC_UNI_CTYPE、LC_UNI_TIME、および LC_UNI_TOD カテゴリーの影響を受ける可能性があります。この関数は、コンパイル・コマンドで LOCALETYPE(*LOCALEUTF) が指定される場合のみ使用可能です。詳細については、 549 ページの『CCSID およびロケールの理解』を参照してください。

ワイド文字関数: 詳細については、 553 ページの『ワイド文字』 を参照してください。

説明

`wcsptime()` 関数は、*format* で指定された形式を使用し、 *buf* により示されるワイド文字ストリングを、*tm* により示される *tm* 構造体に保管される値に変換します。この関数は、`strptime()` と同等ですが、ワイド文字を使用する点が異なります。

書式ストリングの説明については、 402 ページの『strptime()— ストリングから日付/時刻への変換』を参照してください。

戻り値

`wcsptime()` 関数が正常終了すると、解析した最終ワイド文字の後に続く文字へのポインターを戻します。それ以外の場合は、NULL ポインターが戻されます。 `errno` の値は **ECONVERT** (変換エラー) に設定される可能性があります。

`wcsptime()` の使用例

```
#include <stdio.h>
#include <time.h>
#include <wchar.h>

int main(void)
{
    wchar_t buf[100];
    time_t t;
    struct tm *timeptr,result;

    t = time(NULL);
    timeptr = localtime(&t);
    wcsftime(buf, 100, L"%a %m/%d/%Y %r", timeptr);

    if (wcsptime(buf, L"%a %m/%d/%Y %r", &result) == NULL)
```

```
        printf("¥nwcsptime failed¥n");
   else
   {
        printf("tm_hour:  %d¥n",result.tm_hour);
        printf("tm_min:  %d¥n",result.tm_min);
        printf("tm_sec:  %d¥n",result.tm_sec);
        printf("tm_mon:  %d¥n",result.tm_mon);
        printf("tm_mday:  %d¥n",result.tm_mday);
        printf("tm_year:  %d¥n",result.tm_year);
        printf("tm_yday:  %d¥n",result.tm_yday);
        printf("tm_wday:  %d¥n",result.tm_wday);
   }

   return 0;
}

/************************************************************
     The output should be similar to:

      tm_hour:  14
    tm_min:  25
    tm_sec:  34
    tm_mon:  7
    tm_mday:  19
    tm_year:  103
    tm_yday:  230
    tm_wday:  2
************************************************************/
```

関連情報

- 41 ページの『asctime() ― 時間から文字ストリングへの変換』
- 43 ページの『asctime_r() ― 時間から文字ストリングへの変換 (再始動可能)』
- 75 ページの『ctime() ― 時間から文字ストリングへの変換』
- 77 ページの『ctime64() ― 時間から文字ストリングへの変換』
- 80 ページの『ctime64_r() ― 時間から文字ストリングへの変換 (再始動可能)』
- 78 ページの『ctime_r() ― 時間から文字ストリングへの変換 (再始動可能)』
- 167 ページの『gmtime() ― 時間の変換』
- 169 ページの『gmtime64() ― 時間の変換』
- 173 ページの『gmtime64_r() ― 時間の変換 (再始動可能)』
- 171 ページの『gmtime_r() ― 時間の変換 (再始動可能)』
- 192 ページの『localtime() ― 時間の変換』
- 194 ページの『localtime64() ― 時間の変換』
- 197 ページの『localtime64_r() ― 時間の変換 (再始動可能)』
- 195 ページの『localtime_r() ― 時間の変換 (再始動可能)』
- 354 ページの『setlocale() ― ロケールの設定』
- 387 ページの『strftime() ― 日付/時刻からストリングへの変換』
- 402 ページの『strptime()― ストリングから日付/時刻への変換』
- 429 ページの『time() ― 現在時刻の判別』
- 430 ページの『time64() ― 現在時刻の判別』
- 19 ページの『<time.h>』

wcsrchr() — ストリング内でワイド文字が最後に現れる位置の検出

フォーマット

```
#include <wchar.h>
wchar_t *wcsrchr(const wchar_t *string, wchar_t character);
```

言語レベル: ANSI

スレッド・セーフ: はい

ワイド文字関数: 詳細については、 553 ページの『ワイド文字』 を参照してください。

説明

`wcsrchr()` 関数は、*string* が指すストリング内の *character* が最後に現れる位置を検索します。終了ヌル文字 wchar_t は、ストリングの一部と見なされます。

戻り値

`wcsrchr()` 関数は、その文字へのポインターを戻します。ストリング内で *character* が発生しなかった場合は、NULL ポインターを戻します。

`wcsrchr()` の使用例

この例では、`wcschr()` と `wcsrchr()` の使用法を比較します。ストリングを調べて、ワイド文字ストリングで p が最初に現れる位置とが最後に現れる位置を検索します。

```
#include <stdio.h>
#include <wchar.h>

#define SIZE 40

int main(void)
{
  wchar_t buf[SIZE] = L"computer program";
  wchar_t * ptr;
  int    ch = 'p';

  /* This illustrates wcschr */
  ptr = wcschr( buf, ch );
  printf( "The first occurrence of %c in '%ls' is '%ls'¥n", ch, buf, ptr );

  /* This illustrates wcsrchr */
  ptr = wcsrchr( buf, ch );
  printf( "The last occurrence of %c in '%ls' is '%ls'¥n",  ch, buf, ptr );
}

/****************  Output should be similar to:  *****************

The first occurrence of p in 'computer program' is 'puter program'
The last occurrence of p in 'computer program' is 'program'
*/
```

関連情報

- 375 ページの『strchr() — 文字の検索』
- 406 ページの『strrchr() — ストリング内で文字が最後に現れる位置の検出』
- 381 ページの『strcspn() — 最初に一致した文字のオフセットの検索』

- 407 ページの『strspn() ―最初の不一致文字のオフセットの検索』
- 472 ページの『wcschr() ― ワイド文字の検索』
- 474 ページの『wcscmp() ― ワイド文字ストリングの比較』
- 477 ページの『wcscspn() ― 最初に一致したワイド文字のオフセットの検索』
- 485 ページの『wcsncmp() ― ワイド文字ストリングの比較』
- 510 ページの『wcswcs() ― ワイド文字サブストリングの位置検出』
- 490 ページの『wcspbrk() ― ストリング内のワイド文字の位置検出』
- 20 ページの『<wchar.h>』

wcsrtombs() ― ワイド文字ストリングからマルチバイト・ストリングへの変換 (再開可能)

フォーマット

```
#include <wchar.h>
size_t wcsrtombs (char *dst, const wchar_t **src, size_t len,
                  mbstate_t  *ps);
```

言語レベル: ANSI

スレッド・セーフ: 4 番目のパラメーター *ps* が NULL ではない場合は対応しています。

ロケール依存: この関数の振る舞いは、現行ロケールの LC_CTYPE カテゴリーの影響を受ける可能性があります。また、この振る舞いは、LOCALETYPE(*LOCALEUCS2) または LOCALETYPE(*LOCALEUTF) がコンパイル・コマンドに対して指定されている場合は、現行ロケールの LC_UNI_CTYPE カテゴリーの影響を受ける可能性もあります。この関数は、コンパイル・コマンドに対して LOCALETYPE(*CLD) が指定されている場合には使用できません。詳細については、 549 ページの『CCSID およびロケールの理解』を参照してください。

ワイド文字関数: 詳細については、 553 ページの『ワイド文字』 を参照してください。

説明

この関数は、`wcstombs()` の再始動可能バージョンです。

`wcsrtombs()` 関数は、*src* が間接的に指す配列にあるワイド文字列を、対応するマルチバイト文字列 (*ps* が記述するシフト状態で開始されるもの) に変換します。変換された文字列は、*dst* が指す配列に保管されます (*dst* が NULL ポインターでない場合)。変換は終了ヌル・ワイド文字 (これを含む) まで続行し、保管されます。変換中、有効なマルチバイト文字に対応しないコードに達したとき、または (*dst* が NULL ポインターでない場合に) *dst* によって指される配列に、次にマルチバイト・エレメントが保管されると、合計バイト *len* の限界を超えてしまうときは、変換が早い段階で停止してしまいます。それぞれの変換は、`wcrtomb()` の呼び出しであるかのように行われます。

dst が NULL ポインターでない場合には、 *src* が指すオブジェクトは (終了ヌル文字に達したので変換が終了した場合に) NULL ポインターが割り当てられるか、最後のワイド文字が変換された直後にコードのアドレスが割り当てられます。終了ヌル・ワイド文字に達したので変換が停止した場合には、記述されている結果の状態は初期変換状態になります。

戻り値

最初のコードが有効なワイド文字でない場合、エンコード・エラーが発生します。 wcsrtombs() は、マクロ EILSEQ の値を errno に保管し、 (size_t) -1 を戻しますが、変換状態は未変更のままになります。そうでない場合、結果のマルチバイト文字シーケンスでバイト文字を戻します。これは、*dst* が NULL ポインターでないときに変更された配列エレメント数と同じです。

変換エラーが発生した場合、errno は **ECONVERT** に設定される可能性があります。

wcsrtombs() の使用例

```
#include <stdio.h>
#include <wchar.h>
#include <string.h>

#define SIZE 20

int main(void)
{
   char     dest[SIZE];
   wchar_t *wcs = L"string";
   wchar_t *ptr;
   size_t   count = SIZE;
   size_t   length;
   mbstate_t ps = 0;

   ptr = (wchar_t *) wcs;
   length = wcsrtombs(dest, ptr, count, &ps);
   printf("%d characters were converted.¥n", length);
   printf("The converted string is ¥"%s¥"¥n¥n", dest);

   /* Reset the destination buffer */
   memset(dest, '¥0', sizeof(dest));

   /* Now convert only 3 characters */
   ptr = (wchar_t *) wcs;
   length = wcsrtombs(dest, ptr, 3, &ps);
   printf("%d characters were converted.¥n", length);
   printf("The converted string is ¥"%s¥"¥n¥n", dest);
}

/*****************  Output should be similar to:  **********************
6 characters were converted.
The converted string is "string"

3 characters were converted.
The converted string is "str"
*/
```

関連情報

- 205 ページの『mblen() — マルチバイト文字の長さの計算』
- 207 ページの『mbrlen() — マルチバイト文字の長さの計算 (再始動可能)』
- 210 ページの『mbrtowc() — マルチバイト文字からワイド文字への変換 (再始動可能)』
- 214 ページの『mbsrtowcs() — マルチバイト・ストリングからワイド文字ストリングへの変換 (再始動可能)』
- 467 ページの『wcrtomb() — ワイド文字からマルチバイト文字への変換 (再開可能)』
- 505 ページの『wcstombs() — ワイド文字ストリングからマルチバイト・ストリングへの変換』
- 20 ページの『<wchar.h>』

wcsspn() ― 最初の不一致ワイド文字のオフセットの検索

フォーマット

```
#include <wchar.h>
size_t wcsspn(const wchar_t *string1, const wchar_t *string2);
```

言語レベル: ANSI

スレッド・セーフ: はい

ワイド文字関数: 詳細については、 553 ページの『ワイド文字』 を参照してください。

説明

wcsspn() 関数は、*string1* が指すストリングの初期セグメントにある、すべて *string2* が指すストリングからのワイド文字で構成されるワイド文字数を計算します。

戻り値

wcsspn() 関数は、ワイド文字数をセグメントに戻します。

wcsspn() の使用例

この例は、配列 string で、a、b、または c 以外のワイド文字が最初に現れる位置を検索します。この例のストリングは cabbage であるため、wcsspn() 関数は 5 (a、b、または c 以外の文字の前の cabbage のセグメントの指標) を戻します。

```
#include <stdio.h>
#include <wchar.h>

int main(void)
{
  wchar_t * string = L"cabbage";
  wchar_t * source = L"abc";
  int index;

  index = wcsspn( string, L"abc" );
  printf( "The first %d characters of ¥"%ls¥" are found in ¥"%ls¥"¥n",
              index, string, source );
}

/****************  Output should be similar to:  ******************

The first 5 characters of "cabbage" are found in "abc"
*/
```

関連情報

- 375 ページの『strchr() ― 文字の検索』
- 381 ページの『strcspn() ― 最初に一致した文字のオフセットの検索』
- 401 ページの『strpbrk() ― ストリング内の文字の検索』
- 406 ページの『strrchr() ― ストリング内で文字が最後に現れる位置の検出』
- 407 ページの『strspn() ―最初の不一致文字のオフセットの検索』
- 471 ページの『wcscat() ― ワイド文字ストリングの連結』
- 472 ページの『wcschr() ― ワイド文字の検索』
- 474 ページの『wcscmp() ― ワイド文字ストリングの比較』

- 477 ページの『wcscspn() — 最初に一致したワイド文字のオフセットの検索』
- 485 ページの『wcsncmp() — ワイド文字ストリングの比較』
- 490 ページの『wcspbrk() — ストリング内のワイド文字の位置検出』
- 493 ページの『wcsrchr() — ストリング内でワイド文字が最後に現れる位置の検出』
- 496 ページの『wcsspn() — 最初の不一致ワイド文字のオフセットの検索』
- 510 ページの『wcswcs() — ワイド文字サブストリングの位置検出』
- 20 ページの『<wchar.h>』

wcsstr() — ワイド文字サブストリングの位置検出

フォーマット

```
#include <wchar.h>
wchar_t *wcsstr(const wchar_t *wcs1, const wchar_t *wcs2);
```

言語レベル: ANSI

スレッド・セーフ: はい。

ワイド文字関数: 詳細については、 553 ページの『ワイド文字』 を参照してください。

説明

wcsstr() 関数は、*wcs1* で *wcs2* が最初に現れる位置を見つけます。

戻り値

wcsstr() 関数は、*wcs1* 内での *wcs2* が最初に現れる位置の先頭へのポインターを戻します。 *wcs2* が *wcs1* に現れないと、wcsstr() 関数は NULL を戻します。*wcs2* がゼロ長のワイド文字ストリングを指す場合には、*wcs1* を戻します。

wcsstr() の使用例

この例では wcsstr() 関数を使用して、ワイド文字ストリング ″needle in a haystack″ で ″hay″ が最初に現れる位置を検索します。

```
#include <stdio.h>
#include <wchar.h>

int main(void)
{
   wchar_t *wcs1 = L"needle in a haystack";
   wchar_t *wcs2 = L"hay";

   printf("result: ¥"%ls¥"¥n", wcsstr(wcs1, wcs2));
   return 0;

   /**********************************************
      The output should be similar to:

      result: "haystack"
   **********************************************/
}
```

関連情報

- 408 ページの『strstr() ― サブストリングの位置検出』
- 472 ページの『wcschr() ― ワイド文字の検索』
- 493 ページの『wcsrchr() ― ストリング内でワイド文字が最後に現れる位置の検出』
- 510 ページの『wcswcs() ― ワイド文字サブストリングの位置検出』
- 20 ページの『<wchar.h>』

wcstod() ― ワイド文字ストリングから double 型への変換

フォーマット

```
#include <wchar.h>
double wcstod(const wchar_t *nptr, wchar_t **endptr);
```

言語レベル: XPG4

スレッド・セーフ: はい。

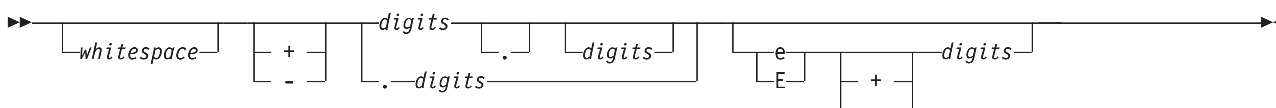
ロケール依存: コンパイル・コマンドで LOCALETYPE(*LOCALE) が指定される場合、この関数の振る舞いは、現行ロケールの LC_CTYPE カテゴリーおよび LC_NUMERIC カテゴリーの影響を受ける可能性があります。また、コンパイル・コマンドで LOCALETYPE(*LOCALEUCS2) または LOCALETYPE (*LOCALEUTF) が指定されている場合、この関数の振る舞いは、現行ロケールの LC_UNI_CTYPE カテゴリーおよび LC_UNI_NUMERIC カテゴリーの影響も受ける可能性があります。この関数は、コンパイル・コマンドに対して LOCALETYPE(*CLD) が指定されている場合には使用できません。詳細については、549 ページの『CCSID およびロケールの理解』を参照してください。

ワイド文字関数: 詳細については、 553 ページの『ワイド文字』 を参照してください。

説明

`wcstod()` 関数は、 *nptr* が指すワイド文字ストリングの先頭部分を double 型値に変換します。*nptr* パラメーターは、数値バイナリー浮動小数点値として解釈できる文字のシーケンスを指します。 `wcstod()` 関数によって、数値の一部として認識できないストリングは、先頭文字のところで読み取りが停止されます。この文字は、ストリングの最後にある wchar_t ヌル文字である場合があります。

`wcstod()` 関数では、次の形式のストリングを *nptr* が指していなければなりません。



最初の文字がこの形式に適合しない場合、走査は停止されます。さらに、INFINITY または NAN のシーケンス (大/小文字を区別しない) が許可されます。

戻り値

`wcstod()` 関数は、変換された double 値を戻します。変換が実行できない場合、 `wcstod()` 関数は 0 を戻します。正しい値が表示可能な値の範囲外にある場合は、 `wcstod()` 関数は +HUGE_VAL または -HUGE_VAL を値の符号に応じて戻し、 errno を ERANGE に設定します。正しい値がアンダーフローを引き起こした場合には、`wcstod()` 関数は 0 を戻し、errno は ERANGE に設定されます。*nptr* が指すスト

リングが空であるか、想定される形式になっていない場合には、変換は行われません。*nptr* の値は、*endptr* が NULL ポインターでない場合に *endptr* が指すオブジェクトに保管されます。

数字以外の文字が、指数として読み取られる E または e に続く場合、 wcstod() 関数は失敗しません。例えば、100elf は浮動小数点値 100.0 に変換されます。

errno の値は **ERANGE** (範囲エラー) に設定される可能性があります。

INFINITY (大/小文字を区別しない) の文字シーケンスは、INFINITY の値をもたらします。NAN の文字値は、Quiet Not-A-Number (NAN) 値をもたらします。

wcstod() の使用例

この例では wcstod() 関数を使用して、ストリング wcs をバイナリー浮動小数点値に変換します。

```
#include <stdio.h>
#include <wchar.h>

int main(void)
{
   wchar_t *wcs = L"3.1415926This stopped it";
   wchar_t *stopwcs;

   printf("wcs = ¥"%ls¥"¥n", wcs);
   printf("   wcstod = %f¥n", wcstod(wcs, &stopwcs));
   printf("   Stop scanning at ¥"%ls¥"¥n", stopwcs);
   return 0;

   /**************************************************
      The output should be similar to:

      wcs = "3.1415926This stopped it"
         wcstod = 3.141593
         Stop scanning at "This stopped it"
   **************************************************/
}
```

関連情報

- 409 ページの『strtod() ― strtof() ― strtold ― 文字ストリングから double、浮動、および long double への変換』
- 412 ページの『strtod32() ― strtod64() ― strtod128() ― 文字ストリングから 10 進浮動小数点への変換』
- 418 ページの『strtol() ― strtoll() ― 文字ストリングから long 型および long long 型整数への変換』
- 『wcstod32() ― wcstod64() ― wcstod128()―ワイド文字ストリングから 10 進浮動小数点への変換』
- 503 ページの『wcstol() ― wcstoll() ― ワイド文字ストリングから long 型および long long 型整数への変換』
- 508 ページの『wcstoul() ― wcstoull() ― ワイド文字ストリングから符号なし long 型整数および符号なし long long 型整数への変換』
- 20 ページの『<wchar.h>』

wcstod32() ― wcstod64() ― wcstod128()―ワイド文字ストリングから 10 進浮動小数点への変換

フォーマット

```
#include <wchar.h>
_Decimal32 wcstod32(const wchar_t *nptr, wchar_t **endptr);
_Decimal64 wcstod64(const wchar_t *nptr, wchar_t **endptr);
_Decimal128 wcstod128(const wchar_t *nptr, wchar_t **endptr);
```

言語レベル: XPG4

スレッド・セーフ: はい。

ロケール依存: コンパイル・コマンドで LOCALETYPE(*LOCALE) が指定される場合、この関数の振る舞いは、現行ロケールの LC_CTYPE カテゴリーおよび LC_NUMERIC カテゴリーの影響を受ける可能性があります。また、コンパイル・コマンドで LOCALETYPE(*LOCALEUCS2) または LOCALETYPE(*LOCALEUTF) が指定される場合、この関数の振る舞いは、現行ロケールの LC_UNI_CTYPE カテゴリーおよび LC_UNI_NUMERIC カテゴリーの影響も受ける可能性があります。これらの関数は、コンパイル・コマンドで LOCALETYPE(*CLD) が指定される場合には使用できません。詳細については、 549 ページの『CCSID およびロケールの理解』を参照してください。

ワイド文字関数: 詳細については、 553 ページの『ワイド文字』 を参照してください。

説明

wcstod32()、wcstod64()、および wcstod128() 関数は、*nptr* が指すワイド文字ストリングの先頭部分を単精度、倍精度、または 4 倍精度の 10 進浮動小数点値へ変換します。パラメーター *nptr* は、数値 10 進浮動小数点値として解釈できる文字のシーケンスを示します。wcstod32()、wcstod64()、および wcstod128() 関数によって、数値の一部として認識できないストリングは、先頭文字のところで読み取りが停止されます。この文字は、ストリングの最後にある wchar_t ヌル文字である場合があります。 *endptr* パラメーターは、 *endptr* が NULL ポインターでない場合、この文字を指すように更新されます。

wcstod32()、wcstod64()、および wcstod128() 関数は、*nptr* が以下の形式のストリングを指すことを想定します。

最初の文字がこの形式に適合しない場合、走査は停止されます。さらに、INFINITY または NAN のシーケンス (大/小文字を区別しない) が許可されます。

戻り値

wcstod32()、wcstod64()、および wcstod128() 関数は、表現がアンダーフローまたはオーバーフローを引き起こす場合を除き、浮動小数点数の値を戻します。オーバーフローの場合、wcstod32() は HUGE_VAL_D32 または -HUGE_VAL_D32 を、wcstod64() は HUGE_VAL_D64 または -HUGE_VAL_D64 を、wcstod128() は HUGE_VAL_D128 または -HUGE_VAL_D128 をそれぞれ戻します。アンダーフローの場合、すべての関数は +0.E0 を戻します。

オーバーフローの場合もアンダーフローの場合も、errno は ERANGE に設定されます。*nptr* が指すストリングが想定する形式ではない場合、 +0.E0 の値が戻されます。*nptr* の値は、 *endptr* が指すオブジェクトに保管されます (但し、*endptr* が NULL ポインターでない場合に限られます)。

数字以外の文字が、指数として読み取られる E または e に続く場合、wcstod32()、wcstod64()、および wcstod128() 関数は失敗しません。例えば、100elf は浮動小数点値 100.0 に変換されます。

INFINITY (大/小文字を区別しない) の文字シーケンスは、INFINITY の値をもたらします。NAN の文字値 (大/小文字を区別しない) は Quiet Not-A-Number (NaN) 値をもたらします。

戻り値は必要に応じて、丸めモード Round to Nearest、Ties to Even を使用して丸められます。

wcstod32()、wcstod64()、および wcstod128() の使用例

この例では、ストリング wcs を単精度、倍精度、および 4 倍精度の 10 進浮動小数点値を変換します。

```
#include <stdio.h>
#include <wchar.h>

int main(void)
{
    wchar_t *wcs = L"3.1415926This stopped it";
    wchar_t *stopwcs;

    printf("wcs = ¥"%ls¥"¥n", wcs);
   printf("wcstod32 = %Hf¥n", wcstod32(wcs, &stopwcs));
   printf(" Stopped scan at ¥"%ls¥"¥n", stopwcs);
   printf("wcs = ¥"%ls¥"¥n", wcs);
   printf("wcstod64 = %Df¥n", wcstod64(wcs, &stopwcs));
   printf(" Stopped scan at ¥"%ls¥"¥n", stopwcs);
   printf("wcs = ¥"%ls¥"¥n", wcs);
   printf("wcstod128 = %DDf¥n", wcstod128(wcs, &stopwcs));
   printf(" Stopped scan at ¥"%ls¥"¥n", stopwcs);
}

/***************** Output should be similar to: *****************

wcs = "3.1415926This stopped it"
 wcstod = 3.141593
 Stopped scan at "This stopped it"
wcs = "3.1415926This stopped it"
 wcstod = 3.141593
 Stopped scan at "This stopped it"
wcs = "3.1415926This stopped it"
 wcstod = 3.141593
 Stopped scan at "This stopped it"

*/
```

関連情報

- 409 ページの『strtod() — strtof() — strtold — 文字ストリングから double、浮動、および long double への変換』
- 412 ページの『strtod32() — strtod64() — strtod128() — 文字ストリングから 10 進浮動小数点への変換』
- 418 ページの『strtol() — strtoll() — 文字ストリングから long 型および long long 型整数への変換』
- 498 ページの『wcstod() — ワイド文字ストリングから double 型への変換』
- 503 ページの『wcstol() — wcstoll() — ワイド文字ストリングから long 型および long long 型整数への変換』
- 508 ページの『wcstoul() — wcstoull() — ワイド文字ストリングから符号なし long 型整数および符号なし long long 型整数への変換』
- 20 ページの『<wchar.h>』

wcstok() ― ワイド文字ストリングのトークン化

フォーマット

```
#include <wchar.h>
wchar_t *wcstok(wchar_t *wcs1, const wchar_t *wcs2, wchar_t **ptr);
```

言語レベル: ANSI

スレッド・セーフ: はい。

ワイド文字関数: 詳細については、 553 ページの『ワイド文字』を参照してください。

説明

wcstok() 関数は、*wcs1* をゼロ個以上のトークンのシリーズとして読み取り、 *wcs2* を *wcs1* 内のトークンの区切り文字として役立つワイド文字のセット読み取ります。 wcstok() 関数の呼び出しのシーケンスにより、 *wcs1* 内のトークンが見つかります。トークンは、*wcs2* からの 1 つ以上の区切り文字によって分離することができます。3 番目の引数は、 wcstok() 関数が同じストリングの走査を実行するために必要な情報を保管するユーザー提供のワイド文字ポインターを指します。

wcstok() 関数は、ワイド文字ストリング *wcs1* について最初に呼び出されると、先行区切り文字をスキップして、*wcs1* 内の最初のトークンを検索します。wcstok() 関数は、最初のトークンへのポインターを戻します。*wcs1* からの次のトークンを読み取るには、最初のパラメーターとして NULL で wcstok() 関数を呼び出します (*wcs1*)。この NULL パラメーターにより、wcstok() 関数は前のトークン・ストリング内で次のトークンを検索します。それぞれの区切り文字は、トークンを終了するために、ヌル文字で置き換えられます。

wcstok() 関数は常にポインター *ptr* に十分な情報を保管するので、以降の呼び出しは NULL を最初のパラメーターとし、未変更のポインター値を 3 番目のパラメーターとして、前に戻されたトークンの直後で検索を開始します。区切り文字のセット (*wcs2*) は呼び出しごとに変更できます。

戻り値

wcstok() 関数は、トークンの最初のワイド文字へのポインターを戻すか、トークンがない場合には NULL ポインターを戻します。同じトークン・ストリングでの以後の呼び出しで、wcstok() 関数は、ストリング内の次のトークンへのポインターを戻します。トークンがそれ以上ないときには、wcstok() 関数は NULL を戻します。

wcstok() の使用例

この例では wcstok() 関数を使用して、ワイド文字ストリング *str1* 内でトークンを見つけます。

```
#include <stdio.h>
#include <wchar.h>

int main(void)
{
   static wchar_t str1[] = L"?a??b,,,#c";
   static wchar_t str2[] = L"¥t ¥t";
   wchar_t *t, *ptr1, *ptr2;

   t = wcstok(str1, L"?", &ptr1);   /* t points to the token L"a" */
   printf("t = '%ls'¥n", t);
   t = wcstok(NULL, L",", &ptr1);   /* t points to the token L"?b"*/
   printf("t = '%ls'¥n", t);
   t = wcstok(str2, L" ¥t,", &ptr2); /* t is a null pointer        */
```

```
   printf("t = '%ls'¥n", t);
   t = wcstok(NULL, L"#,", &ptr1);  /* t points to the token L"c" */
   printf("t = '%ls'¥n", t);
   t = wcstok(NULL, L"?", &ptr1);     /* t is a null pointer          */
   printf("t = '%ls'¥n", t);
   return 0;

   /********************************************************************
      The output should be similar to:

          t = 'a'
          t = '?b'
          t = ''
          t = 'c'
          t = ''
   ********************************************************************/
}
```

関連情報

- 415 ページの『strtok() ― ストリングのトークン化』
- 20 ページの『<wchar.h>』

wcstol() ― wcstoll() ― ワイド文字ストリングから long 型および long long 型整数への変換

フォーマット (wcstol())

```
#include <wchar.h>
long int wcstol(const wchar_t *nptr, wchar_t **endptr, int base);
```

フォーマット (wcstoll())

```
#include <wchar.h>
long long int wcstoll(const wchar_t *nptr, wchar_t **endptr, int base);
```

言語レベル: ANSI

スレッド・セーフ: はい。

ロケール依存: コンパイル・コマンドで LOCALETYPE(*LOCALE) が指定される場合、これらの関数の振る舞いは、現行ロケールの LC_CTYPE カテゴリーの影響を受ける可能性があります。また、コンパイル・コマンドで LOCALETYPE(*LOCALEUCS2) または LOCALETYPE(*LOCALEUTF) が指定される場合、これらの関数の振る舞いは、現行ロケールの LC_UNI_CTYPE カテゴリーの影響も受ける可能性があります。これらの関数は、コンパイル・コマンドで LOCALETYPE(*CLD) が指定される場合には使用できません。詳細については、 549 ページの『CCSID およびロケールの理解』を参照してください。

ワイド文字関数: 詳細については、 553 ページの『ワイド文字』 を参照してください。

説明

`wcstol()` 関数は、 *nptr* が指すワイド文字ストリングの先頭部分を long 型整数値に変換します。 *nptr* パラメーターにより、long 型整数の数値として解釈できるワイド文字のシーケンスが示されます。`wcstol()` 関数は、数値として認識できないワイド文字に遭遇すると、その時点でストリングの読み取りが停止されます。この文字は、ストリングの最後にある wchar_t ヌル文字である場合があります。また、基数以上の数字が最初に登場した場合、その数字が終了文字になる場合もあります。

wcstoll() サブルーチンは、ワイド文字ストリングを long long 型整数に変換します。ワイド文字ストリングは、iswspace サブルーチンによって判別された最初のスペース文字をスキップするように解析されます。スペース以外の文字の場合には、サブジェクト・ストリングが開始されます。これによって、*base* パラメーターで指定された基数で、long long 型整数が形成されます。サブジェクト・シーケンスは、想定した形式の long long 型整数である最長の初期サブストリングになるように定義されています。

endptr パラメーターの値がヌルでない場合、走査を終了した文字へのポインターは *endptr* に保管されます。long long 型整数を形成できない場合、*endptr* パラメーターの値は、*nptr* パラメーターの値に設定されます。

base パラメーターの値が 2 と 36 の間の場合には、サブジェクト・シーケンスの想定される形式は、基数が *base* パラメーターで指定される long long 型整数を表す文字と数字のシーケンスです。このシーケンスは、オプションで先頭に正符号 (+) または負符号 (-) が付きます。a (または A) から z (または Z) までの文字は 10 から 35 までの値と見なされます。この値が *base* パラメーターの値より小さい文字だけが許可されます。*base* パラメーターの値が 16 の場合には、文字 0x または 0X はオプションで一連の文字および数字の前に付くことがありますが、正符号 (+) または負符号 (-) があるとその後に続きます。

base パラメーターの値が 0 である場合には、ストリングによって base が決定されます。したがって、最初の符号 (オプション) の後で、0 から始まっている場合は 8 進変換を表し、0x または 0X で始まっている場合は 16 進変換を表すことになります。

戻り値

wcstol() 関数は、変換された long 型整数値を戻します。変換が実行できない場合、 wcstol() 関数は 0 を戻します。正しい値が表示可能な値の範囲外にある場合は、 wcstol() 関数は LONG_MAX または LONG_MIN を値の符号に応じて戻し、 errno を ERANGE に設定します。*nptr* が指すストリングが空であるか、想定される形式になっていない場合には、変換は行われません。*nptr* の値は、 *endptr* が NULL ポインターでない場合に *endptr* が指すオブジェクトに保管されます。

正常終了の場合、wcstoll() サブルーチンは変換された値を戻します。変換が行われない場合には、0 が戻され、 errno グローバル変数はエラーを示すように設定されます。正しい値が表現可能である範囲外にある場合、wcstoll() サブルーチンは LONG_LONG_MAX または LONG_LONG_MIN の値を戻します。

errno の値は **ERANGE** (範囲エラー) または **EINVAL** (無効な引数) に設定される可能性があります。

wcstol() の使用例

この例では wcstol() 関数を使用して、ワイド文字ストリング *wcs* を long 型整数値に変換します。

```
#include <stdio.h>
#include <wchar.h>

int main(void)
{
   wchar_t *wcs = L"10110134932";
   wchar_t *stopwcs;
   long     l;
   int      base;

   printf("wcs = ¥"%ls¥"¥n", wcs);
   for (base=2; base<=8; base*=2) {
      l = wcstol(wcs, &stopwcs, base);
      printf("   wcstol = %ld¥n"
             "   Stopped scan at ¥"%ls¥"¥n¥n", l, stopwcs);
   }
   return 0;
```

```
/*******************************************************
   The output should be similar to:

   wcs = "10110134932"
      wcstol = 45
      Stopped scan at "34932"

      wcstol = 4423
      Stopped scan at "4932"

      wcstol = 2134108
      Stopped scan at "932"
*******************************************************/
}
```

関連情報

- 409 ページの『strtod() — strtof() — strtold — 文字ストリングから double、浮動、および long double への変換』
- 412 ページの『strtod32() — strtod64() — strtod128() — 文字ストリングから 10 進浮動小数点への変換』
- 418 ページの『strtol() — strtoll() — 文字ストリングから long 型および long long 型整数への変換』
- 420 ページの『strtoul() — strtoull() — 文字ストリングから符号なし long 型整数および符号なし long long 型整数への変換』
- 498 ページの『wcstod() — ワイド文字ストリングから double 型への変換』
- 499 ページの『wcstod32() — wcstod64() — wcstod128()—ワイド文字ストリングから 10 進浮動小数点への変換』
- 508 ページの『wcstoul() — wcstoull() — ワイド文字ストリングから符号なし long 型整数および符号なし long long 型整数への変換』
- 20 ページの『<wchar.h>』

wcstombs() — ワイド文字ストリングからマルチバイト・ストリングへの変換

フォーマット

```
#include <stdlib.h>
size_t wcstombs(char *dest, const wchar_t *string, size_t count);
```

言語レベル: ANSI

スレッド・セーフ: はい。

ロケール依存: この関数の振る舞いは、現行ロケールの LC_CTYPE カテゴリーの影響を受ける可能性があります。また、この振る舞いは、LOCALETYPE(*LOCALEUCS2) または LOCALETYPE(*LOCALEUTF) がコンパイル・コマンドに対して指定されている場合は、現行ロケールの LC_UNI_CTYPE カテゴリーの影響を受ける可能性もあります。詳細については、 549 ページの『CCSID およびロケールの理解』を参照してください。

ワイド文字関数: 詳細については、 553 ページの『ワイド文字』 を参照してください。

説明

wcstombs() 関数は、*string* が指すワイド文字ストリングを *dest* が指すマルチバイト配列に変換します。変換されたストリングは初期シフト状態で始まります。*dest* の *count* バイトがフルになったか、wchar_t ヌル文字が見つかった後で、変換は停止します。

完全なマルチバイト文字のみが *dest* に保管されます。*dest* のスペースの不足により、一部のマルチバイト文字が保管される場合、 wcstombs() は *n* バイトより少ない文字を保管し、無効文字を廃棄します。

戻り値

wcstombs() 関数は、終了ヌル文字を除く、マルチバイト文字ストリングの長さ (バイト単位) を戻します。無効のマルチバイト文字が見つかった場合、値 (size_t)-1 が戻されます。

errno の値は **EILSEQ** (入力文字により変換が停止) または **ECONVERT** (変換) に設定される可能性があります。

wcstombs() の使用例

このプログラムは LOCALETYPE(*LOCALE) および SYSIFCOPT(*IFSIO) でコンパイルされています。

```
#include <stdio.h>
#include <stdlib.h>
#include <locale.h>
#include <wchar.h>

#define  STRLENGTH   10
#define  LOCNAME     "qsys.lib/JA_JP.locale"
#define  LOCNAME_EN  "qsys.lib/EN_US.locale"

int main(void)
{
    char      string[STRLENGTH];
    int length, sl = 0;
    wchar_t   wc2[] = L"ABC";
    wchar_t  wc_string[10];
    mbstate_t ps = 0;
    memset(string, '\0', STRLENGTH);
    wc_string[0] = 0x00C1;
    wc_string[1] = 0x4171;
    wc_string[2] = 0x4172;
    wc_string[3] = 0x00C2;
    wc_string[4] = 0x0000;

    /* In this first example we will convert a wide character string */
    /* to a single byte character string.  We first set the locale   */
    /* to a single byte locale.  We choose a locale with            */
    /* CCSID 37.                                                     */

    if (setlocale(LC_ALL, LOCNAME_EN) == NULL)
        printf("setlocale failed.\n");

    length = wcstombs(string, wc2, 10);

    /* In this case wide characters ABC are converted to */
    /* single byte characters ABC, length is 3.   */

    printf("string = %s, length = %d\n\n", string, length);

    /* Now lets try a multibyte example.  We first must set the */
    /* locale to a multibyte locale.  We choose a locale with     */
    /* CCSID 5026  */

    if (setlocale(LC_ALL, LOCNAME) == NULL)
        printf("setlocale failed.\n");
```

```
    length = wcstombs(string, wc_string, 10);


    /* The hex look at string would now be:                    */
    /* C10E417141720FC2   length will be 8                     */
    /* You would need a device capable of displaying multibyte */
    /* characters to see this string.                          */

    printf("length = %d\n\n", length);

}
/*  The output should look like this:

string = ABC, length = 3

length = 8
                                   */
```

このプログラムは LOCALETYPE(*LOCALEUCS2) および SYSIFCOPT(*IFSIO) でコンパイルされています。

```
#include <stdio.h>
#include <stdlib.h>
#include <locale.h>
#include <wchar.h>

#define  STRLENGTH   10
#define  LOCNAME     "qsys.lib/JA_JP.locale"
#define  LOCNAME_EN  "qsys.lib/EN_US.locale"

int main(void)
{
    char      string[STRLENGTH];
    int length, sl = 0;
    wchar_t   wc2[] = L"ABC";
    wchar_t  wc_string[10];
    mbstate_t ps = 0;
    memset(string, '\0', STRLENGTH);
    wc_string[0] = 0x0041;       /* UNICODE A  */
    wc_string[1] = 0xFF41;
    wc_string[2] = 0xFF42;
    wc_string[3] = 0x0042;       /* UNICODE B  */
    wc_string[4] = 0x0000;
    /* In this first example we will convert a wide character string */
    /* to a single byte character string.  We first set the locale   */
    /* to a single byte locale.  We choose a locale with             */
    /* CCSID 37.                                                     */

    if (setlocale(LC_ALL, LOCNAME_EN) == NULL)
        printf("setlocale failed.\n");

    length = wcstombs(string, wc2, 10);

    /* In this case wide characters ABC are converted to */
    /* single byte characters ABC, length is 3.   */

    printf("string = %s, length = %d\n\n", string, length);

    /* Now lets try a multibyte example.  We first must set the */
    /* locale to a multibyte locale.  We choose a locale with     */
    /* CCSID 5026  */

    if (setlocale(LC_ALL, LOCNAME) == NULL)
        printf("setlocale failed.\n");

    length = wcstombs(string, wc_string, 10);
```

```
    /* The hex look at string would now be:                  */
    /* C10E428142820FC2   length will be 8                   */
    /* You would need a device capable of displaying multibyte */
    /* characters to see this string.                        */

    printf("length = %d\n\n", length);

}
/*  The output should look like this:

string = ABC, length = 3

length = 8
                                    */
```

関連情報

- 216 ページの『mbstowcs() ― マルチバイト・ストリングからワイド文字ストリングへの変換』
- 482 ページの『wcslen() ― ワイド文字ストリング長の計算』
- 494 ページの『wcsrtombs() ― ワイド文字ストリングからマルチバイト・ストリングへの変換 (再開可能)』
- 515 ページの『wctomb() ― ワイド文字からマルチバイト文字への変換』
- 18 ページの『<stdlib.h>』

wcstoul() ― wcstoull() ― ワイド文字ストリングから符号なし long 型整数および符号なし long long 型整数への変換

フォーマット (wcstoul())

```
#include <wchar.h>
unsigned long int wcstoul(const wchar_t *nptr, wchar_t **endptr, int base);
```

フォーマット (wcstoull())

```
#include <wchar.h>
unsigned long long int wcstoull(const wchar_t *nptr, wchar_t **endptr, int base);
```

言語レベル: ANSI

スレッド・セーフ: はい。

ロケール依存: コンパイル・コマンドで LOCALETYPE(*LOCALE) が指定される場合、これらの関数の振る舞いは、現行ロケールの LC_CTYPE カテゴリーの影響を受ける可能性があります。また、コンパイル・コマンドで LOCALETYPE(*LOCALEUCS2) または LOCALETYPE(*LOCALEUTF) が指定される場合、これらの関数の振る舞いは、現行ロケールの LC_UNI_CTYPE カテゴリーの影響も受ける可能性があります。これらの関数は、コンパイル・コマンドで LOCALETYPE(*CLD) が指定される場合には使用できません。詳細については、 549 ページの『CCSID およびロケールの理解』を参照してください。

ワイド文字関数: 詳細については、 553 ページの『ワイド文字』 を参照してください。

説明

`wcstoul()` 関数は、 *nptr* が指すワイド文字ストリングの先頭部分を符号なし long 型整数値に変換します。 *nptr* パラメーターにより、符号なし long 型整数の数値として解釈できるワイド文字のシーケンスが示されます。 `wcstoul()` 関数は、数値として認識できないワイド文字に遭遇すると、その時点でストリン

グの読み取りを停止します。この文字は、ストリングの最後にある wchar_t ヌル文字である場合があります。また、基数以上の数字が最初に登場した場合、その数字が終了文字になる場合もあります。

wcstoull() サブルーチンは、ワイド文字ストリングを符号なし long long 型整数に変換します。ワイド文字ストリングは、iswspace サブルーチンによって判別された最初のスペース文字をスキップするように解析されます。スペース以外の文字の場合には、サブジェクト・ストリングが開始されます。これによって、*base* パラメーターで指定された基数で、long long 型整数が形成されます。サブジェクト・シーケンスは、想定した形式の符号なし long long 型整数である最長の初期サブストリングになるように定義されています。

endptr パラメーターの値がヌルでない場合、走査を終了した文字へのポインターは *endptr* に保管されます。符号なし long long 型整数を形成できない場合、 *endptr* パラメーターの値は *nptr* パラメーターの値に設定されます。

base パラメーターの値が 2 と 36 の間の場合には、サブジェクト・シーケンスの想定される形式は、基数が *base* パラメーターで指定される符号なし long long 型整数を表す文字と数字のシーケンスです。このシーケンスは、オプションで先頭に正符号 (+) または負符号 (-) が付きます。a (または A) から z (または Z) までの文字は 10 から 35 までの値と見なされます。この値が *base* パラメーターの値より小さい文字だけが許可されます。*base* パラメーターの値が 16 の場合には、文字 0x または 0X はオプションで一連の文字および数字の前に付くことがありますが、正符号 (+) または負符号 (-) があるとその後に続きます。

base パラメーターの値が 0 である場合には、ストリングによって base が決定されます。したがって、最初の符号 (オプション) の後で、0 から始まっている場合は 8 進変換を表し、0x または 0X で始まっている場合は 16 進変換を表すことになります。

errno の値は **EINVAL** (*endptr* がヌル、数値が検出されない、または基数が無効) または **ERANGE** (変換後の値が範囲外にある) に設定される可能性があります。

戻り値

wcstoul() 関数は、変換された符号なし long 型整数値を戻します。変換が実行できない場合、wcstoul() 関数は 0 を戻します。正しい値が表示可能な値の範囲外にある場合は、wcstoul() 関数は ULONG_MAX を戻し、errno を ERANGE に設定します。*nptr* が指すストリングが空であるか、想定される形式になっていない場合には、変換は行われません。*nptr* の値は、 *endptr* が NULL ポインターでない場合に *endptr* が指すオブジェクトに保管されます。

正常終了の場合、wcstoull() サブルーチンは変換された値を戻します。変換が行われない場合には、0 が戻され、 errno グローバル変数はエラーを示すように設定されます。正しい値が表示可能値の範囲外にある場合には、wcstoull() サブルーチンは ULONG_LONG_MAX の値を戻します。

wcstoul() の使用例

この例では wcstoul() 関数を使用して、ストリング *wcs* を符号なし long 型整数値に変換します。

```
#include <stdio.h>
#include <wchar.h>

#define BASE 2

int main(void)
{
   wchar_t *wcs = L"1000e13 camels";
   wchar_t *endptr;
```

```
    unsigned long int answer;

    answer = wcstoul(wcs, &endptr, BASE);
    printf("The input wide string used: `%ls`¥n"
           "The unsigned long int produced: %lu¥n"
           "The substring of the input wide string that was not"
           " converted to unsigned long: `%ls`¥n", wcs, answer, endptr);
    return 0;

    /*******************************************************************
       The output should be similar to:

       The input wide string used:  1000e13 camels
       The unsigned long int produced: 8
       The substring of the input wide string that was not converted to
       unsigned long:  e13 camels
    *******************************************************************/

}
```

関連情報

- 409 ページの『strtod() ― strtof() ― strtold ― 文字ストリングから double、浮動、および long double への変換』
- 412 ページの『strtod32() ― strtod64() ― strtod128() ― 文字ストリングから 10 進浮動小数点への変換』
- 418 ページの『strtol() ― strtoll() ― 文字ストリングから long 型および long long 型整数への変換』
- 498 ページの『wcstod() ― ワイド文字ストリングから double 型への変換』
- 499 ページの『wcstod32() ― wcstod64() ― wcstod128()―ワイド文字ストリングから 10 進浮動小数点への変換』
- 503 ページの『wcstol() ― wcstoll() ― ワイド文字ストリングから long 型および long long 型整数への変換』
- 20 ページの『<wchar.h>』

wcswcs() ― ワイド文字サブストリングの位置検出

フォーマット

```
#include <wchar.h>
wchar_t *wcswcs(const wchar_t *string1, const wchar_t *string2);
```

言語レベル: XPG4

スレッド・セーフ: はい。

ワイド文字関数: 詳細については、 553 ページの『ワイド文字』 を参照してください。

説明

`wcswcs()` 関数は、*string1* が指すワイド文字ストリング内の *string2* が最初に現れる位置を見つけます。一致プロセスで、 `wcswcs()` 関数は、*string2* を終了する wchar_t ヌル文字を無視します。

戻り値

`wcswcs()` 関数は、見つけたストリングへのポインターを戻すか、ストリングが見つからない場合は、NULL を戻します。*string2* がゼロ長のストリングを指す場合には、 `wcswcs()` は *string1* を戻します。

wcswcs() の使用例

この例では、buffer1 でワイド文字ストリング pr が最初に現れる位置を検索します。

```
#include <stdio.h>
#include <wchar.h>

#define SIZE 40

int main(void)
{
  wchar_t buffer1[SIZE] = L"computer program";
  wchar_t * ptr;
  wchar_t * wch = L"pr";

  ptr = wcswcs( buffer1, wch );
  printf( "The first occurrence of %ls in '%ls' is '%ls'¥n",
                          wch, buffer1, ptr );
}

/****************  Output should be similar to:  ******************

The first occurrence of pr in 'computer program' is 'program'
*/
```

関連情報

- 375 ページの『strchr() ― 文字の検索』
- 381 ページの『strcspn() ― 最初に一致した文字のオフセットの検索』
- 401 ページの『strpbrk() ― ストリング内の文字の検索』
- 406 ページの『strrchr() ― ストリング内で文字が最後に現れる位置の検出』
- 407 ページの『strspn() ―最初の不一致文字のオフセットの検索』
- 408 ページの『strstr() ― サブストリングの位置検出』
- 472 ページの『wcschr() ― ワイド文字の検索』
- 474 ページの『wcscmp() ― ワイド文字ストリングの比較』
- 477 ページの『wcscspn() ― 最初に一致したワイド文字のオフセットの検索』
- 490 ページの『wcspbrk() ― ストリング内のワイド文字の位置検出』
- 493 ページの『wcsrchr() ― ストリング内でワイド文字が最後に現れる位置の検出』
- 496 ページの『wcsspn() ― 最初の不一致ワイド文字のオフセットの検索』
- 20 ページの『<wchar.h>』

wcswidth() ― ワイド文字ストリングの表示幅の判別

フォーマット

```
#include <wchar.h>
int wcswidth (const wchar_t *wcs, size_t n);
```

言語レベル: XPG4

スレッド・セーフ: はい。

ロケール依存: コンパイル・コマンドで LOCALETYPE(*LOCALE) が指定される場合、この関数の振る舞いは、現行ロケールの LC_CTYPE カテゴリーの影響を受ける可能性があります。また、コンパイル・コマンドで LOCALETYPE(*LOCALEUCS2) または LOCALETYPE(*LOCALEUTF) が指定される場合、この関

数の振る舞いは、現行ロケールの LC_UNI_CTYPE カテゴリーの影響も受ける可能性があります。この関数は、コンパイル・コマンドに対して LOCALETYPE(*CLD) が指定されている場合には使用できません。詳細については、 549 ページの『CCSID およびロケールの理解』を参照してください。

ワイド文字関数: 詳細については、 553 ページの『ワイド文字』 を参照してください。

説明

`wcswidth()` 関数は、*wcs* が指すワイド・ストリング内で *n* 個のワイド文字 (*n* 個のワイド文字に達する前にヌル・ワイド文字が検出された場合には、*n* 個より少ないワイド文字) のグラフィック表示がディスプレイ装置で占有する出力位置の数を判別します。その数は、装置上のその位置とは無関係です。

`errno` の値は、**EINVAL** (非出力ワイド文字) に設定される可能性があります。

戻り値

`wcswidth()` 関数は、以下のいずれかを戻します。
- *wcs* がヌル・ワイド文字を指す場合は 0。
- *wcs* が指すワイド・ストリングによって占有された出力位置数。
- *wcs* が指すワイド・ストリングのワイド文字が出力ワイド文字でない場合は -1。

`wcswidth()` の使用例

```
#include <stdio.h>
#include <wchar.h>

int main(void)
{
   wchar_t *wcs = L"ABC";

   printf("wcs has a width of: %d¥n", wcswidth(wcs,3));
}

/************The output is as follows**************/
/*
 */
/*                wcs has a width of: 3                            */
/*
 */
/**********************************************/
```

関連情報

- 511 ページの『wcswidth() ― ワイド文字ストリングの表示幅の判別』
- 20 ページの『<wchar.h>』

wcsxfrm() ― ワイド文字ストリングの変換

フォーマット

```
#include <wchar.h>
size_t wcsxfrm (wchar_t *wcs1, const wchar_t *wcs2, size_t n);
```

言語レベル: XPG4

スレッド・セーフ: はい。

ロケール依存: コンパイル・コマンドで LOCALETYPE(*LOCALE) が指定される場合、この関数の振る舞いは、現行ロケールの LC_COLLATE カテゴリーの影響を受ける可能性があります。また、コンパイル・コマンドで LOCALETYPE(*LOCALEUTF) が指定される場合、この関数の振る舞いは、現行ロケールの LC_UNI_COLLATE カテゴリーの影響も受ける可能性があります。この関数は、コンパイル・コマンドで LOCALETYPE(*LOCALEUCS2) が指定される場合はサポートされません。この関数は、コンパイル・コマンドに対して LOCALETYPE(*CLD) が指定されている場合には使用できません。詳細については、 549 ページの『CCSID およびロケールの理解』を参照してください。

ワイド文字関数: 詳細については、 553 ページの『ワイド文字』 を参照してください。

説明

`wcsxfrm()` 関数は、 *wcs2* が指すワイド文字ストリングを文字照合重みを表示する値に変形し、結果のワイド文字ストリングを *wcs1* が指すワイド文字ストリングに配置します。

戻り値

`wcsxfrm()` 関数は、変形されたワイド文字ストリングの長さ (終了ヌル・ワイド文字コードは除く) を戻します。戻された値が *n* またはそれを超える場合には、*wcs1* が指す配列の内容は保証されません。

`wcsxfrm()` が正常に実行されなかった場合、errno が変更されます。errno の値は、 EINVAL (*wcs1* または *wcs2* 引数に、現行ロケールで使用不可の文字が含まれている) に設定される可能性があります。

`wcsxfrm()` の使用例

```
#include <stdio.h>
#include <wchar.h>

int main(void)
{
   wchar_t *wcs;
   wchar_t buffer[80];
   int length;

   printf("Type in a string of characters.¥n ");
   wcs = fgetws(buffer, 80, stdin);
   length = wcsxfrm(NULL, wcs, 0);
   printf("You would need a %d element array to hold the wide string¥n", length);
   printf("¥n¥n%S¥n¥n transformed according", wcs);
   printf(" to this program's locale. ¥n");
}
```

関連情報

- 422 ページの『strxfrm() ― ストリングの変換』
- 20 ページの『<wchar.h>』

---

wctob() ― ワイド文字からバイトへの変換

フォーマット

```
#include <stdio.h>
#include <wchar.h>
int wctob(wint_t wc);
```

言語レベル: ANSI

スレッド・セーフ: はい。

ロケール依存: この関数の振る舞いは、現行ロケールの LC_CTYPE カテゴリーの影響を受ける可能性があります。また、この振る舞いは、LOCALETYPE(*LOCALEUCS2) または LOCALETYPE(*LOCALEUTF) がコンパイル・コマンドに対して指定されている場合は、現行ロケールの LC_UNI_CTYPE カテゴリーの影響を受ける可能性もあります。この関数は、コンパイル・コマンドに対して LOCALETYPE(*CLD) が指定されている場合には使用できません。詳細については、 549 ページの『CCSID およびロケールの理解』を参照してください。

ワイド文字関数: 詳細については、 553 ページの『ワイド文字』 を参照してください。

説明

`wctob()` 関数は、マルチバイト文字の長さが初期シフト状態時に 1 バイトである、拡張文字セットのメンバーに *wc* が対応しているかどうかを判別します。

戻り値

c が 1 バイトの長さのマルチバイト文字に対応している場合、 `wctob()` 関数は単一バイト表示を戻します。そうでない場合には、EOF を戻します。

変換エラーが発生した場合、`errno` は **ECONVERT** に設定される可能性があります。

`wctob()` の使用例

この例では `wctob()` 関数を使用し、ワイド文字 A が有効な 1 バイト文字であるかどうかをテストします。

```
#include <stdio.h>
#include <wchar.h>

int main(void)
{
   wint_t wc = L'A';

   if (wctob(wc) == wc)
      printf("%lc is a valid single byte character¥n", wc);
   else
      printf("%lc is not a valid single byte character¥n", wc);
   return 0;

   /************************************************************
      The output should be similar to:

      A is a valid single byte character
   ************************************************************/
}
```

関連情報

- 220 ページの『mbtowc() ― マルチバイト文字からワイド文字への変換』
- 515 ページの『wctomb() ― ワイド文字からマルチバイト文字への変換』
- 505 ページの『wcstombs() ― ワイド文字ストリングからマルチバイト・ストリングへの変換』
- 20 ページの『<wchar.h>』

wctomb() ― ワイド文字からマルチバイト文字への変換

フォーマット

```
#include <stdlib.h>
int wctomb(char *string, wchar_t character);
```

言語レベル: ANSI

スレッド・セーフ: いいえ。代わりに `wcrtomb()` を使用します。

ロケール依存: この関数の振る舞いは、現行ロケールの LC_CTYPE カテゴリーの影響を受ける可能性があります。また、この振る舞いは、LOCALETYPE(*LOCALEUCS2) または LOCALETYPE(*LOCALEUTF) がコンパイル・コマンドに対して指定されている場合は、現行ロケールの LC_UNI_CTYPE カテゴリーの影響を受ける可能性もあります。詳細については、 549 ページの『CCSID およびロケールの理解』を参照してください。

ワイド文字関数: 詳細については、 553 ページの『ワイド文字』 を参照してください。

説明

`wctomb()` 関数は、*character* の wchar_t 値を *string* が指すマルチバイト配列に変換します。 *character* の値が 0 の場合には、関数は初期シフト状態のまま残ります。`wctomb()` 関数は、最大 MB_CUR_MAX 文字を *string* に保管します。

ワイド文字の変換は、`wcstombs()` で説明したものと同じです。Unicode 例については、この関数を参照してください。

戻り値

`wctomb()` 関数は、マルチバイト文字の長さ (バイト単位) を戻します。 *character* が有効なマルチバイト文字でない場合、値 -1 が戻されます。 *string* が NULL ポインターである場合、`wctomb()` 関数はシフト依存のエンコードが使用されるとゼロ以外を戻し、それ以外の場合は 0 を戻します。

変換エラーが発生した場合、`errno` は **ECONVERT** に設定される可能性があります。

`wctomb()` の使用例

この例では、ワイド文字 c をマルチバイト文字に変換します。

```
#include <stdio.h>
#include <stdlib.h>
#include <wchar.h>

#define SIZE 40

int main(void)
{
  static char  buffer[ SIZE ];
  wchar_t wch = L'c';
  int length;

  length = wctomb( buffer, wch );
  printf( "The number of bytes that comprise the multibyte "
            "character is %i¥n", length );
  printf( "And the converted string is ¥"%s¥"¥n", buffer );
}

/****************  Output should be similar to:  ******************

The number of bytes that comprise the multibyte character is 1
And the converted string is "c"
*/
```

関連情報

- 220 ページの『mbtowc() ― マルチバイト文字からワイド文字への変換』
- 482 ページの『wcslen() ― ワイド文字ストリング長の計算』
- 467 ページの『wcrtomb() ― ワイド文字からマルチバイト文字への変換 (再開可能)』
- 505 ページの『wcstombs() ― ワイド文字ストリングからマルチバイト・ストリングへの変換』
- 494 ページの『wcsrtombs() ― ワイド文字ストリングからマルチバイト・ストリングへの変換 (再開可能)』
- 18 ページの『<stdlib.h>』

wctrans() ―文字マッピングのハンドルの取得

フォーマット

```
#include <wctype.h>
wctrans_t wctrans(const char *property);
```

言語レベル: ANSI

スレッド・セーフ: はい。

説明

`wctrans()` 関数は、wctrans_t 型の値を戻します。この値は、ワイド文字間のマッピングを示します。ストリング引数 *property* はワイド文字マッピング名です。ワイド文字マッピング名と同等の wctrans_t は、この関数で戻されます。 `toupper` および `tolower` ワイド文字マッピング名は、すべてのロケールで定義されます。

戻り値

property が有効なワイド文字マッピング名である場合、`wctrans()` 関数は、 `towctrans()` 関数への 2 番目の引数として有効なゼロ以外の値を戻します。それ以外の場合は、0 を戻します。

`wctrans()` の使用例

この例では、英字の小文字を大文字に、英字の大文字を小文字に変換します。

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <wchar.h>
#include <wctype.h>

int main()
{
   char *alpha = "abcdefghijklmnopqrstuvwxyz";
   char *tocase[2] = {"toupper", "tolower"};
   wchar_t *wcalpha;
   int i, j;
   size_t alphalen;

   alphalen = strlen(alpha)+1;
   wcalpha = (wchar_t *)malloc(sizeof(wchar_t)*alphalen);

   mbstowcs(wcalpha, alpha, 2*alphalen);

   for (i=0; i<2; ++i) {
      printf("Input string: %ls¥n", wcalpha);
      for (j=0; j<strlen(alpha); ++j) {
   wcalpha[j] = (wchar_t)towctrans((wint_t)wcalpha[j], wctrans(tocase[i]));
   }
  printf("Output string: %ls¥n", wcalpha);
  printf("¥n");
  }
 return 0;

/****************  Output should be similar to:  *****************

Input string: abcdefghijklmnopqrstuvwxyz
Output string: ABCDEFGHIJKLMNOPQRSTUVWXYZ
Input string: ABCDEFGHIJKLMNOPQRSTUVWXYZ
Output string: abcdefghijklmnopqrstuvwxyz

*******************************************************************/
```

関連情報

- 435 ページの『towctrans() ― ワイド文字の変換』
- 20 ページの『<wctype.h>』

wctype() ― 文字特性種別のハンドルの取得

フォーマット

```
#include <wctype.h>
wctype_t wctype(const char *property);
```

言語レベル: XPG4

スレッド・セーフ: はい。

説明

wctype() 関数は、有効な文字クラス名に対して定義されます。 *property* は、総称文字クラスを識別するストリングです。次の文字クラス名は、すべてのロケールで定義されます (alnum、alpha、blank、cntrl、digit、graph、lower、print、punct、space、upper、xdigit)。この関数は、wctype_t 型の値を戻します。この値は、iswctype() 関数の呼び出しに対する 2 番目の引数として使用できます。

wctype() 関数は、プログラムのロケール (カテゴリー LC_CTYPE) の文字型情報によって定義されたコード化文字セットの規則に従って wctype_t の値を判別します。wctype() によって戻された値は、カテゴリー LC_CTYPE を変更する setlocale() を呼び出すまで有効です。

戻り値

指定された特性名が無効の場合、wctype() 関数はゼロを戻します。そうでない場合には、iswctype() の呼び出しに使用できる型 wctype_t の値を戻します。

wctype() の使用例

```
#include <wchar.h>

#define   UPPER_LIMIT   0xFF
      int   main(void)
      {
         int   wc;
         for   (wc   =   0;   wc  <=   UPPER_LIMIT;   wc++)   {
            printf("%#4x ",   wc);
            printf("%c",   iswctype(wc,   wctype("print"))  ?   wc    : "
");
            printf("%s",   iswctype(wc,   wctype("alnum"))  ?   "AN"  : "
");
            printf("%s",   iswctype(wc,   wctype("alpha"))  ?   "A"   : "
");
            printf("%s",   iswctype(wc,   wctype("blank"))  ?   "B"   : "
");
            printf("%s",   iswctype(wc,   wctype("cntrl"))  ?   "C"   : "
");
            printf("%s",   iswctype(wc,   wctype("digit"))  ?   "D"   : "
");
            printf("%s",   iswctype(wc,   wctype("graph"))  ?   "G"   : "
");
            printf("%s",   iswctype(wc,   wctype("lower"))  ?   "L"   : "
");
            printf("%s",   iswctype(wc,   wctype("punct"))  ?   "PU"  : "
");
            printf("%s",   iswctype(wc,   wctype("space"))  ?   "S"   : "
");
            printf("%s",   iswctype(wc,   wctype("print"))  ?   "PR"  : "
");
            printf("%s",   iswctype(wc,   wctype("upper"))  ?   "U"   : "
");
            printf("%s",   iswctype(wc,   wctype("xdigit")) ?   "X"   : "
");
            putchar('¥n');
         }
         return   0;
        /***********************************************************************
            The   output   should   be   similar   to   :
             :
            0x1f              C
            0x20           B                S    PR
            0x21   !                  G       PU      PR
            0x22   "                  G       PU      PR
            0x23   #                  G       PU      PR
            0x24   $                  G       PU      PR
            0x25   %                  G       PU      PR
            0x26   &                  G       PU      PR
            0x27   '                  G       PU      PR
            0x28   (                  G       PU      PR
            0x29   )                  G       PU      PR
            0x2a   *                  G       PU      PR
            0x2b   +                  G       PU      PR
            0x2c   ,                   G       PU      PR
            0x2d   -                  G       PU      PR
            0x2e   .                   G       PU      PR
            0x2f   /                   G       PU      PR
            0x30   0   AN           D    G             PR      X
            0x31   1   AN           D    G             PR      X
            0x32   2   AN           D    G             PR      X
            0x33   3   AN           D    G             PR      X
            0x34   4   AN           D    G             PR      X
            0x35   5   AN           D    G             PR      X
             :
         ***********************************************************************
      }
```

関連情報

- 20 ページの『<wchar.h>』
- 20 ページの『<wctype.h>』

wcwidth() ― ワイド文字の表示幅の判別

フォーマット

```
#include <wchar.h>
int wcwidth (const wint_t wc);
```

言語レベル: XPG4

スレッド・セーフ: はい。

ロケール依存: この関数の振る舞いは、現行ロケールの LC_CTYPE カテゴリーの影響を受ける可能性があります。また、この振る舞いは、LOCALETYPE(*LOCALEUCS2) または LOCALETYPE(*LOCALEUTF) がコンパイル・コマンドに対して指定されている場合は、現行ロケールの LC_UNI_CTYPE カテゴリーの影響を受ける可能性もあります。この関数は、コンパイル・コマンドに対して LOCALETYPE(*CLD) が指定されている場合には使用できません。詳細については、 549 ページの『CCSID およびロケールの理解』を参照してください。

ワイド文字関数: 詳細については、 553 ページの『ワイド文字』 を参照してください。

説明

`wcwidth()` 関数は、 *wc* のグラフィック表示がディスプレイで占有する出力位置数を判別します。各出力ワイド文字は、ディスプレイ上にそれ独自の出力位置数を占有します。その数は、装置上のその位置とは無関係です。

`errno` の値は、**EINVAL** (非出力ワイド文字) に設定される可能性があります。

戻り値

`wcwidth()` 関数は、以下のいずれかを戻します。

- *wc* がヌル・ワイド文字の場合は 0。
- *wc* が占有する出力位置数。
- *wc* が出力ワイド文字でない場合は -1。

`wcwidth()` の使用例

```
#include <stdio.h>
#include <wchar.h>

int  main(void)
{
   wint_t  wc  =  L'A';

   printf("%lc  has  a  width  of  %d¥n",  wc,  wcwidth(wc));
   return  0;

  /***************************************************************************
   The  output  should  be  similar  to  :
   A  has  a  width  of  1
   ***************************************************************************/

}
```

関連情報

- 511 ページの『wcswidth() ― ワイド文字ストリングの表示幅の判別』
- 20 ページの『<wchar.h>』

wfopen() ―オープン・ファイル

フォーマット

```
#include <ifs.h>
FILE * wfopen(const wchar_t *filename,const wchar_t *mode);
```

言語レベル: ILE C Extension

スレッド・セーフ: はい

ロケール依存: この関数は、コンパイル・コマンドで LOCALETYPE(*LOCALEUCS2) または LOCALETYPE(*LOCALEUTF) が指定されている場合のみ使用可能です。詳細については、 549 ページの『CCSID およびロケールの理解』を参照してください。

統合ファイル・システム・インターフェース: この関数は、コンパイル・コマンドに対して SYSIFCOPT(*NOIFSIO) が指定されている場合には使用できません。

ワイド文字関数: 詳細については、 553 ページの『ワイド文字』 を参照してください。

説明

`wfopen()` 関数は、以下の点を除いて `fopen()` 関数と同じです。

- `wfopen()` は、ワイド文字としてファイル名およびモードを受け入れます。
- コンパイル・コマンドで LOCALETYPE(*LOCALEUCS2) が指定される場合、 `wfopen()` (ccsid=value、o_ccsid=value、および codepage=value キーワードが指定されていない場合に使用) でオープンされたファイルのデフォルト CCSID は UCS2 です。コンパイル・コマンドで LOCALETYPE(*LOCALEUTF) が指定される場合、デフォルト CCSID は UTF-32 です。

wmemchr() ―ワイド文字バッファーでのワイド文字の位置検出

フォーマット

```
#include <wchar.h>
wchar_t *wmemchr(const wchar_t *s, wchar_t c, size_t n);
```

言語レベル: ANSI

スレッド・セーフ: はい。

ワイド文字関数: 詳細については、 553 ページの『ワイド文字』 を参照してください。

説明

`wmemchr()` 関数は、*s* が指すオブジェクトの先頭にある *n* 個のワイド文字内から、*c* が最初に現れる位置を検出します。*n* の値が 0 の場合は、`wmemchr()` 関数は *c* のオカレンスを検出しないで、NULL ポインターを戻します。

戻り値

`wmemchr()` 関数は、検出したワイド文字へのポインターを戻すか、オブジェクト内でワイド文字が発生しなかった場合は、NULL ポインターを戻します。

`wmemchr()` の使用例

この例では、ワイド文字ストリングで 'A' が最初に現れる位置を検索します。

```
#include <stdio.h>
#include <wchar.h>

main()
{
   wchar_t *in = L"1234ABCD";
   wchar_t *ptr;
   wchar_t fnd = L'A';

   printf("¥nEXPECTED: ABCD");
   ptr = wmemchr(in, L'A', 6);
   if (ptr == NULL)
      printf("¥n** ERROR ** ptr is NULL, char L'A' not found¥n");
   else
      printf("¥nRECEIVED: %ls ¥n", ptr);
}
```

関連情報

- 221 ページの『memchr() — バッファーの検索』
- 375 ページの『strchr() — 文字の検索』
- 472 ページの『wcschr() — ワイド文字の検索』
- 『wmemcmp() —ワイド文字バッファーの比較』
- 524 ページの『wmemcpy() —ワイド文字バッファーのコピー』
- 525 ページの『wmemmove() — ワイド文字バッファーのコピー』
- 526 ページの『wmemset() — 値に対するワイド文字バッファーの設定』
- 20 ページの『<wchar.h>』

wmemcmp() —ワイド文字バッファーの比較

フォーマット

```
#include <wchar.h>
int wmemcmp(const wchar_t *s1, const wchar_t *s2, size_t n);
```

言語レベル: ANSI

スレッド・セーフ: はい。

ワイド文字関数: 詳細については、 553 ページの『ワイド文字』 を参照してください。

説明

wmemcmp() 関数は、*s1* が指すオブジェクトの最初の *n* ワイド文字と *s2* が指すオブジェクトの最初の *n* ワイド文字を比較します。 *n* の値が 0 の場合、wmemcmp() 関数は 0 を戻します。

戻り値

wmemcmp() 関数は、2 つのストリング *s1* と *s2* の間の関係に応じて値を戻します。

整数値	**意味**
0 より小さい値	*s1* は *s2* より小さい
0	*s1* は *s2* と等しい
0 より大きい値	*s1* は *s2* より大きい

wmemcmp() の使用例

この例では、wmemcmp() 関数を使用して、ワイド文字ストリングを内から外へ比較します。

```
#include <wchar.h>
#include <stdio.h>
#include <locale.h>

main()
{
   int rc;
   wchar_t *in = L"12345678";
   wchar_t *out = L"12AAAAAB";
   setlocale(LC_ALL, "POSIX");

   printf("¥nGREATER is the expected result");
   rc = wmemcmp(in, out, 3);
   if (rc == 0)
      printf("¥nArrays are EQUAL %ls %ls ¥n", in, out);
   else
   {
   if (rc > 0)
      printf("¥nArray %ls GREATER than %ls ¥n", in, out);
   else
      printf("¥nArray %ls LESS than %ls ¥n", in, out);
   }

   /******************************************************
      The output should be:

      GREATER is the expected result
      Array 12345678 GREATER than 12AAAAAB
   ******************************************************/
}
```

関連情報

- 222 ページの『memcmp() — バッファーの比較』
- 376 ページの『strcmp() — ストリングの比較』
- 474 ページの『wcscmp() — ワイド文字ストリングの比較』
- 485 ページの『wcsncmp() — ワイド文字ストリングの比較』
- 521 ページの『wmemchr() —ワイド文字バッファーでのワイド文字の位置検出』
- 『wmemcpy() —ワイド文字バッファーのコピー』
- 525 ページの『wmemmove() — ワイド文字バッファーのコピー』
- 526 ページの『wmemset() — 値に対するワイド文字バッファーの設定』
- 20 ページの『<wchar.h>』

wmemcpy() —ワイド文字バッファーのコピー

フォーマット

```
#include <wchar.h>
wchar_t *wmemcpy(wchar_t *s1, const wchar_t *s2, size_t n);
```

言語レベル: ANSI

スレッド・セーフ: はい。

ワイド文字関数: 詳細については、 553 ページの『ワイド文字』 を参照してください。

説明

`wmemcpy()` 関数は、*n* ワイド文字を、 *s2* が指すオブジェクトから *s1* が指すオブジェクトへコピーします。*s1* と *s2* がオーバーラップする場合、コピーの結果は予測不能です。*n* の値が 0 の場合、`wmemcpy()` 関数は 0 個のワイド文字をコピーします。

戻り値

`wmemcpy()` 関数は、*s1* の値を戻します。

`wmemcpy()` の使用例

この例では、外から内へ、最初の 4 つの文字をコピーします。予期される出力では、両方のストリングの最初の 4 文字が「ABCD」になります。

```
#include <wchar.h>
#include <stdio.h>

main()
{
   wchar_t *in = L"12345678";
   wchar_t *out = L"ABCDEFGH";
   wchar_t *ptr;

   printf("¥nExpected result: First 4 chars of in change");
   printf(" and are the same as first 4 chars of out");
   ptr = wmemcpy(in, out, 4);
   if (ptr == in)
      printf("¥nArray in %ls array out %ls ¥n", in, out);
   else
   {
   printf("¥n*** ERROR ***");
   printf(" returned pointer wrong");
   }
}
```

関連情報

- 223 ページの『memcpy() ― バイトのコピー』
- 380 ページの『strcpy() ― ストリングのコピー』
- 397 ページの『strncpy() ― ストリングのコピー』
- 476 ページの『wcscpy() ― ワイド文字ストリングのコピー』
- 487 ページの『wcsncpy() ― ワイド文字ストリングのコピー』
- 521 ページの『wmemchr() ―ワイド文字バッファーでのワイド文字の位置検出』
- 522 ページの『wmemcmp() ―ワイド文字バッファーの比較』
- 『wmemmove() ― ワイド文字バッファーのコピー』
- 526 ページの『wmemset() ― 値に対するワイド文字バッファーの設定』
- 20 ページの『<wchar.h>』

wmemmove() ― ワイド文字バッファーのコピー

フォーマット

```
#include <wchar.h>
wchar_t *wmemmove(wchar_t *s1, const wchar_t *s2, size_t n);
```

言語レベル: ANSI

スレッド・セーフ: はい。

ワイド文字関数: 詳細については、 553 ページの『ワイド文字』 を参照してください。

説明

wmemmove() 関数は、*n* ワイド文字を、 *s2* が指すオブジェクトから *s1* が指すオブジェクトへコピーします。まず *s2* が示すオブジェクトの *n* ワイド文字が、*s1* または *s2* が示すオブジェクトと重ならない *n* ワイド文字の一時配列にコピーされます。その後、wmemmove() 関数は、 *n* ワイド文字を、一時配列から *s1* によって指されるオブジェクトへコピーします。*n* の値が 0 の場合、wmemmove() 関数は 0 個のワイド文字をコピーします。

戻り値

wmemmove() 関数は、*s1* の値を戻します。

wmemmove() の使用例

この例では、ストリング内の最初の 5 文字をコピーして、同じストリング内の最後の 5 文字をオーバーレイします。ストリングの長さは 9 文字のみなので、ソースとターゲットはオーバーラップします。

```
#include <wchar.h>
#include <stdio.h>

void main()
{
   wchar_t *theString = L"ABCDEFGHI";

   printf("¥nThe original string: %ls ¥n", theString);
   wmemmove(theString+4, theString, 5);
   printf("¥nThe string after wmemmove: %ls ¥n", theString);

   return;

   /********************************************************
      The output should be:

      The original string: ABCDEFGHI
      The string after wmemmove: ABCDABCDE
   ********************************************************/
}
```

関連情報

- 226 ページの『memmove() ― バイトのコピー』
- 521 ページの『wmemchr() ―ワイド文字バッファーでのワイド文字の位置検出』
- 524 ページの『wmemcpy() ―ワイド文字バッファーのコピー』
- 522 ページの『wmemcmp() ―ワイド文字バッファーの比較』
- 『wmemset() ― 値に対するワイド文字バッファーの設定』
- 20 ページの『<wchar.h>』

wmemset() ― 値に対するワイド文字バッファーの設定

フォーマット

```
#include <wchar.h>
wchar_t *wmemset(wchar_t *s, wchar_t c, size_t n);
```

言語レベル: ANSI

スレッド・セーフ: はい。

ワイド文字関数: 詳細については、 553 ページの『ワイド文字』 を参照してください。

説明

wmemset() 関数は、*c* の値を *s* が指すオブジェクトの最初の *n* ワイド文字にコピーします。*n* の値が 0 の場合、wmemset() 関数は 0 個のワイド文字をコピーします。

戻り値

wmemset() 関数は、*s* の値を戻します。

wmemset() の使用例

この例では、最初の 6 ワイド文字をワイド文字 ″A″ に設定します。

```
#include <wchar.h>
#include <stdio.h>

void main()
{
   wchar_t *in = L"1234ABCD";
   wchar_t *ptr;

   printf("¥nEXPECTED: AAAAAACD");
   ptr = wmemset(in, L'A', 6);
   if (ptr == in)
      printf("¥nResults returned - %ls ¥n", ptr);
   else
      {
      printf("¥n** ERROR ** wrong pointer returned¥n");
      }
}
```

関連情報

- 227 ページの『memset() ― 値へのバイトの設定』
- 521 ページの『wmemchr() ―ワイド文字バッファーでのワイド文字の位置検出』
- 524 ページの『wmemcpy() ―ワイド文字バッファーのコピー』
- 522 ページの『wmemcmp() ―ワイド文字バッファーの比較』
- 525 ページの『wmemmove() ― ワイド文字バッファーのコピー』
- 20 ページの『<wchar.h>』

wprintf() ― データのワイド文字としてのフォーマット設定と出力

フォーマット

```
#include <stdio.h>
int wprintf(const wchar_t *format,...);
```

言語レベル: ANSI

スレッド・セーフ: はい。

ロケール依存: この関数の振る舞いは、現行ロケールの LC_CTYPE カテゴリーおよび LC_NUMERIC カテゴリーの影響を受ける可能性があります。また、LOCALETYPE(*LOCALEUCS2) または LOCALETYPE(*LOCALEUTF) がコンパイル・コマンドで指定されている場合、この振る舞いは、現行ロケールの LC_UNI_CTYPE カテゴリーおよび LC_UNI_NUMERIC カテゴリーの影響も受ける可能性があります。この関数は、コンパイル・コマンドに対して LOCALETYPE(*CLD) が指定されている場合には使用できません。詳細については、 549 ページの『CCSID およびロケールの理解』を参照してください。

統合ファイル・システム・インターフェース: この関数は、コンパイル・コマンドに対して SYSIFCOPT(*NOIFSIO) が指定されている場合には使用できません。

ワイド文字関数: 詳細については、 553 ページの『ワイド文字』 を参照してください。

説明

wprintf(format, ...) は fwprintf(stdout, format, ...) と同等です。

戻り値

wprintf() 関数は、送信されたワイド文字の数を戻します。出力エラーが発生した場合、wprintf() 関数は負の値を戻します。

wprintf() の使用例

この例では、ワイド文字 *a* を出力します。日付および時刻は、ロケールの表現に応じてフォーマット設定できます。出力先は stdout です。

```
#include <wchar.h>
#include <stdarg.h>
#include <locale.h>

int main(void)

{
   setlocale(LC_ALL, "POSIX");
   wprintf (L"%c¥n", L'a');
   return(0);
}

 /* A long 'a' is written to stdout  */
```

関連情報

- 238 ページの『printf() ― 定様式の文字の出力』
- 56 ページの『btowc() ― 1 バイト文字のワイド文字への変換』
- 210 ページの『mbrtowc() ― マルチバイト文字からワイド文字への変換 (再始動可能)』
- 447 ページの『vfwprintf() ― 引数データのワイド文字としてのフォーマット設定とストリームへの書き込み』
- 149 ページの『fwprintf() ― ワイド文字としてのデータのフォーマット設定とストリームへの書き込み』
- 459 ページの『vswprintf() ― ワイド文字のフォーマット設定とバッファーへの書き込み』
- 20 ページの『<wchar.h>』

---

wscanf() ― ワイド文字書式ストリングを使用したデータの読み取り

フォーマット

```
#include <stdio.h>
int wscanf(const wchar_t *format,...);
```

言語レベル: ANSI

スレッド・セーフ: はい。

ロケール依存: この関数の振る舞いは、現行ロケールの LC_CTYPE カテゴリーおよび LC_NUMERIC カテゴリーの影響を受ける可能性があります。また、LOCALETYPE(*LOCALEUCS2) または LOCALETYPE(*LOCALEUTF) がコンパイル・コマンドで指定されている場合、この振る舞いは、現行ロケールの LC_UNI_CTYPE カテゴリーおよび LC_UNI_NUMERIC カテゴリーの影響も受ける可能性がありま

す。この関数は、コンパイル・コマンドに対して LOCALETYPE(*CLD) が指定されている場合には使用できません。詳細については、 549 ページの『CCSID およびロケールの理解』を参照してください。

統合ファイル・システム・インターフェース: この関数は、コンパイル・コマンドに対して SYSIFCOPT(*NOIFSIO) が指定されている場合には使用できません。

ワイド文字関数: 詳細については、 553 ページの『ワイド文字』 を参照してください。

説明

wscanf() 関数は、その引数の前に引数 stdin を置くと、fwscanf() 関数と同じ意味になります。

戻り値

変換の前に入力障害が起こった場合、wscanf() 関数はマクロ EOF の値を戻します。

そうでない場合、wscanf() 関数は割り当てられた入力項目の数を戻します。以前のマッチングが失敗した場合、この数は提供された数より少ないか、ゼロの場合もあります。

wscanf() の使用例

この例では、さまざまなタイプのデータを走査します。

```
#include <stdio.h>
#include <wchar.h>

int main(void)
{
   int i;
   float fp;
   char c,s[81];

   printf("Enter an integer, a real number, a character and a string : ¥n");
   if (wscanf(L"%d %f %c %s", &i, &fp,&c, s) != 4)
      printf("Some fields were not assigned¥n");
   else {
      printf("integer = %d¥n", i);
      printf("real number = %f¥n", fp);
      printf("character = %c¥n", c);
      printf("string = %s¥n", s);
   }
   return 0;

   /********************************************************************
      The output should be similar to:

      Enter an integer, a real number, a character and a string :
      12 2.5 a yes
      integer = 12
      real number = 2.500000
      character = a
      string = yes
   ********************************************************************/
}
```

関連情報

- 138 ページの『fscanf() ― フォーマット済みデータの読み取り』
- 149 ページの『fwprintf() ― ワイド文字としてのデータのフォーマット設定とストリームへの書き込み』

- 153 ページの『fwscanf() ― ワイド文字を使用したストリームからのデータの読み取り』
- 344 ページの『scanf() ― データの読み取り』
- 371 ページの『sscanf() ― データの読み取り』
- 423 ページの『swprintf() ― ワイド文字のフォーマット設定とバッファーへの書き込み』
- 425 ページの『swscanf() ― ワイド文字データの読み取り』
- 446 ページの『vfscanf() ― フォーマット済みデータの読み取り』
- 449 ページの『vfwscanf() ― フォーマット済みワイド文字データの読み取り』
- 453 ページの『vscanf() ― フォーマット済みデータの読み取り』
- 458 ページの『vsscanf() ― フォーマット済みデータの読み取り』
- 461 ページの『vswscanf() ― フォーマット済みワイド文字データの読み取り』
- 465 ページの『vwscanf() ― フォーマット済みワイド文字データの読み取り』
- 527 ページの『wprintf() ― データのワイド文字としてのフォーマット設定と出力』
- 20 ページの『<wchar.h>』

第 3 章 ランタイムに関する考慮事項

この章では、以下の内容について説明します。
- 例外および条件管理
- 言語をまたがるデータ型の互換性
- CCSID (コード化文字セット ID) ソース・ファイル変換
- | ヒープ・メモリー

errno マクロ

次の表は、ILE C ライブラリー関数が設定できるエラー・マクロをリストしたものです。

表 12. errno マクロ

エラー・マクロ	説明	設定元の関数
EBADDATA	メッセージ・データが無効です。	perror、strerror
EBADF	カタログ記述子が無効です。	catclose、catgets、clearerr、fgetc、fgetpos、fgets、fileno、freopen、fseek、fsetpos、getc、rewind
EBADKEYLN	指定されたキーの長さは無効です。	_Rreadk、_Rlocate
EBADMODE	指定されたファイル・モードは無効です。	fopen、freopen、_Ropen
EBADNAME	無効なファイル名が指定されました。	fopen、freopen、_Ropen
EBADPOS	指定された位置は無効です。	fsetpos
EBADSEEK	シーク操作のオフセットが無効です。	fgetpos、fseek
EBUSY	レコードまたはファイルが使用中です。	perror、strerror
ECONVERT	変換エラーです。	wcstomb、wcswidth
EDOM	数学関数内のドメイン・エラーです。	acos、asin、atan2、cos、exp、fmod、gamma、hypot、j0、j1、jn、y0、y1、yn、log、log10、pow、sin、strtol、strtoul、sqrt、tan
EGETANDPUT	書き込み操作後に、無許可の読み取り操作が発生しました。	fgetc、fread、getc、getchar
EILSEQ	有効なマルチバイト文字で文字シーケンスが構成されていません。	fgetwc、fgetws、getwc、mblen、mbrlen、mbrtowc、mbsrtowcs、mbstowcs、mbtowc、printf、scanf、ungetwc、wcrtomb、wcsrtombs、wcstombs、wctomb、wcswidth、wcwidth
EINVAL	シグナルが無効です。	printf、scanf、signal、swprintf、swscanf、wcstol、wcstoll、wcstoul、wcstoull

© Copyright IBM Corp. 1999, 2010

表 12. errno マクロ (続き)

エラー・マクロ	説明	設定元の関数
EIO	連続した入出力呼び出しが発生しました。	入出力
EIOERROR	リカバリー不能な入出力エラーが発生しました。	すべての入出力関数
EIORECERR	リカバリー可能な入出力エラーが発生しました。	すべての入出力関数
ENODEV	誤ったデバイスに対して、操作が試行されました。	fgetpos、fsetpos、fseek、ftell、rewind
ENOENT	ファイルまたはライブラリーが見つかりません。	perror、strerror
ENOPOS	指定された位置にレコードがありません。	fsetpos
ENOREC	レコードが見つかりません。	fread、perror、strerror
ENOTDLT	ファイルがオープンされていないため、削除操作が行えません。	_Rdelete
ENOTOPEN	ファイルがオープンされていません。	clearerr、fclose、fflush、fgetpos、fopen、freopen、fseek、ftell、setbuf、setvbuf、_Ropen、_Rclose
ENOTREAD	ファイルがオープンされていないため、読み取り操作が行えません。	fgetc、fread、ungetc、_Rreadd、_Rreadf、_Rreadindv、_Rreadk、_Rreadl、_Rreadn、_Rreadnc、_Rreadp、_Rreads、_Rlocate
ENOTUPD	ファイルがオープンされていないため、更新操作が行えません。	_Rrlslck、_Rupdate
ENOTWRITE	ファイルがオープンされていないため、書き込み操作が行えません。	fputc、fwrite、_Rwrite、_Rwrited、_Rwriterd
ENUMMBRS	メンバーが複数存在します。	ftell
ENUMRECS	レコードが多すぎます。	ftell
EPAD	書き込み操作で埋め込みが発生しました。	fwrite
EPERM	アクセス権限が不十分です。	perror、strerror
EPUTANDGET	読み取り操作の後、正しくない書き込み操作が発生しました。	fputc、fwrite、fputs、putc、putchar
ERANGE	数学関数内の範囲エラーです。	cos、cosh、gamma、exp、j0、j1、jn、y0、y1、yn、log、log10、ldexp、pow、sin、sinh、strtod、strtol、strtoul、tan、wcstol、wcstoll、wcstoul、wcstoull、wcstod
ERECIO	レコード入出力用にファイルがオープンされているため、文字単位の処理関数は使用できません。	fgetc、fgetpos、fputc、fread、fseek、fsetpos、ftell
ESTDERR	stderr がオープンできません。	feof、ferror、fgetpos、fputc、fseek、fsetpos、ftell、fwrite

表 12. errno マクロ (続き)

エラー・マクロ	説明	設定元の関数
ESTDIN	stdin がオープンできません。	fgetc、fgetpos、fread、fseek、fsetpos、ftell
ESTDOUT	stdout がオープンできません。	fgetpos、fputc、fseek、fsetpos、ftell、fwrite
ETRUNC	入出力操作で切り捨てが発生しました。	レコードの読み取りまたは書き込みを行う入出力関数によって、errno が ETRUNC に設定されます。

統合ファイル・システム対応 C ストリーム入出力の errno 値

統合ファイル・システム対応のストリーム入出力を使用する場合に指定できる設定について、次の表で紹介します。

表 13. 統合ファイル・システム対応 C ストリーム入出力の errno 値

C ストリーム関数	指定できる errno 値
clearerr	EBADF
fclose	EAGAIN、EBADF、EIO、ESCANFAILURE、EUNKNOWN
feof	EBADF
ferror	EBADF
fflush	EACCES、EAGAIN、EBADF、EBUSY、EDAMAGE、EFAULT、EFBIG、EINVAL、EIO、ENOMEM、ENOSPC、ETRUNC、EUNKNOWN、EPUTANDGET、ENOTWRITE、EPAD
fgetc	EBADF、EACCES、EAGAIN、EBUSY、EDAMAGE、EFAULT、EINVAL、EIO、ENOMEM、EUKNOWN、EGETANDPUT、EDOM、ENOTREAD
fgetpos	EACCESS、EAGAIN、EBADF、EBUSY、EDAMAGE、EFAULT、EINVAL、EIO、ENOSYSRSC、EUNATCH、EUNKNOWN
fgets	EBADF、EACCES、EAGAIN、EBUSY、EDAMAGE、EFAULT、EINVAL、EIO、ENOMEM、EUKNOWN、EGETANDPUT、EDOM、ENOTREAD
fgetwc	EBADF、EILSEQ
fgetws	EBADF、EILSEQ
fopen	EAGAIN、EBADNAME、EBADF、ECONVERT、EDAMAGE、EEXITS、EFAULT、EINVAL、EIO、EISDIR、ELOOP、ENOENT、ENOMEM、ENOSPC、ENOSYS、ENOSYSRSC、ENOTDIR、ESCANFAILURE
fprintf	EACCES、EAGAIN、EBADF、EBUSY、EDAMAGE、EFAULT、EFBIG、EINVAL、EIO、ENOMEM、ENOSPC、ETRUNC、EUNKNOWN、EPUTANDGET、ENOTWRITE、EPAD
fputc	EACCES、EAGAIN、EBADF、EBUSY、EDAMAGE、EFAULT、EFBIG、EINVAL、EIO、ENOMEM、ENOSPC、ETRUNC、EUNKNOWN、EPUTANDGET、ENOTWRITE、EPAD
fputs	EACCES、EAGAIN、EBADF、EBUSY、EDAMAGE、EFAULT、EFBIG、EINVAL、EIO、ENOMEM、ENOSPC、ETRUNC、EUNKNOWN、EPUTANDGET、ENOTWRITE、EPAD
fread	EBADF、EACCES、EAGAIN、EBUSY、EDAMAGE、EFAULT、EINVAL、EIO、ENOMEM、EUKNOWN、EGETANDPUT、EDOM、ENOTREAD
freopen	EACCES、EAGAIN、EBADNAME、EBADF、EBUSY、ECONVERT、EDAMAGE、EEXITS、EFAULT、EINVAL、EIO、EISDIR、ELOOP、EMFILE、ENAMETOOLONG、ENFILE、ENOENT、ENOMEM、ENOSPC、ENOSYS、ENOSYSRSC、ENOTDIR

表 13. 統合ファイル・システム対応 C ストリーム入出力の errno 値 (続き)

C ストリーム関数	指定できる errno 値
fscanf	EBADF、EACCES、EAGAIN、EBUSY、EDAMAGE、EFAULT、EINVAL、EIO、ENOMEM、EUKNOWN、EGETANDPUT、EDOM、ENOTREAD
fseek	EACCES、EAGAIN、EBADF、EBUSY、EDAMAGE、EINVAL、EIO、ENOENT、ENOSPC、ENOSYSRSC、ESPIPE、EUNKNOWN、EFAULT、EPERM、EUNATCH、EUNKNOWN
fsetpos	EACCES、EAGAIN、ABADF、EBUSY、EDAMAGE、EINVAL、EIO、ENOENT、ENOSPC、ENOSYSRSC、ESPIPE、EUNKNOWN、EFAULT、EPERM、EUNATCH、EUNKNOWN
ftell	EACCESS、EAGAIN、EBADF、EBUSY、EDAMAGE、EFAULT、EINVAL、EIO、ENOSYSRSC、EUNATCH、EUNKNOWN
fwrite	EACCESS、EAGAIN、EBADF、EBUSY、EDAMAGE、EFAULT、EINVAL、EIO、ENOSYSRSC、EUNATCH、EUNKNOWN
getc	EBADF、EACCES、EAGAIN、EBUSY、EDAMAGE、EFAULT、EINVAL、EIO、ENOMEM、EUKNOWN、EGETANDPUT、EDOM、ENOTREAD
getchar	EBADF、EACCES、EAGAIN、EBUSY、EDAMAGE、EFAULT、EINVAL、EIO、ENOMEM、EUKNOWN、EGETANDPUT、EDOM、ENOTREAD
gets	EBADF、EACCES、EAGAIN、EBUSY、EDAMAGE、EFAULT、EINVAL、EIO、ENOMEM、EUKNOWN、EGETANDPUT、EDOM、ENOTREAD
getwc	EBADF、EILSEQ
perror	EBADF
printf	EACCES、EAGAIN、EBADF、EBUSY、EDAMAGE、EFAULT、EFBIG、EILSEQ、EINVAL、EIO、ENOMEM、ENOSPC、ETRUNC、EUNKNOWN、EPUTANDGET、ENOTWRITE、EPAD
putc	EACCES、EAGAIN、EBADF、EBUSY、EDAMAGE、EFAULT、EFBIG、EINVAL、EIO、ENOMEM、ENOSPC、ETRUNC、EUNKNOWN、EPUTANDGET、ENOTWRITE、EPAD
putchar	EACCES、EAGAIN、EBADF、EBUSY、EDAMAGE、EFAULT、EFBIG、EINVAL、EIO、ENOMEM、ENOSPC、ETRUNC、EUNKNOWN、EPUTANDGET、ENOTWRITE、EPAD
puts	EACCES、EAGAIN、EBADF、EBUSY、EDAMAGE、EFAULT、EFBIG、EINVAL、EIO、ENOMEM、ENOSPC、ETRUNC、EUNKNOWN、EPUTANDGET、ENOTWRITE、EPAD
remove	EACCES、EAGAIN、EBADNAME、EBADF、EBUSY、ECONVERT、EDAMAGE、EEXITS、EFAULT、EINVAL、EIO、EISDIR、ELOOP、ENAMETOOLONG、ENOENT、ENOMEM、ENOSPC、ENOTDIR、EPERM、EROOBJ、EUNKNOWN、EXDEV
rename	EACCES、EAGAIN、EBADNAME、EBUSY、ECONVERT、EDAMAGE、EEXIST、EFAULT、EINVAL、EIO、EISDIR、ELOOP、ENAMETOOLONG、ENOTEMPTY、ENOENT、ENOMEM、ENOSPC、ENOTDIR、EMLINK、EPERM、EUNKNOWN、EXDEV
rewind	EACCES、EAGAIN、EBADF、EBUSY、EDAMAGE、EINVAL、EIO、ENOENT、ENOSPC、ENOSYSRSC、ESPIPE、EUNKNOWN、EFAULT、EPERM、EUNATCH、EUNKNOWN
scanf	EBADF、EACCES、EAGAIN、EBUSY、EDAMAGE、EFAULT、EILSEQ、EINVAL、EIO、ENOMEM、EUKNOWN、EGETANDPUT、EDOM、ENOTREAD
setbuf	EBADF、EINVAL、EIO
setvbuf	EBADF、EINVAL、EIO

表 13. 統合ファイル・システム対応 C ストリーム入出力の errno 値 (続き)

C ストリーム関数	指定できる errno 値
tmpfile	EACCES、EAGAIN、EBADNAME、EBADF、EBUSY、ECONVERT、EDAMAGE、EEXITS、EFAULT、EINVAL、EIO、EISDIR、ELOOP、EMFILE、ENAMETOOLONG、ENFILE、ENOENT、ENOMEM、ENOSPC、ENOSYS、ENOSYSRSC、ENOTDIR、EPERM、EROOBJ、EUNKNOW N、EXDEV
tmpnam	EACCESS、EAGAIN、EBADF、EBUSY、EDAMAGE、EFAULT、EINVAL、EIO、ENOENT、ENOSYSRSC、EUNATCH、EUNKNOWN
ungetc	EBADF、EIO
ungetwc	EBADF、EILSEQ
vfprintf	EACCES、EAGAIN、EBADF、EBUSY、EDAMAGE、EFAULT、EFBIG、EINVAL、EIO、ENOMEM、ENOSPC、ETRUNC、EUNKNOWN、EPUTANDGET、ENOTWRITE、EPAD
vprintf	EACCES、EAGAIN、EBADF、EBUSY、EDAMAGE、EFAULT、EFBIG、EINVAL、EIO、ENOMEM、ENOSPC、ETRUNC、EUNKNOWN、EPUTANDGET、ENOTWRITE、EPAD

例外マッピングに対するレコード入出力エラー・マクロ

SIGIO シグナルが起動された場合に発生する事象について、次の表で説明します。*ESCAPE、*NOTIFY、および *STATUS のメッセージのみがモニターされます。

表 14. 例外マッピングに対するレコード入出力エラー・マクロ

説明	メッセージ (_EXCP_MSGID)	errno 設定
*STATUS および *NOTIFY	CPF4001 から CPF40FF まで、CPF4401 から CPF44FF まで、CPF4901 から CPF49FF まで、CPF5004	errno は設定されません。デフォルト応答がオペレーティング・システムに返されます。
リカバリー可能な入出力エラー	CPF4701 から CPF47FF まで、CPF4801 から CPF48FF まで、CPF5001 から CPF5003 まで、CPF5005 から CPF50FF まで	EIORECERR
リカバリー不能な入出力エラー[2]	CPF4101 から CPF41FF まで、CPF4201 から CPF42FF まで、CPF4301 から CPF43FF まで、CPF4501 から CPF45FF まで、CPF4601 から CPF46FF まで、CPF5101 から CPF51FF まで、CPF5201 から CPF52FF まで、CPF5301 から CPF53FF まで、CPF5401 から CPF54FF まで、CPF5501 から CPF55FF まで、CPF5601 から CPF56FF まで	EIOERROR
入出力操作で切り捨てが発生しました	C2M3003	ETRUNC
ファイルがオープンされません	C2M3004	ENOTOPEN
読み取り操作用にファイルがオープンされません	C2M3005	ENOTREAD
書き込み操作用にファイルがオープンされません	C2M3009	ENOTWRITE

表 14. 例外マッピングに対するレコード入出力エラー・マクロ (続き)

説明	メッセージ (_EXCP_MSGID)	errno 設定
無効なファイル名が指定されました	C2M3014	EBADNAME
指定されたファイル・モードは無効です	C2M3015	EBADMODE
更新操作用にファイルがオープンされません	C2M3041	ENOTUPD
削除操作用にファイルがオープンされません	C2M3042	ENOTDLT
指定されたキーの長さは無効です	C2M3044	EBADKEYLN
リカバリー不能な入出力エラーが発生しました	C2M3101	EIOERROR
リカバリー可能な入出力エラーが発生しました	C2M3102	EIORECERR

注:

- [1] エラーはユーザーにパーコレートされます。これによって、ユーザーのダイレクト・モニター・ハンドラー、ILE C 条件処理プログラム、およびシグナル・ハンドラーでコントロールが可能になります。 SIGIO の初期設定は SIG_IGN です。
- [2] リカバリー可能かリカバリー不能かは、デバイスのタイプによって決まります。以下の IBM 資料には、システム例外のリカバリー可能例およびリカバリー不能例について、個々のファイル・タイプ別に記載されています。
  - *ICF Programming*
  - *ADTS/400: Advanced Printer Function*
  - *System i アプリケーション表示プログラミング*
  - *データベース・プログラミング*

シグナル処理アクションの定義

コンパイル・コマンドで SYSIFCOPT(*NOASYNCSIGNAL) が指定された場合の、C シグナル値とそれらの処理アクション定義の初期状態について、次の表で説明します。 SIG_DFL は、常に状態をハンドラーにパーコレートします。「再開」とは、例外が処理され、アプリケーションが続行されることを意味します。

表 15. シグナル値の処理アクション定義

シグナル値	初期状態	SIG_DFL	SIG_IGN	ハンドラーからの戻り
SIGABRT[1]	SIG_DFL	パーコレート	無視	再開
SIGALL[2]	SIG_DFL	パーコレート	無視	再開
SIGFPE	SIG_DFL	パーコレート	無視[3]	再開[4]
SIGILL	SIG_DFL	パーコレート	無視[3]	再開[4]
SIGINT	SIG_DFL	パーコレート	無視	再開
SIGIO	SIG_IGN	パーコレート	無視	再開
SIGOTHER	SIG_DFL	パーコレート	無視[3]	再開[4]
SIGSEGV	SIG_DFL	パーコレート	無視[3]	再開[4]
SIGTERM	SIG_DFL	パーコレート	無視	再開
SIGUSR1	SIG_DFL	パーコレート	無視	再開
SIGUSR2	SIG_DFL	パーコレート	無視	再開

表 15. シグナル値の処理アクション定義 (続き)

シグナル値	初期状態	SIG_DFL	SIG_IGN	ハンドラーからの戻り
注:				
• [1] シグナル通知できるのは、raise() 関数または abort() 関数のみです。				
• [2] raise() 関数では、SIGALL をシグナル通知することはできません。				
• [3] シグナルの値が SIGFPE、SIGILL、または SIGSEGV の場合、振る舞いは予期できません。				
• [4] シグナルがハードウェア生成の場合、振る舞いは予期できません。				

コンパイル・コマンドで SYSIFCOPT(*ASYNCSIGNAL) が指定された場合の、C シグナル値とそれらの処理アクション定義の初期状態について、次の表で説明します。

表 16. シグナル値のデフォルト・アクション

値	デフォルト・アクション	意味
SIGABRT	2	異常終了
SIGFPE	2	マスクされない演算例外 (オーバーフロー、ゼロによる割り算、誤った演算など)
SIGILL	2	誤った関数イメージの検出
SIGINT	2	対話式アテンション
SIGSEGV	2	ストレージへの誤ったアクセス
SIGTERM	2	プログラムへ送信された終了要求
SIGUSR1	2	ユーザー・アプリケーションによる使用目的
SIGUSR2	2	ユーザー・アプリケーションによる使用目的
SIGALRM	2	alarm() が発信するタイムアウト・シグナル
SIGHUP	2	制御端末がハングアップしたか、または制御プロセスが終了した。
SIGKILL	1	取得できないか、または無視できない終了シグナル
SIGPIPE	3	読み取り中ではないパイプへの書き込み
SIGQUIT	2	端末の終了シグナル
SIGCHLD	3	終了または停止された子プロセス。SIGCLD は、このシグナルの別名。
SIGCONT	5	続行 (停止した場合)
SIGSTOP	4	取得できないか、または無視できないストップ・シグナル
SIGTSTP	4	端末のストップ・シグナル
SIGTTIN	4	制御端末から読み取ろうとしたバックグラウンド・プロセス
SIGTTOU	4	制御端末へ書き込もうとしたバックグラウンド・プロセス
SIGIO	3	入力または出力の完了
SIGURG	3	高帯域幅データがソケットで使用可能。
SIGPOLL	2	ポーリング可能イベント
SIGBUS	2	指定例外
SIGPRE	2	プログラミング例外
SIGSYS	2	無効なシステム呼び出し
SIGTRAP	2	トレース・トラップまたはブレークポイント・トラップ
SIGPROF	2	プロファイル・タイマーの有効期限切れ。

表 16. シグナル値のデフォルト・アクション (続き)

SIGVTALRM	2	仮想タイマーの有効期限切れ。
SIGXCPU	2	プロセッサーの制限時間を超過。
SIGXFSZ	2	ファイル・サイズの限界を超過。
SIGDANGER	2	システム・クラッシュ寸前。
SIGPCANCEL	2	取得できないか、または無視できないスレッド終了シグナル。

デフォルト・アクション:

1 プロセスの即時終了。

2 要求の終了。

3 シグナルの無視。

4 プロセスの停止。

5 現在停止している場合には、プロセスを続行。そうでない場合には、シグナルを無視。

i5/OS 例外マッピングに対するシグナル

シグナルにマップされるシステム例外メッセージについて、次の表で説明します。 *ESCAPE 例外メッセージは、すべてシグナルにマップされます。 SIGIO にマップされる *STATUS メッセージおよび *NOTIFY メッセージ (535 ページの表 14 で定義されている) は、シグナルにマップされます。

表 17. i5/OS 例外マッピングに対するシグナル

シグナル	メッセージ
SIGABRT	C2M1601
SIGALL	C2M1610 (明示的に生成される場合)
SIGFPE	C2M1602、MCH1201 から MCH1204 まで、MCH1206 から MCH1215 まで、MCH1221 から MCH1224 まで、MCH1838 から MCH1839 まで
SIGILL	C2M1603、MCH0401、MCH1002、MCH1004、MCH1205、 MCH1216 から MCH1219 まで、MCH1801 から MCH1802 まで、MCH1807 から MCH1808 まで、MCH1819 から MCH1820 まで、 MCH1824 から MCH1825 まで、MCH1832、MCH1837、MCH1852、MCH1854 から MCH1857 まで、MCH1867、 MCH2003 から MCH2004 まで、MCH2202、MCH2602、MCH2604、MCH2808、MCH2810 から MCH2811 まで、 MCH3201 から MCH3203 まで、MCH4201 から MCH4211 まで、MCH4213、MCH4296 から MCH4298 まで、 MCH4401 から MCH4403 まで、MCH4406 から MCH4408 まで、MCH4421、MCH4427 から MCH4428 まで、 MCH4801、MCH4804 から MCH4805 まで、MCH5001 から MCH5003 まで、MCH5401 から MCH5402 まで、 MCH5601、MCH6001 から MCH6002 まで、MCH6201、MCH6208、MCH6216、MCH6220、MCH6403、 MCH6601 から MCH6602 まで、MCH6609 から MCH6612 まで
SIGINT	C2M1604
SIGIO	C2M1609。例外マッピングについては、 535 ページの表 14 を参照。
SIGOTHER	C2M1611 (明示的に生成される場合)

表 17. *i5/OS 例外マッピングに対するシグナル (続き)*

シグナル	メッセージ
SIGSEGV	C2M1605、MCH0201、MCH0601 から MCH0606 まで、MCH0801 から MCH0803 まで、MCH1001、MCH1003、MCH1005 から MCH1006 まで、MCH1220、MCH1401 から MCH1402 まで、 MCH1602、MCH1604 から MCH1605 まで、MCH1668、MCH1803 から MCH1806 まで、MCH1809 から MCH1811 まで、 MCH1813 から MCH1815 まで、MCH1821 から MCH1823 まで、MCH1826 から MCH1829 まで、MCH1833、MCH1836、MCH1848、MCH1850、MCH1851、MCH1864 から MCH1866 まで、MCH1898、MCH2001 から MCH2002 まで、 MCH2005 から MCH2006 まで、MCH2201、MCH2203 から MCH2205 まで、MCH2401、MCH2601、MCH2603、 MCH2605、MCH2801 から MCH2804 まで、MCH2806 から MCH2809 まで、MCH3001、MCH3401 から MCH3408 まで、 MCH3410、MCH3601 から MCH3602 まで、MCH3603 から MCH3604 まで、MCH3802、MCH4001 から MCH4002 まで、MCH4010、MCH4212、MCH4404 から MCH4405 まで、MCH4416 から MCH4420 まで、MCH4422 から MCH4426 まで、 MCH4429 から MCH4437 まで、MCH4601、MCH4802 から MCH4803 まで、MCH4806 から MCH4812 まで、 MCH5201 から MCH5204 まで、MCH5602 から MCH5603 まで、MCH5801 から MCH5804 まで、MCH6203 から MCH6204 まで、MCH6206、MCH6217 から MCH6219 まで、MCH6221 から MCH6222 まで、MCH6401 から MCH6402 まで、MCH6404、 MCH6603 から MCH6608 まで、MCH6801
SIGTERM	C2M1606
SIGUSR1	C2M1607
SIGUSR2	C2M1608

ハンドラー理由コードの取り消し

次の表は、理由コードに設定されるビットをリストしたものです。活動化グループを停止すると、「**活動化グループが停止する**」ビットも理由コードに設定されます。これらのビットは、<except.h> の _CNL_Hndlr_Parms_T にある _CNL_MASK_T と相互に関連付けられている必要があります。第 2 列には、<except.h> の取り消し理由マスクに定義されたマクロ定数が記載されています。

表 18. *取り消された呼び出し理由コードの判別*

機能	理由コードに設定されたビット	論拠
ライブラリー・ルーチン		
exit	_EXIT_VERB	`exit` の定義は、処理の正常終了です。したがって、この関数によって取り消された呼び出しは、理由コード *normal* で実行されます。
abort	_ABNORMAL_TERM _EXIT_VERB	`abort` の定義は、処理の異常終了です。したがって、この関数によって取り消された呼び出しは、理由コード *abnormal* で実行されます。
longjmp	_JUMP	通常は、例外ハンドラーから戻る場合に、`longjmp()` 関数を使用します。ただし、例外以外の状態においても使用されることがあります。これは、プログラムの「通常」パスの一部として使用されます。したがって、これが原因で取り消された呼び出しは、理由コード *normal* で取り消されます。
未処理機能チェック	_ABNORMAL_TERM_ UNHANDLED_EXCP	異常状態の例外が未処理です。
システム API		

表 18. 取り消された呼び出し理由コードの判別 (続き)

機能	理由コードに設定されたビット	論拠
CEEMRCR	_ABNORMAL_TERM _EXCP_SENT	この API が使用されるのは、例外処理時のみです。通常は、再開できない呼び出し、または制御が再開されても振る舞いが予期できない呼び出しを取り消す目的で使用されます。これらの呼び出しによって、例外を処理することも可能でしたが、実際には実行されませんでした。この API によって取り消された呼び出しは、理由コード *abnormal* で実行されます。
QMHSNDPM /QMHRSNEM (エスケープ・メッセージ) メッセージ・ハンドラー API	_ABNORMAL_TERM _EXCP_SENT	ターゲット呼び出しにまで至るすべての呼び出しが、例外を処理する機会がないまま取り消されます。トピック『API』では、これらの API に関する情報を紹介しています。
i5/OS コマンド		
プロセスの終了	_ABNORMAL_TERM _PROCESS_TERM _AG_TERMINATING	活動化グループのシャットダウンが外部から開始された場合は、異常と見なされます。
RCLACTGRP	_ABNORMAL_TERM _RCLRSC	デフォルトは、異常終了です。 normal/abnormal フラグがコマンドに追加されていると、正常終了する場合があります。

表 19. 呼び出しの取り消しに対する共通の理由コード

ビット	説明	ヘッダー・ファイル定数 <except.h>
ビット 0	予約済み	
ビット 1	例外メッセージの送信により取り消された呼び出し	_EXCP_SENT
ビット 2 - 15	予約済み	
ビット 16	0 - プロセスの正常終了、1 - プロセスの異常終了	_ABNORMAL_TERM
ビット 17	活動化グループが終了。	_AG_TERMINATING
ビット 18	***活動化グループ再利用 (RCLACTGRP)*** によって開始。	_RCLRSC
ビット 19	プロセスの終了によって開始。	_PROCESS_TERM
ビット 20	`exit()` 関数によって開始。	_EXIT_VERB
ビット 21	未処理機能チェックによって開始。	_UNHANDLED_EXCP
ビット 22	`longjmp()` 関数により取り消された呼び出し。	_JUMP
ビット 23	例外処理のために jump により取り消された呼び出し	_JUMP_EXCP
ビット 24 - 31	予約済み (0)	

例外クラス

制御言語プログラムでは、**例外 ID** に基づいて、選択した例外のグループまたは単一の例外をモニターすることができます。例外ハンドラーがモニターする class2 の値は _C2_MH_ESCAPE、_C2_MH_STATUS、_C2_MH_NOTIFY、および _C2_MH_FUNCTION_CHECK のみです。 #pragma 例外ハンドラー・ディレクティブの使用方法の詳細については、「*IBM Rational Development Studio for i: ILE C/C++ コンパイラー参照*」を参照してください。この表では、指定可能な例外クラスが

すべて定義されています。

表 20. 例外クラス

ビット位置	<except.h> 内のヘッダー・ファイル定数	例外クラス
0	_C1_BINARY_OVERFLOW	2 進数のオーバーフローまたはゼロ除算
1	_C1_DECIMAL_OVERFLOW	10 進数のオーバーフローまたはゼロ除算
2	_C1_DECIMAL_DATA_ERROR	10 進データ・エラー
3	_C1_FLOAT_OVERFLOW	浮動小数点のオーバーフローまたはゼロ除算
4	_C1_FLOAT_UNDERFLOW	浮動小数点のアンダーフローまたは不正確な結果
5	_C1_INVALID_FLOAT_OPERAND	浮動小数点の無効オペランドまたは変換エラー
6	_C1_OTHER_DATA_ERROR	その他のデータ・エラー (編集マスクなど)
7	_C1_SPECIFICATION_ERROR	仕様 (オペランドの境界調整) エラー
8	_C1_POINTER_NOT_VALID	設定されていないポインター/無効なポインター型
9	_C1_OBJECT_NOT_FOUND	オブジェクトが見つからない
10	_C1_OBJECT_DESTROYED	オブジェクトが破棄された
11	_C1_ADDRESS_COMP_ERROR	アドレス計算のアンダーフローまたはオーバーフロー
12	_C1_SPACE_ALLOC_ERROR	指定したオフセットにスペースが割り振られていない
13	_C1_DOMAIN_OR_STATE_VIOLATION	ドメイン/状態の保護違反
14	_C1_AUTHORIZATION_VIOLATION	権限違反
15	_C1_JAVA_THROWN_CLASS	Java クラスにスローされた例外
16-28	_C1_VLIC_RESERVED	VLIC は予約済み
29	_C1_OTHER_MI_EXCEPTION	その他の MI 生成例外 (機能チェック以外のもの)
30	_C1_MI_GEN_FC_OR_MC	MI 生成の機能チェックまたはマシン・チェック
31	_C1_MI_SIGEXP_EXCEPTION	シグナル例外命令により生成されたメッセージ
32-39	適用不能	予約
40	_C2_MH_ESCAPE	*ESCAPE
41	_C2_MH_NOTIFY	*NOTIFY
42	_C2_MH_STATUS	*STATUS
43	_C2_MH_FUNCTION_CHECK	機能チェック
44-63	適用不能	予約

データ型の互換性

高水準言語は、それぞれさまざまなデータ型を持っています。複数の言語で作成されているプログラム間でデータを受け渡す場合は、それらの違いについて熟知している必要があります。

ILE C プログラミング言語の一部のデータ型は、他の言語に直接相当するものがありません。しかし、ILE C データ型を使用する他の言語で、データ型をシミュレートすることができます。

ILE C データ型と ILE RPG との互換性について、次の表で説明します。

表 21. ILE C データ型と ILE RPG との互換性

プロトタイプでの ILE C 宣言	ILE RPG D 仕様、列 33 から 39 まで	長さ	コメント
char[n] char *	nA	n	文字の配列 (n は、1 から 32766 まで)。
char	1A	1	*IN で始まる変数である標識。
char[n]	nS 0	n	ゾーン 10 進数。
char[2n]	nG	2n	追加されたグラフィック。
char[2n+2]	サポートされません。	2n+2	グラフィック・データ型
_Packed struct {short i; char[n]}	サポートされません。	n+2	可変長フィールド (i は予定の長さ、n は最大長)。
char[n]	D	8, 10	日付フィールド
char[n]	T	8	時刻フィールド
char[n]	Z	26	タイム・スタンプ・フィールド
short int	5I 0	2	整数フィールド
short unsigned int	5U 0	2	符号なし整数フィールド
int	10I 0	4	整数フィールド
unsigned int	10U 0	4	符号なし整数フィールド
long int	10I 0	4	整数フィールド
long unsigned int	10U 0	4	符号なし整数フィールド
struct {unsigned int : n}x;	サポートされません。	4	4 バイトの符号なし整数。ビット・フィールド。
float	サポートされません。	4	4 バイトの浮動小数点
double	サポートされません。	8	8 バイトの double 型
long double	サポートされません。	8	8 バイトの long double 型
enum	サポートされません。	1, 2, 4	列挙型
*	*	16	ポインター
decimal(n,p)	nP p	n/2+1	パック 10 進数。n は、必ず 30 以下。
union.element	キーワード OVERLAY(最長フィールド) 付きの <type>	エレメントの長さ	共用体のエレメント
data_type[n]	キーワード DIM(n) 付きの <type>	16	C がポインターを渡す先の配列
struct	データ構造体	n	構造体。この構造体には _Packed 修飾子を使用します。

表 21. ILE C データ型と ILE RPG との互換性 (続き)

プロトタイプでの ILE C 宣言	ILE RPG D 仕様、列 33 から 39 まで	長さ	コメント
関数へのポインター	* キーワード PROCPTR 付き	16	16 バイトのポインター

ILE C データ型と ILE COBOL との互換性について、次の表で説明します。

表 22. ILE C データ型と ILE COBOL との互換性

プロトタイプでの ILE C 宣言	ILE COBOL の LINKAGE SECTION	長さ	コメント
char[n] char *	PIC X(n)	n	文字の配列 (n は、1 から 3,000,000 まで)。
char	PIC 1 INDIC ..	1	標識
char[n]	PIC S9(n) DISPLAY	n	ゾーン 10 進数。
wchar_t[n]	PIC G(n)	2n	グラフィック・データ型
_Packed struct {short i; char[n]}	05 VL-FIELD. 10 i PIC S9(4) COMP-4. 10 data PIC X(n).	n+2	可変長フィールド (i は予定の長さ、n は最大長)。
char[n]	PIC X(n)	6	日付フィールド
char[n]	PIC X(n)	5	日フィールド
char	PIC X	1	曜日フィールド
char[n]	PIC X(n)	8	時刻フィールド
char[n]	PIC X(n)	26	タイム・スタンプ・フィールド
short int	PIC S9(4) COMP-4	2	2 バイトの符号付き整数。範囲は -9999 から +9999 まで。
short int	PIC S9(4) BINARY	2	2 バイトの符号付き整数。範囲は -9999 から +9999 まで。
int	PIC S9(9) COMP-4	4	4 バイトの符号付き整数。範囲は -999999999 から +999999999 まで。
int	PIC S9(9) BINARY	4	4 バイトの符号付き整数。範囲は -999999999 から +999999999 まで。
int	USAGE IS INDEX	4	4 バイト整数
long int	PIC S9(9) COMP-4	4	4 バイトの符号付き整数。範囲は -999999999 から +999999999 まで。
long int	PIC S9(9) BINARY	4	4 バイトの符号付き整数。範囲は -999999999 から +999999999 まで。
struct {unsigned int : n}x;	PIC 9(9) COMP-4 PIC X(4)	4	16 進数リテラルを使用して、ビット・フィールドを操作可能。
float	サポートされません。	4	4 バイトの浮動小数点
double	サポートされません。	8	8 バイトの double 型
long double	サポートされません。	8	8 バイトの long double 型
enum	サポートされません。	1, 2, 4	列挙型

表 22. ILE C データ型と ILE COBOL との互換性 (続き)

プロトタイプでの ILE C 宣言	ILE COBOL の LINKAGE SECTION	長さ	コメント
*	USAGE IS POINTER	16	ポインター
decimal(n,p)	PIC S9(n-p)V9(p) COMP-3	n/2+1	パック 10 進数。
decimal(n,p)	PIC S9(n-p) 9(p) PACKED-DECIMAL	n/2+1	パック 10 進数。
union.element	REDEFINES	エレメントの長さ	共用体のエレメント
data_type[n]	OCCURS	16	C がポインターを渡す先の配列
struct	01 record 05 field1 05 field2	n	構造体。この構造体には _Packed 修飾子を使用します。構造体の内容を変更する場合、受け渡す構造体はポインターとして構造体に渡す必要があります。
関数へのポインター	PROCEDURE-POINTER	16	プロシージャーへのポインター (16 バイト)
サポートされません。	PIC S9(18) COMP-4	8	8 バイト整数
サポートされません。	PIC S9(18) BINARY	8	8 バイト整数

ILE C データ型と ILE CL との互換性について、次の表で説明します。

表 23. ILE C データ型と ILE CL との互換性

プロトタイプでの ILE C 宣言	CL	長さ	コメント
char[n] char *	*CHAR LEN(&N)	n	文字の配列 (n は、1 から 32766 まで)。ヌル終了ストリング。例: CHGVAR &V1 VALUE (&V *TCAT X'00')。&V1 は、&V より 1 バイトだけ大きな値になります。
char	*LGL	1	「1」または「0」が保持されます。
_Packed struct {short i; char[n]}	サポートされません。	n+2	可変長フィールド (i は予定の長さ、n は最大長)。
整数型	サポートされません。	1, 2, 4	1 バイト、2 バイト、または 4 バイトの符号付き/符号なし整数。
浮動小数点定数	CL 定数のみ	4	4 バイトまたは 8 バイトの浮動小数点
decimal(n,p)	*DEC	n/2+1	パック 10 進数。n の限度は 15。p の限度は 9。
union.element	サポートされません。	エレメントの長さ	共用体のエレメント
struct	サポートされません。	n	構造体。この構造体には _Packed 修飾子を使用します。
関数へのポインター	サポートされません。	16	16 バイトのポインター

ILE C データ型と OPM RPG/400® との互換性について、次の表で説明します。

表 24. ILE C データ型と OPM RPG/400 との互換性

プロトタイプでの ILE C 宣言	OPM RPG/400 I 仕様、DS サブフィールド列の仕様	長さ	コメント
char[n] char *	1 10	n	文字の配列 (n は、1 から 32766 まで)。
char	*INxxxx	1	*IN で始まる変数である標識。
char[n]	1 nd (d>=0)	n	ゾーン 10 進数。n の限度は 30。
char[2n+2]	サポートされません。	2n+2	グラフィック・データ型
_Packed struct {short i; char[n]}	サポートされません。	n+2	可変長フィールド (i は予定の長さ、n は最大長)。
char[n]	サポートされません。	6, 8, 10	日付フィールド
char[n]	サポートされません。	8	時刻フィールド
char[n]	サポートされません。	26	タイム・スタンプ・フィールド
short int	B 1 20	2	2 バイトの符号付き整数。範囲は -9999 から +9999 まで。
int	B 1 40	4	4 バイトの符号付き整数。範囲は -999999999 から +999999999 まで。
long int	B 1 40	4	4 バイトの符号付き整数。範囲は -999999999 から +999999999 まで。
struct {unsigned int : n}x;	サポートされません。	4	4 バイトの符号なし整数。ビット・フィールド。
float	サポートされません。	4	4 バイトの浮動小数点
double	サポートされません。	8	8 バイトの double 型
long double	サポートされません。	8	8 バイトの long double 型
enum	サポートされません。	1, 2, 4	列挙型
*	サポートされません。	16	ポインター
decimal(n,p)	P 1 n/2+1d	n/2+1	パック 10 進数。n は、必ず 30 以下。
union.element	データ構造体のサブフィールド	エレメントの長さ	共用体のエレメント
data_type[n]	E-SPEC 配列	16	C がポインターを渡す先の配列
struct	データ構造体	n	構造体。この構造体には _Packed 修飾子を使用します。
関数へのポインター	サポートされません。	16	16 バイトのポインター

ILE C データ型と OPM COBOL/400® との互換性について、次の表で説明します。

表 25. ILE C データ型と OPM COBOL/400 との互換性

プロトタイプでの ILE C 宣言	OPM COBOL の LINKAGE SECTION	長さ	コメント
char[n] char *	PIC X(n)	n	文字の配列 (n は、1 から 3,000,000 まで)。
char	PIC 1 INDIC ..	1	標識

表 25. ILE C データ型と OPM COBOL/400 との互換性 (続き)

プロトタイプでの ILE C 宣言	OPM COBOL の LINKAGE SECTION	長さ	コメント
char[n]	PIC S9(n) USAGE IS DISPLAY	n	ゾーン 10 進数。n の限度は 18。
_Packed struct {short i; char[n]}	05 VL-FIELD. 10 i PIC S9(4) COMP-4. 10 data PIC X(n).	n+2	可変長フィールド (i は予定の長さ、n は最大長)。
char[n]	PIC X(n)	6, 8, 10	日付フィールド
char[n]	PIC X(n)	8	時刻フィールド
char[n]	PIC X(n)	26	タイム・スタンプ・フィールド
short int	PIC S9(4) COMP-4	2	2 バイトの符号付き整数。範囲は -9999 から +9999 まで。
int	PIC S9(9) COMP-4	4	4 バイトの符号付き整数。範囲は -999999999 から +999999999 まで。
long int	PIC S9(9) COMP-4	4	4 バイトの符号付き整数。範囲は -999999999 から +999999999 まで。
struct {unsigned int : n}x;	PIC 9(9) COMP-4 PIC X(4)	4	16 進数リテラルを使用して、ビット・フィールドを操作可能。
float	サポートされません。	4	4 バイトの浮動小数点
double	サポートされません。	8	8 バイトの double 型
long double	サポートされません。	8	8 バイトの long double 型
enum	サポートされません。	1, 2, 4	列挙型
*	USAGE IS POINTER	16	ポインター
decimal(n,p)	PIC S9(n-p)V9(p) COMP-3	n/2+1	パック 10 進数。n と p の限度は 18。
union.element	REDEFINES	エレメントの長さ	共用体のエレメント
data_type[n]	OCCURS	16	C がポインターを渡す先の配列
struct	01 record	n	構造体。この構造体には _Packed 修飾子を使用します。構造体の内容を変更する場合、受け渡す構造体はポインターとして構造体に渡す必要があります。
関数へのポインター	サポートされません。	16	16 バイトのポインター
サポートされません。	PIC S9(18) COMP-4	8	8 バイト整数

ILE C データ型と CL との互換性について、次の表で説明します。

表 26. ILE C データ型と CL との互換性

プロトタイプでの ILE C 宣言	CL	長さ	コメント
char[n] char *	*CHAR LEN(&N)	n	文字の配列 (n は、1 から 32766 まで)。ヌル終了ストリング。例: CHGVAR &V1 VALUE (&V *TCAT X’00’)。&V1 は、&V より 1 バイトだけ大きな値になります。n の限度は 9999。
char	*LGL	1	「1」または「0」が保持されます。
_Packed struct {short i; char[n]}	サポートされません。	n+2	可変長フィールド (i は予定の長さ、n は最大長)。
整数型	サポートされません。	1, 2, 4	1 バイト、2 バイト、または 4 バイトの符号付き/符号なし整数。
浮動小数点定数	CL 定数のみ	4	4 バイトまたは 8 バイトの浮動小数点
decimal(n,p)	*DEC	n/2+1	パック 10 進数。n の限度は 15。p の限度は 9。
union.element	サポートされません。	エレメントの長さ	共用体のエレメント
struct	サポートされません。	n	構造体。この構造体には _Packed 修飾子を使用します。
関数へのポインター	サポートされません。	16	16 バイトのポインター

コマンド行の CL 呼び出しから ILE C プログラムへ引数を渡す方法について、次の表で説明します。

表 27. コマンド行の CL 呼び出しから ILE C プログラムへ渡される引数

コマンド行の引数	Argv 配列	ILE C の引数
	argv[0]	″LIB/PGMNAME″
	argv[1..255]	通常のパラメーター
’123.4’	argv[1]	″123.4″
123.4	argv[2]	0000000123.40000D
’Hi’	argv[3]	″Hi″
Lo	argv[4]	″LO″

CL 文字配列 (ストリング) が ILE C プログラムへ渡されるときは、ヌル終了にはなりません。このような引数を制御言語プログラムから受け取る C プログラムでは、ストリングがヌル終了することは見込めません。 QCMDEXC を使用すれば、すべての引数をヌル終了させることができます。

コンパイル済みの制御言語プログラムから ILE C プログラムへ CL 定数を渡す方法について、次の表で説明します。

表 28. コンパイル済みの制御言語プログラムから ILE C プログラムへ渡される CL 定数

制御言語プログラム引数のコンパイル	Argv 配列	ILE C の引数
	argv[0]	″LIB/PGMNAME″
	argv[1..255]	通常のパラメーター

表 28. コンパイル済みの制御言語プログラムから ILE C プログラムへ渡される CL 定数 (続き)

制御言語プログラム引数のコンパイル	Argv 配列	ILE C の引数
'123.4'	argv[1]	″123.4″
123.4	argv[2]	0000000123.40000D
'Hi'	argv[3]	″Hi″
Lo	argv[4]	″LO″

コマンド処理プログラム (CPP) は、 547 ページの表 28 に定義されているように CL 定数を渡します。 ILE C プログラムを呼び出すために、コマンド作成 (CRTCMD) コマンドを使用して独自の CL コマンドを作成する場合は、ILE C プログラムをコマンド処理プログラムとして定義します。

コンパイル済みの制御言語プログラムから ILE C プログラムへ CL 変数を渡す方法について、次の表で説明します。 CL から C への引数の受け渡しは、すべて参照によって行われます。

表 29. コンパイル済みの制御言語プログラムから ILE C プログラムへ渡される CL 変数

CL 変数	ILE C の引数
DCL VAR(&v) TYPE(*CHAR) LEN(10) VALUE('123.4')	123.4
DCL VAR(&d) TYPE(*DEC) LEN(10) VALUE(123.4)	0000000123.40000D
DCL VAR(&h) TYPE(*CHAR) LEN(10) VALUE('Hi')	Hi
DCL VAR(&i) TYPE(*CHAR) LEN(10) VALUE(Lo)	LO
DCL VAR(&j) TYPE(*LGL) LEN(1) VALUE('1')	1

CL 変数と数値定数は、ヌル終了ストリングで ILE C プログラムに渡されることはありません。文字定数と論理リテラルは、ヌル終了ストリングとして受け渡されますが、ブランクで埋め込まれてはいません。パック 10 進数などの数値制約は、15,5 (8 バイト) として受け渡されます。

ランタイムの文字セット

EBCDIC CCSID は、それぞれインバリアント文字とバリアント文字の 2 つの文字タイプで構成されています。

C 文字セットにおけるインバリアント文字の 16 進表記を、次の表で紹介します。

表 30. インバリアント文字

. 0x4b	< 0x4c	(0x4d	+ 0x4e	& 0x50	* 0x5c	) 0x5d	; 0x5e
- 0x60	¦ 0x6a	, 0x6b	% 0x6c	_ 0x6d	> 0x6e	? 0x6f	: 0x7a
@ 0x7c	' 0x7d	= 0x7e	″ 0x7f	a-i 0x81 - 0x89	j-r 0x91 - 0x99	s-z 0xa2 - 0xa9	A-I 0xc1 - 0xc9
J-R 0xd1 - 0xd9	S-Z 0xe2 - 0xe9	0-9 0xf0 - 0xf9	'¥a' 0x2f	'¥b' 0x16	'¥t' 0x05	'¥v' 0x0b	'¥f' 0x0c
'¥r' 0x0d	'¥n' 0x15	' ' 0x40					

注: すべての EBCDIC 文字セットのインバリアント・コード・ポイントに、すべてのインバリアント文字が含まれているわけではありません。次のような例外があります。

- コード・ページ 290 (日本語 CCSID 290、930、および 5026 で使用) では、標準外の位置に小文字のローマ字 (a から z まで) があります。
- コード・ページ 420 (一部のアラビア語 CCSID で使用) では、16 進値が 0x7a のバック引用符 (`) がありません。
- コード・ページ 423 (一部の旧ギリシャ語 CCSID で使用) には、16 進値が 0x50 のアンパーサンド (&) がありません。
- コード・ページ 905 および 1026 (共に一部のトルコ語 CCSID で使用) の二重引用符は、インバリアント 16 進値 0x7f の代わりに 16 進値 0xfc のものを使います。

最も一般的に使用される CCSID 用の C 文字セットにおけるバリアント文字の 16 進表記について、以下の表で紹介します。

表 31. さまざまな CCSID でのバリアント文字

<table>
<tr><th>CC-
SID</th><th>|</th><th>!</th><th></th><th>\</th><th>`</th><th>#</th><th>~</th><th>[</th><th>]</th><th>^</th><th>{</th><th>}</th><th>/</th><th>¢</th><th>$</th></tr>
<tr><td>037</td><td>0x4f</td><td>0x5a</td><td>0x5f</td><td>0xe0</td><td>0x79</td><td>0x7b</td><td>0xa1</td><td>0xba</td><td>0xbb</td><td>0xb0</td><td>0xc0</td><td>0xd0</td><td>0x61</td><td>0x4a</td><td>0x5b</td></tr>
<tr><td>256</td><td>0xbb</td><td>0x4f</td><td>0xba</td><td>0xe0</td><td>0x79</td><td>0x7b</td><td>0xa1</td><td>0x4a</td><td>0x5a</td><td>0x5f</td><td>0xc0</td><td>0xd0</td><td>0x61</td><td>0xb0</td><td>0x5b</td></tr>
<tr><td>273</td><td>0xbb</td><td>0x4f</td><td>0xba</td><td>0xec</td><td>0x79</td><td>0x7b</td><td>0x59</td><td>0x63</td><td>0xfc</td><td>0x5f</td><td>0x43</td><td>0xdc</td><td>0x61</td><td>0xb0</td><td>0x5b</td></tr>
<tr><td>277</td><td>0xbb</td><td>0x4f</td><td>0xba</td><td>0xe0</td><td>0x79</td><td>0x4a</td><td>0xdc</td><td>0x9e</td><td>0x9f</td><td>0x5f</td><td>0x9c</td><td>0x47</td><td>0x61</td><td>0xb0</td><td>0x67</td></tr>
<tr><td>278</td><td>0xbb</td><td>0x4f</td><td>0xba</td><td>0x71</td><td>0x51</td><td>0x63</td><td>0xdc</td><td>0xb5</td><td>0x9f</td><td>0x5f</td><td>0x43</td><td>0x47</td><td>0x61</td><td>0xb2</td><td>0x67</td></tr>
<tr><td>280</td><td>0xbb</td><td>0x4f</td><td>0xba</td><td>0x48</td><td>0xdd</td><td>0xb1</td><td>0x58</td><td>0x90</td><td>0x51</td><td>0x5f</td><td>0x44</td><td>0x45</td><td>0x61</td><td>0xb0</td><td>0x5b</td></tr>
<tr><td>284</td><td>0x4f</td><td>0xbb</td><td>0x5f</td><td>0xe0</td><td>0x79</td><td>0x69</td><td>0xbd</td><td>0x4a</td><td>0x5a</td><td>0xba</td><td>0xc0</td><td>0xd0</td><td>0x61</td><td>0xb0</td><td>0x5b</td></tr>
<tr><td>285</td><td>0x4f</td><td>0x5a</td><td>0x5f</td><td>0xe0</td><td>0x79</td><td>0x7b</td><td>0xbc</td><td>0xb1</td><td>0xbb</td><td>0xba</td><td>0xc0</td><td>0xd0</td><td>0x61</td><td>0xb0</td><td>0x4a</td></tr>
<tr><td>297</td><td>0xbb</td><td>0x4f</td><td>0xba</td><td>0x48</td><td>0xa0</td><td>0xb1</td><td>0xbd</td><td>0x90</td><td>0x65</td><td>0x5f</td><td>0x51</td><td>0x54</td><td>0x61</td><td>0xb0</td><td>0x5b</td></tr>
<tr><td>500</td><td>0xbb</td><td>0x4f</td><td>0xba</td><td>0xe0</td><td>0x79</td><td>0x7b</td><td>0xa1</td><td>0x4a</td><td>0x5a</td><td>0x5f</td><td>0xc0</td><td>0xd0</td><td>0x61</td><td>0xb0</td><td>0x5b</td></tr>
</table>

他の IBM CCSID のバリアント文字のコーディングについて詳しくは、トピック『i5/OS のグローバリゼーション』を参照してください。

CCSID およびロケールの理解

文字および文字ストリングの CCSID

すべての文字または文字ストリングには、CCSID が関連付けられています。文字または文字ストリングの CCSID は、データの発信元によって異なります。文字または文字ストリングの CCSID には、注意を払う必要があります。また、必要なときに適切な CCSID に値が変換されることも重要です。

LOCALETYPE(*LOCALEUTF) がコンパイル・コマンドで指定されていない場合、以下のことが想定されます。

- ジョブの CCSID が、現行ロケールの LC_CTYPE カテゴリーの CCSID と同じである。
- 文字リテラル値の CCSID が、現行ロケールの LC_CTYPE カテゴリーの CCSID と同じである。
- 現行ロケールの LC_CTYPE カテゴリーの CCSID が、EBCDIC CCSID である。

- 使用される CCSID の適切な位置に、インバリアント文字のすべてが含まれている。さらに、一部の関数で、特定のバリアント文字に CCSID 37 の場合と同じ 16 進値が含まれていることが想定される。

LOCALETYPE(*LOCALEUTF) が指定された場合、ほとんどの関数 (特に指定のない限り) において、文字データのソースに関係なく、現行ロケールの LC_CTYPE カテゴリーの CCSID で文字データが入力されることが予想されます。詳細については、 557 ページの『Unicode のサポート』を参照してください。

バリアント文字とインバリアント文字の詳細については、 548 ページの『ランタイムの文字セット』を参照してください。 CCSID、コード・ページ、およびその他のグローバリゼーション概念の詳細については、トピック『i5/OS のグローバリゼーション』を参照してください。

文字リテラルの CCSID

文字リテラルの CCSID とは、コンパイル済みソース・コードにおける文字リテラルおよび文字ストリング・リテラルの CCSID のことです。プログラマーが特別の処置を行わない限り、これらのリテラルの CCSID がソース・ファイルの CCSID に設定されます。 TGTCCSID オプションをコンパイル・コマンドで使用することで、コンパイル単位のすべてのリテラルの CCSID を変更できます。 `#pragma convert` ディレクティブを使用すると、C または C++ のソース・コード内にある文字リテラルおよび文字ストリング・リテラルの CCSID を変更できます。詳細については、*IBM Rational Development Studio for i: ILE C/C++ コンパイラー参照*を参照してください。

コンパイル・コマンドで LOCALETYPE(*CLD) または LOCALETYPE(*LOCALE) を指定した場合、すべてのワイド文字リテラルがソース・ファイルの CCSID にあるワイド EBCDIC リテラルになります。コンパイル・コマンドで LOCALETYPE(*LOCALEUCS2) を指定した場合、すべてのワイド文字リテラルが UCS-2 リテラルになります。コンパイル・コマンドで LOCALETYPE(*LOCALEUTF) を指定した場合、すべてのワイド文字が UTF-32 リテラルになります。

文字リテラル値の CCSID について、プログラマーは常に意識しておく必要があります。文字リテラルの CCSID は、実行時に取得することはできません。

ジョブの CCSID

ジョブの CCSID は、常に EBCDIC CCSID になります。ジョブの CCSID では、ASCII と Unicode はサポートされません。ファイルから読み取られるデータは、ジョブの CCSID 内に置かれる場合があります。ジョブの CCSID を出力する関数 (`getenv()` など) もありますし、ジョブの CCSID の入力を要求する関数 (`putenv()` など) もあります。 C ランタイムがもっとも頻繁に使用する CCSID は、現行ロケールの LC_CTYPE カテゴリーの CCSID です。ジョブの CCSID がロケールの CCSID に一致しない場合は、変換が必要になります。

JOBI0400 レシーバーの可変長フォーマットを使用すると、ジョブの CCSID 値を実行時に取得できます (QUSRJOBI API を利用)。「デフォルトのコード化文字セット ID」フィールドには、ジョブの CCSID 値が入ります。

ファイルの CCSID

ファイルをオープンすると、そのファイルと CCSID が関連付けられます。文字とストリングの値の読み取り操作を行うと、ファイルの CCSID にデータが戻されます。ファイルに対する書き込み操作を行うと、ファイルの CCSID でデータが要求されます。ファイルのオープン時に関連付けられる CCSID は、ファイルのオープンに使用される関数によって異なります。

- `catopen` 関数

catopen を使用してオープンされるカタログ・ファイルに関連付けられる CCSID は、oflag パラメーターの内容に依存します。パラメーターに指定できるフラグは、NL_CAT_JOB_MODE と NL_CAT_CTYPE_MODE の 2 つです。これらのフラグは、相互に排他的です。

- NL_CAT_JOB_MODE を指定すると、ジョブの CCSID がファイルに関連付けられます。
- NL_CAT_CTYPE_MODE を指定すると、現行ロケールの LC_CTYPE カテゴリーの CCSID がファイルに関連付けられます。
- どちらのフラグも指定されない場合、変換は実行されず、返されるメッセージの CCSID はメッセージ・ファイルの CCSID と同じになります。

- fdopen() 関数
  - LOCALETYPE(*LOCALEUTF) を指定しない場合、ファイルのデフォルト CCSID はジョブの CCSID になります。キーワード ccsid=value、o_ccsid=value、または codepage=value を、ファイル・オープン・コマンドのモード・ストリングに使用することで、ファイルに関連付けられている CCSID を変更することができます。キーワード o_ccsid=value の使用をお勧めします。標準ファイルは、常にファイルのデフォルト CCSID に関連付けられるため、ジョブの CCSID にも関連付けられます。
  - LOCALETYPE(*LOCALEUTF) を指定した場合に fopen() 関数を呼び出すと、ファイルのデフォルト CCSID は、現行ロケールの LC_CTYPE カテゴリーの CCSID になります。前の段落で説明したキーワードを使用して、ファイルに関連付けられた CCSID を指定変更することも可能です。標準ファイルは、常にファイルのデフォルト CCSID に関連付けられるため、ファイルのオープン時に現行ロケールの LC_CTYPE カテゴリーの CCSID に関連付けられます。
- fopen() 関数および freopen() 関数
  - LOCALETYPE(*LOCALEUTF) を指定しない場合、ファイルのデフォルト CCSID はジョブの CCSID になります。
    - コンパイル・コマンドで SYSIFCOPT(*NOIFSIO) を指定した場合、キーワード ccsid=value をファイル・オープン・コマンドのモード・ストリングに使用することで、ファイルから読み取るデータまたはファイルに書き込むデータの CCSID を変更することができます。
    - コンパイル・コマンドで SYSIFCOPT(*NOIFSIO) を指定していない場合、キーワード ccsid=value、o_ccsid=value、または codepage=value をファイル・オープン・コマンドのモード・ストリングに使用することで、ファイルに関連付けられている CCSID を変更することができます。キーワード o_ccsid=value の使用をお勧めします。

    標準ファイルは、常にファイルのデフォルト CCSID に関連付けられるため、ジョブの CCSID にも関連付けられます。
  - LOCALETYPE(*LOCALEUTF) を指定した場合に fopen() 関数または freopen() 関数を呼び出すと、ファイルのデフォルト CCSID は、現行ロケールの LC_CTYPE カテゴリーの CCSID になります。キーワード ccsid=value、o_ccsid=value、または codepage=value を使用して、ファイルに関連付けられている CCSID を指定変更することも可能です。標準ファイルは、常にファイルのデフォルト CCSID に関連付けられるため、ファイルのオープン時に現行ロケールの LC_CTYPE カテゴリーの CCSID に関連付けられます。
- _Ropen() 関数

  _Ropen() 関数を使用してオープンされたファイルに関連付けられるデフォルト CCSID は、ジョブの CCSID になります。キーワード ccsid=value を _Ropen() 関数のモード・パラメーターに使用することで、ファイルに関連付けられた CCSID を変更することができます。
- wfopen() 関数
  - LOCALETYPE(*LOCALEUCS2) を指定した場合、ファイルのデフォルト CCSID は UCS-2 になります。キーワード ccsid=value、o_ccsid=value、または codepage=value を、ファイル・オープン・コマン

ドのモード・ストリングに使用することで、ファイルに関連付けられている CCSID を変更することができます。キーワード o_ccsid=value の使用をお勧めします。

– LOCALETYPE(*LOCALEUTF) を指定した場合、ファイルのデフォルト CCSID は UTF-32 になります。前の段落で説明したキーワードを使用して、ファイルに関連付けられた CCSID を指定変更することも可能です。

ロケール CCSID

ロケールの各カテゴリーに、CCSID が関連付けられます (ロケール・カテゴリーのリストは、354 ページの『setlocale() — ロケールの設定』を参照)。ロケールから最も頻繁に使用される CCSID は、ロケールの LC_CTYPE カテゴリーに関連付けられている CCSID です。異なるロケール・カテゴリーで異なる CCSID 値を使用すると、混乱が生じるおそれがあります。すべてのロケール・カテゴリーで同一の CCSID 値を使用することをお勧めします。現行ロケールの LC_CTYPE カテゴリーの CCSID を取得するには、`nl_langinfo()` 関数を使用し、nl_item として CODESET を指定します。以下は、ロケール CCSID の追加説明です。コンパイル・コマンドで指定する LOCALETYPE オプションで分類されています。

- LOCALETYPE(*CLD)

  LOCALETYPE(*CLD) は、ILE C コンパイラーでのみサポートされます。 LOCALETYPE(*CLD) を指定した場合、POSIX 関数の大半がサポートされません。 LOCALETYPE(*CLD) オプションを使用するメリットの 1 つに、すべての *CLD ロケールが CCSID 37 であることが挙げられます。 LOCALETYPE(*CLD) で使用できるシステムには、ロケール・オブジェクトが若干数添付されます。これらのオブジェクトのオブジェクト・タイプは、すべて *CLD です。 *CLD ロケール・オブジェクトのリストを取得するには、次のコマンドを使用してください。

  WRKOBJ OBJ(QSYS/*ALL) OBJTYPE(*CLD)

  *CLD ロケールの詳細については、「*IBM Rational Development Studio for i: ILE C/C++ コンパイラー参照*」を参照してください。

- LOCALETYPE(*LOCALE)

  ILE C コンパイラーおよび ILE C++ コンパイラー用の、デフォルト LOCALETYPE 設定です。通常、デフォルト・ロケール値の CCSID は、ジョブの CCSID と同じになります。この設定におけるロケール・オブジェクトは、広範囲にわたります。これらのロケール・オブジェクトが持つオブジェクト・タイプは、*LOCALE です。 LOCALETYPE(*LOCALE) オプションは、LOCALETYPE(*CLD) オプションに比べて、より多くの CCSID と関数をサポートします。

- LOCALETYPE(*LOCALEUCS2)

  この設定により、UCS-2 文字用のロケール・カテゴリーの新規セットが導入されます。これらのロケール・カテゴリー名は、LC_UNI_ サブストリングで始まります。元のロケール・カテゴリーも、引き続き存在します。LOCALETYPE(*LOCALE) に対する前述の注記は、すべて LOCALETYPE(*LOCALEUCS2) にも適用されます。この設定により、ワイド文字は、ワイド EBCDIC 文字ではなく UCS-2 文字として解釈されます。詳細については、557 ページの『Unicode のサポート』を参照してください。

- LOCALETYPE(*LOCALEUTF)

  ワイド以外のロケール・カテゴリーの CCSID は、デフォルトでは UTF-8 (CCSID 1208) になりますが、1 バイトまたはマルチバイトの CCSID を持つように変更できます。ワイド文字 (LC_UNI_*) のロケール・カテゴリーの CCSID は、UTF-32 です。この設定には、CCSID ニュートラルが限定的に組み込まれます。 LOCALETYPE(*LOCALEUTF) は、*LOCALE 型のロケール・オブジェクトを使用します。詳細については、557 ページの『Unicode のサポート』を参照してください。

ワイド文字

ILE C/C++ コンパイラーでは、以下のようなサポートを行っています。

- コンパイル・コマンドで LOCALETYPE(*CLD) または LOCALETYPE(*LOCALE) を指定している場合、ワイド文字は 2 バイトのワイド EBCDIC 文字として扱われます。
- コンパイル・コマンドで LOCALETYPE(*LOCALEUCS2) を指定している場合、ワイド文字は 2 バイトの UCS-2 文字として扱われます。
- コンパイル・コマンドで LOCALETYPE(*LOCALEUTF) を指定している場合、ワイド文字は 4 バイトの UTF-32 文字として扱われます。

EBCDIC ワイド文字を使用する場合、EBCDIC 文字の CCSID は、現行ロケールの LC_CTYPE カテゴリーの CCSID に依存します。 Unicode 文字の詳細については、 557 ページの『Unicode のサポート』を参照してください。

1 バイト文字またはマルチバイト文字とワイド文字との交互変換

文字変換ルーチンは、現行ロケールの LC_CTYPE カテゴリーの CCSID 設定を検査して、ワイド文字との交互変換が予想されるのは 1 バイト文字かマルチバイト文字かを判定します。

ワイド文字変換 (1 バイトまたはマルチバイト文字ストリングとの交互変換) の処理は、コンパイル・コマンドで指定された LOCALETYPE パラメーター値に依存します。この処理は、1 バイトまたはマルチバイトの文字ストリングのシフト状態に依存します。 `mbtowc`、`mbstowcs`、`wctomb`、および `wcstombs` の各関数では、内部のシフト状態変数が維持されます。 `mbrtowc`、`mbsrtowcs`、`wcrtomb`、および `wcsrtombs` の各関数は、シフト状態変数をパラメーターとして受け渡すことができます。 2 番目の関数セットは、汎用性が高く、スレッド・セーフでもあるため、使用することをお勧めします。

LOCALETYPE(*CLD) および LOCALETYPE(*LOCALE) の振る舞い: 1 バイト CCSID からワイド EBCDIC へ変換する場合、1 バイト文字にゼロ・バイトを追加することにより、ワイド EBCDIC 文字が構成されます。例えば、1 バイトの CCSID 37 文字である A (16 進値 0xC1) をワイド EBCDIC 文字に変換すると、16 進値は 0x00C1 になります。

マルチバイト CCSID からワイド EBCDIC へ変換する場合、変換方式は入力ストリングのシフト状態に依存します。初期シフト状態における文字は、シフトアウト文字 (16 進値の 0x0E) が読み取られるまで、1 バイト文字であるかのように読み取られます。シフトアウト文字は、2 バイト・シフト状態への移動を指示します。 2 バイト・シフト状態では、一度に 2 バイトが読み取られます。最初のバイトは EBCDIC ワイド文字の第 1 バイトになり、2 番目のバイトは EBCDIC ワイド文字の第 2 バイトになります。シフトイン文字 (16 進値の 0x0F) が登場すると、関数は初期シフト状態の構文解析に戻ります。例えば、16 進値 C10E43DA0FC2 によって表されるマルチバイト・ストリングは、16 進値 00C143DA00C2 の EBCDIC ワイド文字に変換されます。

ワイド EBCDIC を 1 バイト CCSID に変換するときに、0x00FF より大きな 16 進値が文字に含まれていると、EOF が返されます。それ以外の場合、上位バイトは切り捨てられて、下位バイトが返されます。例えば、16 進値 0x00C1 のワイド EBCDIC 文字は、16 進値 0xC1 の 1 バイト文字に変換されます。

ワイド EBCDIC からマルチバイト CCSID へ変換する場合の変換方式は、出力ストリングのシフト状態によって決定されます。

- 出力ストリングが初期シフト状態にある場合、16 進値が 0x00FF 以下の EBCDIC ワイド文字は、すべて 1 バイトに切り捨てられて出力ストリングに配置されます。
- 出力ストリングが初期シフト状態にある場合、16 進値が 0x00FF を超える値の EBCDIC ワイド文字によって、シフトアウト文字 (16 進値が 0x0E) が出力ストリングに生成されます。出力ストリングのシフト状態は 2 バイトに更新され、EBCDIC ワイド文字の両バイトは出力ストリングにコピーされます。

- 出力ストリングが 2 バイト・シフト状態にある場合に、16 進値が 0x00FF 以下の EBCDIC ワイド文字が登場すると、シフトイン文字 (16 進値が 0x0F) が出力ストリングに配置されます。シフトイン文字の後に、1 バイトに切り捨てられた EBCDIC ワイド文字の値が続きます。出力ストリングのシフト状態は、1 バイトに変更されます。
- 出力ストリングが 2 バイト・シフト状態にある場合に、値が 0x00FF を超える EBCDIC ワイド文字が登場すると、2 バイトの EBCDIC ワイド文字が出力ストリングにコピーされます。

例えば、16 進値が 00C143DA00C2 の EBCDIC ワイド文字ストリングは、16 進値が C10E43DA0FC2 のマルチバイト・ストリングに変換されます。

LOCALETYPE(*LOCALEUCS2) および LOCALETYPE(*LOCALEUTF) の振る舞い: コンパイル・コマンドで LOCALETYPE(*LOCALEUCS2) を指定した場合、ワイド文字値は 2 バイトの UCS-2 文字値になります。 UCS-2 ストリングと、1 バイトまたはマルチバイトのストリング間のすべての変換は、`iconv()` 関数を使用した場合と同様に行われます。 UCS-2 ストリングには、CCSID 13488 が使用され、1 バイトまたはマルチバイトのストリングには、現行ロケールの LC_CTYPE カテゴリーの CCSID が使用されます。

コンパイル・コマンドで LOCALETYPE(*LOCALEUTF) を指定した場合、ワイド文字値は 4 バイトの UTF-32 文字値になります。 UTF-32 ストリングと、1 バイトまたはマルチバイト・ストリング間のすべての変換は、`iconv()` 関数を使用した場合と同様に行われます。 `iconv()` 関数では、UTF-32 はサポートされません。したがって、UTF-32 ストリングと 1 バイトまたはマルチバイトのストリング間での変換では、仲介のデータ型として UTF-16 (CCSID 1200) が使用されます。 UTF-32 と UTF-16 間の変換は、QlgTransformUCSData() API を使用して行われます。 UTF-16 と、現行ロケールの LC_CTYPE カテゴリーの CCSID との間の変換には、iconv() API が使用されます。

ワイド文字とファイル入出力

ワイド文字の書き込みルーチン: `fwprintf`、`vwprintf`、`vfwprintf`、`wprintf`、`fputwc`、`fputws`、`putwc`、`putwchar`、`ungetwc` などのルーチンは、ワイド文字をファイルに書き込む目的に使用できます。 LOCALETYPE(*CLD) または SYSIFCOPT(*NOIFSIO) のいずれかをコンパイル・コマンドで指定している場合は、これらのルーチンを使用することはできません。

コンパイル・コマンドで LOCALETYPE(*LOCALE) を指定した場合、書き込まれるワイド文字は、ファイルの CCSID におけるコード・ポイントのワイド文字に相当すると見なされます。ファイルの CCSID は、1 バイトまたはマルチバイトの EBCDIC CCSID と見なされます。

LOCALETYPE(*LOCALEUCS2) または LOCALETYPE(*LOCALEUTF) をコンパイル・コマンドで指定した場合、書き込まれるワイド文字は Unicode 文字と見なされます。 LOCALETYPE(*LOCALEUCS2) の場合は、2 バイトの UCS-2 文字と見なされます。 LOCALETYPE(*LOCALEUTF) の場合は、4 バイトの UTF-32 文字と見なされます。書き込み先のファイルが標準ファイルのいずれかでない場合、バイナリー・モードでの書き込み用にファイルがオープンされるように見えますが、そのファイルには Unicode 文字が直接書き込まれます。ファイルの CCSID は、ロケール設定に一致する Unicode CCSID であると見なされます。書き込み先のファイルが標準ファイルである場合、Unicode 入力はジョブの CCSID に変換されて、そのファイルに書き込まれます。

ワイド文字以外の書き込みルーチン: ワイド文字以外の書き込みルーチン (`fprintf`、`vfprintf`、`vprintf`、および `printfcan`) は、ワイド文字を入力データとして取り込むことができます。

すべての場合において、wctomb 関数または wcstombs 関数を使用した場合と同様に、ワイド文字が現行ロケールの LC_CTYPE カテゴリーの CCSID でマルチバイト文字ストリングに変換されます。ファイルの CCSID は、現行ロケールの LC_CTYPE カテゴリーの CCSID に一致すると見なされます。

コンパイル・コマンドで LOCALETYPE(*LOCALEUTF) を指定した場合、書き込み先のファイルが標準ファイルであれば、現行ロケールの LC_CTYPE カテゴリーの CCSID から、そのファイルの CCSID (通常は、ジョブの CCSID と一致) へ、出力が自動的に変換されます。

ワイド文字の読み取りルーチン: ファイルからワイド文字を読み取ることができるルーチンには、fgetwc、fgetws、fwscanf、getwc、getwchar、vfwscanf、vwscanf、wscanf などがあります。LOCALETYPE(*CLD) または SYSIFCOPT(*NOIFSIO) のいずれかをコンパイル・コマンドで指定している場合は、これらのルーチンを使用することはできません。

コンパイル・コマンドで LOCALETYPE(*LOCALE) を指定している場合、ファイルから読み取られるワイド文字は、ファイルの CCSID におけるコード・ポイントに相当する EBCDIC ワイド文字であると見なされます。

LOCALETYPE(*LOCALEUCS2) または LOCALETYPE(*LOCALEUTF) をコンパイル・コマンドで指定している場合、入力されたワイド文字とファイル内の文字は Unicode 文字と見なされます。LOCALETYPE(*LOCALEUCS2) の場合は、2 バイトの UCS-2 文字と見なされます。LOCALETYPE(*LOCALEUTF) の場合は、4 バイトの UTF-32 文字と見なされます。読み取るファイルが標準ファイルのいずれかでない場合、そのファイルがバイナリー・モードでオープンされるように見えますが、そのファイルからは Unicode 文字が直接読み取られます。ファイルの CCSID は、ロケール設定に一致する Unicode CCSID であると見なされます。読み取るファイルが標準ファイルである場合、そのファイルから読み取られるジョブの CCSID 入力は、該当する Unicode CCSID に変換されます。

ワイド文字以外の読み取りルーチン: ワイド文字以外の読み取りルーチン (fscanf、scanf、vfscanf、および vscanf) は、ワイド文字を出力として生成できます。

すべての場合において、mbtowc 関数または mbstowcs 関数を使用した場合と同様に、現行ロケールの LC_CTYPE カテゴリーの CCSID のマルチバイト文字ストリングから、ロケール設定の該当するワイド文字タイプへ、ワイド文字が変換されます。

非同期シグナル・モデル

C モジュール作成 (CRTCMOD) または結合 C プログラム作成 (CRTBNDC) のコンパイル・オプションに、SYSIFCOPT(*ASYNCSIGNAL) オプションを指定した場合、非同期シグナル・モデル (ASM) が使用されます。 C++ モジュール作成 (CRTCPPMOD) または結合 C++ プログラム作成 (CRTBNDCPP) のコンパイル・コマンドに、RTBND(*LLP64) オプションを指定した場合にも、ASM が使用されます。これは、UNIX オペレーティング・システムからインポートされたアプリケーションとの互換性を確保するためのものです。 ASM を使用するモジュールの場合、signal() 関数と raise() 関数は、i5/OS シグナル API (詳細は、i5/OS Information Center の見出し『プログラミング』の下にあるトピック『アプリケーション・プログラミング・インターフェース』を参照) を使用して、インプリメントされます。

ASM のモジュールまたはプログラムに送られた i5/OS 例外は、非同期シグナルに変換されます。この例外は、非同期シグナル・ハンドラーによって処理されます。

ASM を使用するためにコンパイルされたモジュールは、同一のプロセス、プログラム、およびサービス・プログラム内でオリジナル・シグナル・モデル (OSM) を使用するモジュールと混合させることができます。この 2 つのシグナル・モデル間には、以下のような違いがあります。

- OSM は例外をベースにしていますが、ASM は非同期シグナルをベースにしています。
- OSM におけるシグナル・ベクトルおよびシグナル・マスクは、活動化グループを対象とします。 ASM では、プロセスごとにシグナル・ベクトルが 1 つ、スレッドごとにシグナル・マスクが 1 つ、それぞれ存在します。両タイプのシグナル・ベクトルとシグナル・マスクは、実行時に維持されます。
- 同一の非同期のシグナル・ベクトルおよびシグナル・マスクが、それぞれが属する活動化グループに関係なく、スレッド内のすべての ASM モジュールによって操作されます。シグナル・ベクトルとシグナル・マスクの状態を保存および復元して、ASM モジュールによって変更されないようにする必要があります。 OSM では、非同期のシグナル・ベクトルおよびシグナル・マスクを使用しません。
- OSM モジュールが発したシグナルは、例外として送信されます。 OSM では、`_C_exception_router` 関数 (ユーザーの OSM シグナル・ハンドラーを直接呼び出します) によって、例外の受信および処理が行われます。

  非同期シグナルは、例外にマップされません。さらに、OSM 下に登録されているシグナル・ハンドラーによる処理も行われません。 ASM では、`_C_async_exception_router` 関数 (例外を非同期シグナルにマップします) によって、例外の受信および処理が行われます。 ASM シグナル・ハンドラーは、i5/OS 非同期シグナル・コンポーネントからコントロールを受け取ります。

  OSM モジュールがシグナルを発すると、生成された例外が、例外モニターを見つけるまで呼び出しスタックをパーコレートします。前の呼び出しが OSM 関数であった場合、`_C_exception_router` が例外をキャッチして、OSM シグナル・アクションを実行します。 ASM シグナル・ハンドラーは、シグナルを受信しません。

  前の呼び出しが ASM 関数であった場合、`_C_async_exception_router` が例外を処理して、非同期シグナルにマップします。その場合、非同期シグナルの処理は、トピック『i5/OS シグナル管理』に定義されているように、スレッドの非同期シグナルのベクトルおよびマスクの状態に応じて異なります。

  前の呼び出しが別のプログラムまたはサービス・プログラム内にある ASM 関数であった場合、2 つのアクションのいずれかが発生します。シグナルを発する OSM プログラムが、ASM プログラムと共に同一の活動化グループ内で実行されている場合、前述のマッピングを使用して、例外が非同期シグナルにマップされます。例外がシグナルにマップされる場合も、シグナル ID は保持されます。したがって、非同期シグナル・モデルで登録されたシグナル・ハンドラーでも、オリジナル・シグナル・モデル下で生成されたシグナルを受信できます。このアプローチを使用して、2 つのプログラムをさまざまなシグナル・モデルと統合することができます。

  シグナルを発する OSM プログラムが、ASM プログラムとは異なる活動化グループ内で実行されている場合、その活動化グループ内でモニターされないシグナルがあれば、そのシグナルによってプログラムおよび活動化グループが終了します。その場合、モニターされないシグナルは、CEE9901 例外として呼び出し側プログラムへパーコレートされます。 CEE9901 例外は、SIGSEGV 非同期シグナルへマップされます。
- ASM では、C 関数の `raise()` および `signal()` は、`kill()` や `sigaction()` などのシステム・シグナル関数と統合されます。これら 2 セットの API は、同じように使用することができます。 OSM では、このようなことはできません。
- `#pragma exception_handler` によって設定された、ユーザー指定の例外モニターは、コンパイラー生成モニター (`_C_async_exception_router` を呼び出す) よりも高い優先順位を持ちます。ある状態においては、この優先順位により、コンパイラー生成モニター (`_C_async_exception_router` を呼び出す) を迂回させることができます。
- ASM において、シグナルに関連付けられた例外 ID を取得する際に、`_GetExcData()` 関数を使用することはできません。ただし、sigaction() 関数によって拡張シグナル・ハンドラーが設定されている場合は、

シグナル固有のデータ構造から例外情報にアクセスすることができます。詳細については、 161 ページの『_GetExcData() ― 例外データの取得』を参照してください。

Unicode のサポート

Unicode 標準は、標準化文字コードの一種で、各国のテキストを表示および保管用にエンコードする目的で設計されています。固有の 16 ビット値または 32 ビット値を使用して、プラットフォームや言語、プログラムなどに依存せず、それぞれの文字を個別に表します。 Unicode を使用すると、さまざまなプラットフォームや言語、国、地域で稼働するソフトウェア製品を開発することができます。また、Unicode を使用することにより、複数の異なるシステムを介してデータを転送することもできます。

コンパイラーおよびランタイムから使用できる Unicode サポートには、2 種類の形式があります。このセクションでは、この 2 種類の Unicode サポートについて説明すると共に、各サポートのフィーチャーや、サポート使用時の考慮事項についても触れます。 Unicode 関する追加情報を参照するには、Unicode ホーム・ページ (英語) (www.unicode.org) にアクセスしてください。

1 つ目の Unicode サポートは、UCS-2 サポートです。コンパイル・コマンドで LOCALETYPE (*LOCALEUCS2) オプションを指定すると、コンパイラーおよびランタイムは、2 バイトの Unicode 文字を表すワイド文字 (wchar_t タイプの文字) およびワイド文字ストリング (wchar_t * タイプのストリング) を使用します。ナロー (非ワイド) 文字およびナロー文字ストリングは、UCS-2 サポートが有効でない場合と同様に、EBCDIC 文字を表します。 Unicode 文字は、CCSID 13488 でコード・ポイントを表します。

2 番目のタイプの Unicode サポートは、UTF-8/UTF-32 サポートです (UTF サポートとも呼ばれます)。コンパイル・コマンドで LOCALETYPE(*LOCALEUTF) オプションを指定すると、コンパイラーおよびランタイムは、4 バイトの Unicode 文字を表すワイド文字およびワイド文字ストリングを使用します。各 4 バイト文字は、UTF-32 の 1 文字を表します。ナロー文字およびナロー文字ストリングは、UTF-8 文字を表します。各 UTF-8 文字のサイズは、1 バイトのものから 4 バイトのものまであります。通常文字の大部分は、1 バイトのサイズです。実際には、7 ビットの ASCII 文字は、すべて UTF-8 に直接マップされて、そのサイズは 1 バイトになります。 UTF-8 文字は、CCSID 1208 でコード・ポイントを表します。

UTF サポートを有効にすると、ワイド文字が UTF-32 Unicode になるばかりでなく、ナロー文字も UTF-8 Unicode になります。次の HelloWorld プログラムを例に、説明します。

```
#include <stdio.h>

int main() {
   printf("Hello World\n");
   return 0;
}
```

このプログラムを UTF サポートによってコンパイルすると、EBCDIC 文字ではなく、UTF-8 文字として、文字ストリングがプログラム内に格納されます。 `printf()` 関数はこのことを認識し、UTF-8 文字を構文解析して、意図に沿った出力を生成します。しかし、UTF-8 文字の処理方法を理解しないユーザー提供ルーチンなどが、このプログラムによって呼び出された場合、他のルーチンでも誤った結果が出力されるか、誤った振る舞いが行われることになります。

Unicode サポートを使用する理由

アプリケーションで Unicode サポートを使用すべき状態は、2 つあります。 1 つ目の状態は、該当するアプリケーションが多国語アプリケーションであり、複数の異なる言語のサポートが必要となる場合です。 Unicode 文字セットを使えば、さまざまな言語または文字セットを、単一のアプリケーションで容易に処理できるようになります。 Unicode 文字を使用することで、すべての入力、処理、および出力をアプリケー

ションで実行することができます。 Unicode サポートを使用すべきもう 1 つの状態は、7 ビットの ASCII アプリケーションを移植する場合です。 UTF-8 文字セットは、7 ビット ASCII のスーパーセットです。したがって、ASCII アプリケーションを UTF-8 環境へ移植する際には、EBCDIC 環境へ移植する場合よりも容易に行うことができます。

疑似 CCSID ニュートラル

UTF サポートを使用してプログラムをコンパイルすると、ランタイムにより UTF-8 文字を超えるものが容認され、これにより実質的な CCSID ニュートラル が発生します。現行ロケール内に含まれるどのような CCSID であっても、ランタイムは処理します。デフォルトでは、UTF サポートを使用してプログラムをコンパイルすると、ロードされるロケールは UTF-8 (CCSID 1208) ロケールになります。これにより、ランタイムで CCSID 1208 を処理できるようになります。 setlocale() ルーチンを呼び出して、ロケールを EBCDIC ロケール (例えば、CCSID 37 ロケール) に設定すると、ランタイムが CCSID 37 を処理します。これによって、コンパイラー内での #pragma convert サポートに加えて、多国語アプリケーションのサポートを提供することができます。以下は、サンプルです。

```
#include <stdio.h>
#include <locale.h>

int main() {
   /* This string is in CCSID 1208 */
   printf("Hello World\n");

   /* Change locale to a CCSID 37 locale */
   setlocale(LC_ALL, "/QSYS.LIB/EN_US.LOCALE");
   #pragma convert(37)

   /* This string is in CCSID 37 */
   printf("Hello World\n");

   return 0;
}
```

他の ILE 言語からの Unicode

適切なヘッダー・ファイルをインクルードし、適切な LOCALETYPE オプションを C または C++ のコンパイル・コマンドに使用することで、C 言語および C++ 言語で Unicode ルーチンに簡単にアクセスすることができます。 RPG、COBOL、CL など、他の ILE 言語のヘッダー・ファイルが提供されていなくても、これらの言語から Unicode ルーチンにアクセスすることができます。

UCS-2 サポート用に追加されたルーチンを、次の表で紹介します。これらのサポート・ルーチンでは、接頭部 _UCS2_ または _C_UCS2_ が、標準のルーチン名に追加されています。 Unicode ルーチンには、標準 (非 Unicode) ルーチンと同じパラメーターがあります。

_C_UCS2_btowc
_C_UCS2_fgetwc
_C_UCS2_fgetws
_C_UCS2_fprintf
_C_UCS2_fputwc
_C_UCS2_fputws
_C_UCS2_fscanf
_C_UCS2_fwprintf
_C_UCS2_fwscanf
_C_UCS2_getwc
_C_UCS2_getwchar
_C_UCS2_iswalnum
_C_UCS2_iswalpha
_C_UCS2_iswcntrl
_C_UCS2_iswctype
_C_UCS2_iswdigit
_C_UCS2_iswgraph
_C_UCS2_iswlower
_C_UCS2_iswprint
_C_UCS2_iswpunct
_C_UCS2_iswspace
_C_UCS2_iswupper
_C_UCS2_iswxdigit
_C_UCS2_mblen
_C_UCS2_mbrlen
_C_UCS2_mbrtowc
_C_UCS2_mbsinit
_C_UCS2_mbsrtowcs
_C_UCS2_printf
_C_UCS2_putwc
_C_UCS2_putwchar
_C_UCS2_scanf
_C_UCS2_snprintf
_C_UCS2_sprintf
_C_UCS2_sscanf
_C_UCS2_swprintf
_C_UCS2_swscanf
_C_UCS2_towlower
_C_UCS2_towupper
_C_UCS2_ungetwc
_C_UCS2_vfprintf
_C_UCS2_vfscanf
_C_UCS2_vfwprintf
_C_UCS2_vfwscanf
_C_UCS2_vprintf
_C_UCS2_vscanf
_C_UCS2_vsnprintf
_C_UCS2_vsprintf
_C_UCS2_vsscanf
_C_UCS2_vswprintf
_C_UCS2_vswscanf
_C_UCS2_vwprintf
_C_UCS2_vwscanf
_C_UCS2_wcsftime
_C_UCS2_wcsicmp
_C_UCS2_wcslocaleconv
_C_UCS2_wcsnicmp
_C_UCS2_wcsrtombs
_C_UCS2_wcstod32
_C_UCS2_wcstod64
_C_UCS2_wcstod128
_C_UCS2_wctob
_C_UCS2_wprintf
_C_UCS2_wscanf
_UCS2_mbstowcs
_UCS2_mbtowc
_UCS2_setlocale
_UCS2_wcrtomb
_UCS2_wcstod
_UCS2_wcstol
_UCS2_wcstoll
_UCS2_wcstombs
_UCS2_wcstoul
_UCS2_wcstoull
_UCS2_wcswidth
_UCS2_wctomb
_UCS2_wcwidth

C コンパイラーまたは C++ コンパイラーのいずれかで LOCALETYPE(*LOCALEUCS2) オプションを使用すると、デフォルトの UCS-2 ロケールがプログラムの開始時にロードされます。前述の表にある Unicode ルーチンのいずれかを異なる言語から使用する場合は、デフォルトの UCS2 ロケールが確実にロードされるように、アプリケーションの開始時に _UCS2_setlocale(LC_ALL, "") に対する呼び出しを追加する必要があります。

CCSID ニュートラルおよび UTF-8 サポート用に追加されるルーチンを、次の表で紹介します。これらのルーチンでは、接頭部 _C_NEU_DM_ (データ管理入出力関数用)、または _C_NEU_IFS_、_C_UTF_IFS (IFS 入出力関数用)、または _C_NEU_、_C_UTF_ が、標準のルーチン名に追加されています。 Unicode ルーチンには、標準 (非 Unicode) ルーチンと同じパラメーターがあります。

ワイド文字で作動するルーチンの接頭部は、UTF になります。ワイド文字で作動しないルーチンの接頭部は、NEU になります。

_C_NEU_asctime
_C_NEU_asctime_r
_C_NEU_atof
_C_NEU_atoi
_C_NEU_catopen
_C_NEU_ctime
_C_NEU_ctime_r
_C_NEU_ctime64
_C_NEU_ctime64_r
_C_NEU_DM_clearerr
_C_NEU_DM_feof
_C_NEU_DM_ferror
_C_NEU_DM_fflush
_C_NEU_DM_fgetc
_C_NEU_DM_fgetpos
_C_NEU_DM_fgets
_C_NEU_DM_fopen
_C_NEU_DM_fprintf
_C_NEU_DM_fputc
_C_NEU_DM_fputs
_C_NEU_DM_fread
_C_NEU_DM_freopen
_C_NEU_DM_fscanf
_C_NEU_DM_fseek
_C_NEU_DM_fsetpos
_C_NEU_DM_ftell
_C_NEU_DM_fwrite
_C_NEU_DM_getc
_C_NEU_DM_getchar
_C_NEU_DM_gets
_C_NEU_DM_perror
_C_NEU_DM_printf
_C_NEU_DM_putc
_C_NEU_DM_putchar
_C_NEU_DM_puts
_C_NEU_DM_remove
_C_NEU_DM_rename
_C_NEU_DM_rewind
_C_NEU_DM_ropen
_C_NEU_DM_scanf
_C_NEU_DM_setbuf
_C_NEU_DM_setvbuf
_C_NEU_DM_tmpfile
_C_NEU_DM_tmpnam
_C_NEU_DM_ungetc
_C_NEU_DM_vfprintf
_C_NEU_DM_vfscanf
_C_NEU_DM_vprintf
_C_NEU_DM_vscanf
_C_NEU_gcvt
_C_NEU_gmtime
_C_NEU_gmtime_r
_C_NEU_IFS_clearerr
_C_NEU_IFS_fdopen
_C_NEU_IFS_feof
_C_NEU_IFS_ferror
_C_NEU_IFS_fflush
_C_NEU_IFS_fgetc
_C_NEU_IFS_fgetpos
_C_NEU_IFS_fgetpos64
_C_NEU_IFS_fgets
_C_NEU_IFS_fopen
_C_NEU_IFS_fopen64
_C_NEU_IFS_fprintf
_C_NEU_IFS_fputc
_C_NEU_IFS_fputs
_C_NEU_IFS_fread
_C_NEU_IFS_freopen
_C_NEU_IFS_freopen64
_C_NEU_IFS_fscanf
_C_NEU_IFS_fseek
_C_NEU_IFS_fseeko
_C_NEU_IFS_fseeko64
_C_NEU_IFS_fsetpos
_C_NEU_IFS_fsetpos64
_C_NEU_IFS_ftell
_C_NEU_IFS_ftello
_C_NEU_IFS_ftello64
_C_NEU_IFS_fwrite
_C_NEU_IFS_getc
_C_NEU_IFS_getchar
_C_NEU_IFS_gets
_C_NEU_IFS_perror
_C_NEU_IFS_printf
_C_NEU_IFS_putc
_C_NEU_IFS_putchar
_C_NEU_IFS_puts
_C_NEU_IFS_remove
_C_NEU_IFS_rename_keep
_C_NEU_IFS_rename_unlink
_C_NEU_IFS_rewind
_C_NEU_IFS_scanf
_C_NEU_IFS_setbuf
_C_NEU_IFS_setvbuf
_C_NEU_IFS_tmpfile
_C_NEU_IFS_tmpfile64
_C_NEU_IFS_tmpnam
_C_NEU_IFS_ungetc
_C_NEU_IFS_vfprintf
_C_NEU_IFS_vfscanf
_C_NEU_IFS_vprintf
_C_NEU_IFS_vscanf
_C_NEU_isalnum
_C_NEU_isalpha
_C_NEU_iscntrl
_C_NEU_isdigit
_C_NEU_isgraph
_C_NEU_islower
_C_NEU_isprint
_C_NEU_ispunct
_C_NEU_isspace
_C_NEU_isupper
_C_NEU_isxdigit
_C_NEU_itoa
_C_NEU_localeconv
_C_NEU_localtime
_C_NEU_localtime_r
_C_NEU_localtime64
_C_NEU_localtime64_r
_C_NEU_ltoa
_C_NEU_memicmp
_C_NEU_mktime
_C_NEU_mktime64
_C_NEU_nl_langinfo
_C_NEU_snprintf
_C_NEU_sprintf
_C_NEU_sscanf
_C_NEU_strcasecmp
_C_NEU_strchr
_C_NEU_strcspn
_C_NEU_strerror
_C_NEU_strfmon
_C_NEU_strftime
_C_NEU_strncasecmp
_C_NEU_strpbrk
_C_NEU_strptime
_C_NEU_strrchr
_C_NEU_strspn
_C_NEU_strtod
_C_NEU_strtod32
_C_NEU_strtod64
_C_NEU_strtod128
_C_NEU_strtof
_C_NEU_strtok
_C_NEU_strtok_r
_C_NEU_strtol
_C_NEU_strtold
_C_NEU_strtoll
_C_NEU_strtoul
_C_NEU_strtoull
_C_NEU_system
_C_NEU_toascii
_C_NEU_tolower
_C_NEU_toupper
_C_NEU_ultoa
_C_NEU_vsnprintf
_C_NEU_vsprintf
_C_NEU_vsscanf
_C_NEU_wctrans
_C_NEU_wctype
_C_UTF_btowc
_C_UTF_IFS_fgetwc
_C_UTF_IFS_fgetws
_C_UTF_IFS_fputwc
_C_UTF_IFS_fputws
_C_UTF_IFS_fwprintf
_C_UTF_IFS_fwscanf
_C_UTF_IFS_getwc
_C_UTF_IFS_getwchar
_C_UTF_IFS_putwc
_C_UTF_IFS_putwchar
_C_UTF_IFS_ungetwc
_C_UTF_IFS_vfwprintf
_C_UTF_IFS_vfwscanf

```
_C_UTF_IFS_vwprintf
_C_UTF_IFS_vwscanf
_C_UTF_IFS_wfopen
_C_UTF_IFS_wfopen64
_C_UTF_IFS_wprintf
_C_UTF_IFS_wscanf
_C_UTF_isalnum
_C_UTF_isalpha
_C_UTF_isascii
_C_UTF_iscntrl
_C_UTF_isdigit
_C_UTF_isgraph
_C_UTF_islower
_C_UTF_isprint
_C_UTF_ispunct
_C_UTF_isspace
_C_UTF_isupper
_C_UTF_iswalnum
_C_UTF_iswalpha
_C_UTF_iswcntrl
_C_UTF_iswctype
_C_UTF_iswdigit
_C_UTF_iswgraph
_C_UTF_iswlower
_C_UTF_iswprint
_C_UTF_iswpunct
_C_UTF_iswspace
_C_UTF_iswupper
_C_UTF_iswxdigit
_C_UTF_isxdigit
_C_UTF_mblen
_C_UTF_mbrlen
_C_UTF_mbrtowc
_C_UTF_mbsinit
_C_UTF_mbsrtowcs
_C_UTF_mbstowcs
_C_UTF_mbtowc
_C_UTF_regcomp
_C_UTF_regerror
_C_UTF_regexec
_C_UTF_setlocale
_C_UTF_strcoll
_C_UTF_strxfrm
_C_UTF_swprintf
_C_UTF_swscanf
_C_UTF_toascii
_C_UTF_tolower
_C_UTF_toupper
_C_UTF_towctrans
_C_UTF_towlower
_C_UTF_towupper
_C_UTF_vswprintf
_C_UTF_vswscanf
_C_UTF_wcrtomb
_C_UTF_wcscat
_C_UTF_wcschr
_C_UTF_wcscmp
_C_UTF_wcscoll
_C_UTF_wcscpy
_C_UTF_wcscspn
_C_UTF_wcsfmon
_C_UTF_wcsftime
_C_UTF_wcsicmp
_C_UTF_wcslen
_C_UTF_wcslocaleconv
_C_UTF_wcsncat
_C_UTF_wcsncmp
_C_UTF_wcsncpy
_C_UTF_wcsnicmp
_C_UTF_WCS_nl_langinfo
_C_UTF_wcspbrk
_C_UTF_wcsptime
_C_UTF_wcsrchr
_C_UTF_wcsrtombs
_C_UTF_wcsspn
_C_UTF_wcsstr
_C_UTF_wcstod
_C_UTF_wcstod32
_C_UTF_wcstod64
_C_UTF_wcstod128
_C_UTF_wcstok
_C_UTF_wcstol
_C_UTF_wcstoll
_C_UTF_wcstombs
_C_UTF_wcstoul
_C_UTF_wcstoull
_C_UTF_wcswcs
_C_UTF_wcswidth
_C_UTF_wcsxfrm
_C_UTF_wctob
_C_UTF_wctomb
_C_UTF_wcwidth
_C_UTF_wmemchr
_C_UTF_wmemcmp
_C_UTF_wmemcpy
_C_UTF_wmemmove
_C_UTF_wmemset
```

C コンパイラーまたは C++ コンパイラーのいずれかで LOCALETYPE(*LOCALEUTF) オプションを使用すると、デフォルトの UTF ロケールはプログラム開始処理時にロードされます。前述の表にある Unicode ルーチンのいずれかを異なる言語から使用する場合は、デフォルトの UTF ロケールが確実にロードされるように、アプリケーションの開始時に _C_UTF_setlocale(LC_ALL, "") に対する呼び出しを追加する必要があります。

標準ファイル

UTF サポートを使用する場合、デフォルトの標準入出力ファイル stdin、stdout、および stderr には、ランタイムによって特殊な処理が行われます。 UTF サポートを使用するプログラムには UTF-8 のデータが含まれており、スクリーン・ファイルおよびスプール・ファイルと標準ファイルが対話するため、データの不一致が起こる可能性があります。スクリーン・ファイルおよびスプール・ファイルのルーチンは、オペレーティング・システムが提供するため、EBCDIC が要求されます。 stdout と stderr の場合、ランタイムによって UTF-8 データが EBCDIC に自動的に変換されます。 stdin の場合、ランタイムによって着信 EBCDIC が UTF-8 データに自動的に変換されます。

考慮事項

i5/OS のデフォルト環境は主として EBCDIC 環境であるため、アプリケーションで UTF サポートを使用する場合は、このトピックに記載されている状態について熟知しておく必要があります。

UTF サポートを使用してコンパイルしたモジュールと、UTF サポートを使用しないでコンパイルしたモジュールが、プログラムまたはサービス・プログラムに含まれている場合、予期しない不一致が起こらないように注意を払う必要があります。ワイド文字とワイド文字ストリングのサイズは、非 UTF モジュールの場合は 2 バイトになり、UTF モジュールの場合には 4 バイトになります。したがって、モジュール間でワイド文字を共用した場合、正常に処理されないおそれがあります。ナロー (非ワイド) 文字および文字ストリングは、非 UTF モジュールの場合はジョブの CCSID で表記され、UTF モジュールの場合は CCSID 1208 で表記されます。したがって、モジュール間でナロー文字を共用すると、いずれも正常に処理されないおそれがあります。

setlocale() を実行して、ロケールを別の CCSID に設定するときは、必ず標準出力ファイルをフラッシュして、複数の CCSID を含む文字データによるバッファリング問題を回避する必要があります。 stdout は、デフォルトではライン・バッファーであるため、各出力行が改行文字で終了すれば問題は発生しません。しかし、そのようになっていない場合には、出力が意図どおりに表示されないおそれがあります。この問題について、以下のサンプルで説明します。

```
#include <stdio>
#include <locale.h>

int main() {
   /* This string is in CCSID 1208 */
   printf("Hello World");

   /* Change locale to a CCSID 37 locale */
   setlocale(LC_ALL, "/QSYS.LIB/EN_US.LOCALE");
   #pragma convert(37)

   /* This string is in CCSID 37 */
   printf("Hello World\n");

   return 0;
}
```

この場合、最初の printf() により、CCSID 1208 のストリング 『Hello World』 が stdout バッファーへコピーされます。 setlocale() が実行される前に、stdout がフラッシュされて、そのストリングが画面にコピーされます。 2 番目の printf() により、CCSID 37 のストリング 『Hello World¥n』 が stdout バッファーへコピーされます。末尾にある改行文字によって、バッファーはその時点でフラッシュされ、全バッファーが画面にコピーされます。現行ロケールの CCSID は 37 であり、この画面で問題を起こすことなく CCSID 37 を処理できるため、全バッファーは変換なしでコピーされます。 CCSID 1208 の文字は、読めない文字として表示されます。フラッシュが実行済みであれば、CCSID 1208 の文字が CCSID 37 に変換され、正常に表示されます。

ほぼすべてのランタイム・ルーチンは UTF をサポートするように変更されていますが、そうでないルーチンも若干存在します。例外処理を扱うルーチンおよび構造体 (_GetExcData() 関数、_EXCP_MSGID 変数、例外ハンドラー構造体 _INTRPT_Hndlr_Parms_T など) は、ランタイムではなく、オペレーティング・システムによって提供されます。これらは、完全に EBCDIC です。 getenv() 関数と putenv() 関数は、EBCDIC のみを処理します。 QXXCHGDA() 関数と QXXRTVDA() 関数は、EBCDIC のみを処理します。 argv パラメーターおよび envp パラメーターも、EBCDIC のみです。

レコード入出力ルーチン (つまり、_R で始まる関数) は、UTF を完全にサポートしているわけではありません。 UTF をサポートしないルーチンには、_Rformat()、_Rcommit()、_Racquire()、_Rrelease()、

_Rpgmdev()、_Rindara()、_Rdevatr() などがあります。 UTF オプションによってコンパイルする際に、これらを使用することはできますが、受け付けおよび生成可能なものは EBCDIC のみになります。さらに、_R 関数によって返される構造体内の文字データは、UTF ではなく EBCDIC となります。

他のオペレーティング・システム・ルーチンでは、UTF をサポートするための変更は行われていません。例えば、open() などの統合ファイル・システム・ルーチンでは、引き続きジョブの CCSID を使用できます。その他のオペレーティング・システム API でも、引き続きジョブの CCSID を使用できます。 UTF アプリケーションの場合、これらのルーチンに提供される文字および文字ストリングを、QTQCVRT、iconv()、#pragma convert などの方法を使用して、ジョブの CCSID に変換する必要があります。

ファイルのデフォルト CCSID

fopen() 関数を使用してファイルをオープンする場合、ファイルのデフォルト CCSID は、UTF サポートを使用するかどうかに応じて異なります。 UTF サポートを使用しない場合 (つまり、LOCALETYPE(*CLD)、LOCALETYPE(*LOCALE)、または LOCALETYPE(*LOCALEUCS2) をコンパイル・コマンドで指定している場合) は、ファイルのデフォルト CCSID は現行ジョブの CCSID になります。通常、ジョブの CCSID が正常に設定されれば、この CCSID に合わせて現行ロケールが設定されるため、これで問題はありません。

UTF サポートを使用しても、システムの制約があるためジョブの CCSID を UTF-8 に設定することはできません。 LOCALETYPE(*LOCALEUTF) を指定すると、ファイルのデフォルト CCSID は現行ロケールの CCSID になります。デフォルト・ロケールを使用する場合、デフォルト CCSID は UTF-8 (CCSID 1208) になります。このデフォルトを使用したくない場合は、キーワード *ccsid* または *o_ccsid* を、fopen() 呼び出しの第 2 パラメーターに指定することができます。ただし、DB2® for i5/OS では UTF-8 を完全にサポートしていないため、データベース・ファイルは例外になります。 SYSIFCOPT(*NOIFSIO) を指定した場合に、現行ロケールの CCSID が 1208 のときは、ファイルのデフォルト CCSID は、CCSID 1208 ではなく CCSID 65535 (変換なし) になります。これにより、CCSID 1208 をデータベース・ファイルに使用することができます。ファイルの CCSID について詳しくは、114 ページの『fopen() ー ファイルのオープン』を参照してください。

改行文字

UTF サポートを使用しない場合、コンパイラーによって ¥n 文字用に生成され、ランタイムによって使用される 16 進値は、異なる 2 つの値になります。 SYSIFCOPT(*NOIFSIO) をコンパイル・コマンドで指定している場合は、16 進値 0x15 が使用されます。 SYSIFCOPT(*IFSIO) または SYSIFCOPT(*IFS64IO) をコンパイル・コマンドで指定している場合は、16 進値 0x25 が使用されます。 UTF サポートを使用する場合、UTF-8 の改行文字は、使用される SYSIFCOPT 値にかかわらず 16 進値 0x0a になります。

変換エラー

一部のランタイム・ルーチンでは、UTF-8 をサポートしないオペレーティング・システム関数と相互作用する必要がある場合に、UTF-8 から EBCDIC CCSID への CCSID 変換を実行します。このような場合に変換エラーが発生すると、変換情報の付いた C2M1217 メッセージがジョブ・ログに生成されます。

ヒープ・メモリー

ヒープ・メモリーの概要

ヒープ・メモリーは、アプリケーション内で動的メモリー割り振りに使用される、フリー・メモリーの共通プールです。

ヒープ・メモリー・マネージャー

ヒープ・メモリー・マネージャーは、ヒープ・メモリーの管理を行います。ヒープ・メモリー・マネージャーは、以下の基本的なメモリー操作を実行します。

- 割り振り - malloc および calloc によって実行されます。
- 割り振り解除 - free によって実行されます。
- 再割り振り - realloc によって実行されます。

ILE ランタイムは、以下の 3 つの異なるヒープ・メモリー・マネージャーを提供します。

- デフォルト・メモリー・マネージャー - 汎用メモリー・マネージャー
- 高速プール・メモリー・マネージャー - プール・メモリー・マネージャー
- デバッグ・メモリー・マネージャー - アプリケーション・ヒープ問題のデバッグ用メモリー・マネージャー

また、各メモリー・マネージャーには、2 つの異なるバージョン、すなわち、単一レベル・ストア・バージョンとテラスペース・バージョンがあります。ほとんどの場合、2 つのバージョンの振る舞いは類似していますが、単一レベル・ストア・バージョンはポインターを単一レベル・ストア・ストレージに戻し、テラスペース・バージョンはポインターをテラスペース・ストレージに戻すという点が異なります。単一レベル・ストア・バージョンには、単一割り振りについて、16 MB をわずかに下回る制限があります。また、単一レベル・ストア・バージョンには、割り振り済みヒープ・ストレージの最大量について、4 GB をわずかに下回る制限があります。テラスペース・バージョンには、これらの制限はありません。単一レベル・ストア・ストレージおよびテラスペース・ストレージについて詳しくは、「*ILE 概念*」のマニュアルを参照してください。

デフォルト・メモリー・マネージャーは、ほとんどのアプリケーション用に望ましい選択といえます。また、デフォルト・メモリー・マネージャーはデフォルトで使用可能にされているメモリー・マネージャーです。その他のメモリー・マネージャーには、特定の環境では効果的なこともある固有の特徴があります。使用するヒープ・マネージャーを指定するため、およびヒープ・マネージャー・オプションを提供するために、環境変数を使用することができます。場合によっては、使用するヒープ・マネージャーを指定するために、関数を使用することもできます。

注: ヒープ・マネージャー環境変数は、活動化グループ内で呼び出された最初のヒープ関数で、活動化グループごとに 1 回だけチェックされます。環境変数が確実に使用されるようにするには、活動化グループを作成する前に環境変数をセットアップします。

デフォルト・メモリー・マネージャー

デフォルト・メモリー・マネージャーは、パフォーマンスとメモリー所要量のバランスを取ることを試行する汎用のメモリー・マネージャーです。デフォルト・メモリー・マネージャーは、大部分のアプリケーションに対して適切なパフォーマンスを提供しながら必要な追加メモリー量を最小化しようとします。

メモリー・マネージャーは、カルテシアン・バイナリー・サーチ・ツリー内のノードとしてヒープ内のフリー・スペースを維持します。このデータ構造では、ツリーによってサポートされるブロック・サイズ数に制限はなく、考えられる広い範囲のブロック・サイズを使用することができます。

割り振り

各割り振り要求には、少量の追加メモリーが必要です。この追加メモリーは、各割り振り上のヘッダーの必要性、および各メモリー・ブロックの位置合わせの必要性によるものです。各割り振りのヘッダー・サイズは、16 バイトです。各ブロックは、16 バイト境界に位置合わせされている必要があります。したがって、サイズ *n* の割り振りに必要なメモリーの総量は次のようになります。

```
size = ROUND (n+16, 16)
```

例えば、サイズ 37 を割り振ると、ROUND(37+16, 16) のサイズ、すなわち 64 バイトが必要になります。

必要なサイズ以上のツリー・ノードは、ツリーから除去されます。検出されたブロックが必要なサイズを超えている場合、そのブロックは、2 つのブロック、すなわち必要なサイズのブロックと残りのブロックに分割されます。2 番目のブロックは、フリー・ツリーに戻されて将来の割り振りに使用されます。最初のブロックは呼び出し元に戻されます。

フリー・ツリー内で十分なサイズのブロックが検出されない場合、次の処理が発生します。

- ヒープが拡張されます。
- 獲得された拡張のサイズを持つブロックがフリー・ツリーに追加されます。
- 割り振りは上記の説明のように継続します。

割り振り解除

free 操作によって割り振り解除されたメモリー・ブロックは、ツリーのルートに戻されます。新規ノードの挿入ポイントまでのパスに沿った各ノードが検査されて、挿入されるノードに隣接していないかどうかが調べられます。隣接している場合は、2 つのノードはマージされ、新たにマージされたノードはツリー内に再配置されます。隣接ブロックが見付からない場合は、ノードは、ツリー内の適切な場所に挿入されます。隣接ブロックをマージすると、ヒープのフラグメント化を減らすことができます。

再割り振り

再割り振り済みブロックのサイズが元のブロックのサイズより大きく、元のブロックは新規サイズ (例えば、位置合わせ要求のための) に対応するための十分なスペースを既に持っている場合、データ移動を伴わずに、元のブロックが戻されます。再割り振り済みブロックのサイズが元のブロックのサイズより大きい場合は、以下の処理が発生します。

- 要求されたサイズの新規ブロックが割り振られます。
- データが元のブロックから新しいブロックに移動されます。
- 元のブロックが free 操作によってフリー・ツリーに戻されます。
- 新しいブロックが呼び出し元に戻されます。

| 再割り振り済みブロックのサイズが元のブロックのサイズより小さいときに、サイズの差異が少ない場合、
| 元のブロックが戻されます。そうでない場合、再割り振り済みブロックのサイズが元のブロックのサイズよ
| り小さいならば、ブロックは分割されて、残りの部分がフリー・ツリーに戻されます。

| **デフォルト・メモリー・マネージャーの使用可能化**

| デフォルト・メモリー・マネージャーは、デフォルトで使用可能になり、以下の環境変数を設定することに
| よって構成されます。

```
QIBM_MALLOC_TYPE=DEFAULT
QIBM_MALLOC_DEFAULT_OPTIONS=options
```

| デフォルト・メモリー・マネージャーにユーザー指定の構成オプションを指定するには、
| QIBM_MALLOC_DEFAULT_OPTIONS=*options* を設定します。ここで、*options* は、1 つ以上の構成オプシ
| ョンをブランクで区切ったリストです。

| QIBM_MALLOC_TYPE=DEFAULT 環境変数が指定されているときに `_C_Quickpool_Init()` 関数が呼び出
| された場合は、環境変数の設定が `_C_Quickpool_Init()` 関数より優先され、`_C_Quickpool_Init()` 関数
| は、代替ヒープ・マネージャーが既に使用可能にされていることを示す値 -1 を戻します。

| **構成オプション**

| 以下の構成オプションを使用できます。

| MALLOC_INIT:N

| このオプションを使用して、割り振られたメモリーの各バイトを指定値に初期化することを指定できます。
| 値 *N* は、0 から 255 までの範囲の整数を表します。

| このオプションは、デフォルトでは使用可能になりません。

| FREE_INIT:N

| このオプションを使用して、解放されたメモリーの各バイトを指定値に初期化することを指定できます。値
| *N* は、0 から 255 までの範囲の整数を表します。

| このオプションは、デフォルトでは使用可能になりません。

| 任意の数のオプションを任意の順序で指定することができます。ブランクが、構成オプションを区切るため
| に有効な唯一の区切り文字です。各構成オプションは、1 回だけ指定できます。1 つの構成オプションが複
| 数回指定された場合は、最後のインスタンスのみが適用されることになります。構成オプションが無効な値
| で指定された場合、その構成オプションは無視されます。

| **例**

```
ADDENVVAR ENVVAR(QIBM_MALLOC_DEFAULT_OPTIONS) LEVEL(*JOB) REPLACE(*YES) VALUE('')

ADDENVVAR ENVVAR(QIBM_MALLOC_DEFAULT_OPTIONS) LEVEL(*JOB) REPLACE(*YES)
VALUE('MALLOC_INIT:255 FREE_INIT:0')
```

| 1 番目の例は、デフォルトの構成値を示します。2 番目の例は、指定されているすべてのオプションを示し
| ます。

| **関連機能**

| デフォルト・メモリー・マネージャー用の構成オプションを有効にするために、または指定するために使用
| 可能な関数はありません。環境変数サポートを使用する必要があります。

| **関連情報**

- 58 ページの『calloc() ― ストレージの予約と初期化』
- 134 ページの『free() ― ストレージ・ブロックの解放』
- 203 ページの『malloc() ― ストレージ・ブロックの予約』
- 276 ページの『realloc() ― 予約ストレージ・ブロック・サイズの変更』
- 82 ページの『_C_TS_malloc_debug() ― 使用されるテラスペース・メモリー量の判別 (オプションのダンプおよび検査を使用)』
- 84 ページの『_C_TS_malloc_info() ― 使用されるテラスペース・メモリー量の判別』
- 71 ページの『_C_Quickpool_Init() ― 高速プール・メモリー・マネージャーの初期化』

高速プール・メモリー・マネージャー

高速プール・メモリー・マネージャーは、メモリーを一連のプールに分割します。高速プール・メモリー・マネージャーは、小さな割り振り要求を数多く出すアプリケーションのヒープ・パフォーマンスを改善することを目的としています。高速プール・メモリー・マネージャーが使用可能な場合、指定された範囲内のブロック・サイズに収まる割り振り要求には、プール内でセルが割り当てられます。これらの要求は、この範囲から外れた要求よりも迅速に処理できます。この範囲から外れた割り振り要求は、デフォルト・メモリー・マネージャーと同じ方法で処理されます。

プールは、メモリーのブロック (エクステントと呼ばれます) で構成されます。そのブロックは、事前定義された数のさらに小さな同一サイズのブロック (セルと呼ばれます) に細分化されています。各セルは、メモリーのブロックとして割り振ることができます。各プールはプール番号を使用して識別されます。最初のプールがプール 1 で、2 番目のプールがプール 2 で、3 番目のプールがプール 3 です。それ以降も同様です。最初のプールが最も小さく、その後の各プールのサイズは、前のプールと等しいかそれより大きくなります。

プール数および各プールのセル・サイズは、高速プール・メモリー・マネージャーの初期設定時に決定されます。

割り振り

割り振り要求がプールによって定義されているセル・サイズの範囲内に収まっているときに、プールのいずれかから 1 つのセルが割り振られることになります。それぞれの割り振り要求は、スペースを節約して使うために、選択できる最小のプールからサービスを受けます。

プールに対して最初の要求が発生すると、1 つのエクステントがそのプールについて割り振られ、要求はこのエクステントから満たされます。そのエクステントがすべて使用されるまで、このプールに対する後続の要求も、そのエクステントによって満たされます。エクステントがいっぱいになると、プールに新しいエクステントが割り振られます。

割り振り解除

`free` 操作によって割り振り解除されたメモリー・ブロック (セル) は、そのセルを含むプールに関連付けられたフリー・キューに追加されます。各プールには、解放されまだ再割り振りされていないセルを含むフリー・キューがあります。そのプールからの追加割り振り要求は、フリー・キューからのセルを使用します。

再割り振り

再割り振り済みブロックのサイズが元のブロックと同じプール内に収まっている場合、データ移動を伴わずに、元のブロックが戻されます。そうでない場合、要求されたサイズの新規ブロックが割り振られ、データは元のブロックから新規ブロックに移動され、元のブロックは `free` 操作によってフリー・キューに戻され、新規ブロックが呼び出し元に戻されます。

高速プール・メモリー・マネージャーの使用可能化

高速プール・メモリー・マネージャーは、デフォルトでは使用可能になりません。高速プール・メモリー・マネージャーは、`_C_Quickpool_Init()` 関数および `_C_Quickpool_Debug()` 関数を呼び出すか、または以下の環境変数を設定することによって使用可能になり構成されます。

```
QIBM_MALLOC_TYPE=QUICKPOOL
QIBM_MALLOC_QUICKPOOL_OPTIONS=options
```

デフォルトの設定値を使用して高速プール・メモリー・マネージャーを使用可能にする場合、QIBM_MALLOC_QUICKPOOL_OPTIONS 環境変数を指定する必要はありません。この場合は、QIBM_MALLOC_TYPE=QUICKPOOL のみを指定する必要があります。ユーザー指定の構成オプションを使用して高速プール・メモリー・マネージャーを使用可能にするには、QIBM_MALLOC_QUICKPOOL_OPTIONS=*options* を設定します。ここで、*options* は、1 つ以上の構成オプションをブランクで区切ったリストです。

QIBM_MALLOC_TYPE=QUICKPOOL 環境変数が指定されているときに `_C_Quickpool_Init()` 関数が呼び出された場合は、環境変数の設定が `_C_Quickpool_Init()` 関数より優先され、`_C_Quickpool_Init()` 関数は、代替ヒープ・マネージャーが既に使用可能にされていることを示す値 -1 を戻します。

QIBM_MALLOC_TYPE=QUICKPOOL 環境変数が指定されているときに、高速プール・メモリー・マネージャー特性を変更するために `_C_Quickpool_Debug()` 関数が呼び出された場合、`_C_Quickpool_Debug()` 関数のパラメーターで指定された設定は環境変数の設定をオーバーライドします。

構成オプション

以下の構成オプションを使用できます。

POOLS:(C_1 E_1)(C_2 E_2)...(C_n E_n)

このオプションを使用して、使用するプールの数とともに各プールのセル・サイズおよびエクステント・セル数を指定することができます。添え字値 *n* は、プールの数を示します。*n* の最小有効値は 1 です。*n* の最大有効値は 64 です。

値 C_1 はプール 1 のセル・サイズを示し、C_2 はプール 2 のセル・サイズを示し、C_n はプール *n* のセル・サイズを示します。それ以降も同様です。この値は 16 バイトの倍数である必要があります。この値が 16 バイトの倍数でない数に指定された場合、そのセル・サイズは 16 バイトの倍数に一番近い数に切り上げられます。最小有効値は 16 で最大有効値は 4096 です。

値 E_1 はプール 1 のエクステント・セル数を示し、E_2 はプール 2 のエクステント・セル数を示し、E_n はプール *n* のエクステント・セル数を示します。それ以降も同様です。値は、単一エクステント内のセル数を指定します。値には任意の非負数を指定できますが、アーキテクチャーによる制約のためにエクステントの合計サイズは制限される場合があります。値ゼロは、実装で大きな値を選択できることを示します。

このオプションのデフォルト値は、"POOLS:(16 4096) (32 4096) (64 1024) (128 1024) (256 512) (512 512) (1024 256) (2048 256) (4096 256)" です。デフォルトは、サイズが 16、32、64、128、256、512、1024、2048、および 4096 バイトのセルを持つ、9 個のプールを表します。それぞれのエクステント内のセル数は、4096、4096、1024、1024、512、512、256、256、および 256 です。

MALLOC_INIT:N

このオプションを使用して、割り振られたメモリーの各バイトを指定値に初期化することを指定できます。値 *N* は、0 から 255 までの範囲の整数を表します。

このオプションは、デフォルトでは使用可能になりません。

FREE_INIT:N

このオプションを使用して、解放されたメモリーの各バイトを指定値に初期化することを指定できます。値 *N* は、0 から 255 までの範囲の整数を表します。

このオプションは、デフォルトでは使用可能になりません。

COLLECT_STATS

このオプションを使用することにより、高速プール・メモリー・マネージャーが、統計を収集してアプリケーションの終了時にその統計を報告するように指定することができます。高速プール・メモリー・マネージャーは、このオプションが指定されているときに、atexit(_C_Quickpool_Report) を呼び出すことによって統計を収集します。その報告書内に含まれる情報についての詳細は、_C_Quickpool_Report() の説明に記載されています。

このオプションは、デフォルトでは使用可能になりません。

任意の数のオプションを任意の順序で指定することができます。ブランクが、構成オプションを区切るために有効な唯一の区切り文字です。各構成オプションは、1 回だけ指定するようにしてください。1 つの構成オプションが複数回指定された場合は、最後のインスタンスのみが適用されることになります。構成オプションが無効な値で指定された場合、その構成オプションは無視されます。

例

```
ADDENVVAR ENVVAR(QIBM_MALLOC_QUICKPOOL_OPTIONS) LEVEL(*JOB) REPLACE(*YES)
VALUE('POOLS:(16 4096) (32 4096) (64 1024) (128 1024) (256 512) (512 512) (1024 256)
(2048 256) (4096 256)')

ADDENVVAR ENVVAR(QIBM_MALLOC_QUICKPOOL_OPTIONS) LEVEL(*JOB) REPLACE(*YES)
VALUE('POOLS:(16 1000) MALLOC_INIT:255 FREE_INIT:0 COLLECT_STATS')
```

1 番目の例は、デフォルトの構成値を示します。2 番目の例は、指定されているすべてのオプションを示します。

関連機能

_C_Quickpool_Init() 関数は、高速プール・メモリー・マネージャーを使用可能にすることができます。また、_C_Quickpool_Init() 関数は、使用するプールの数、各プールのセル・サイズおよびエクステント・セル数を指定します。

_C_Quickpool_Debug() 関数は、その他の構成オプションを使用可能にします。

_C_Quickpool_Report() 関数が使用されるのは、メモリー統計を報告するためです。

注:

1. 高速プール・メモリー・マネージャーのデフォルト構成は、小さな割り振り要求を数多く出す多くのアプリケーションにおいて、パフォーマンスの改善を実現します。ただし、デフォルト構成を変更することにより、さらに高い効果を得ることができる場合があります。デフォルト構成を変更する前に、アプリケーションのメモリー所要量および使用方法に精通する必要があります。高速プール・メモリー・マネージャー構成の細かな調整を行う COLLECT_STATS オプションを指定して、高速プール・メモリー・マネージャーを使用可能にすることができます。
2. メモリー所要量および使用方法はさまざまであるため、アプリケーションによっては、高速プール・メモリー・マネージャーが使用しているメモリー割り振り方式の利点を得ることができない場合があります。したがって、高速プール・メモリー・マネージャーをシステム全体で使用可能にするのは望ましくありません。最良のパフォーマンスを得るには、高速プール・メモリー・マネージャーをアプリケーションごとに使用可能にして構成します。
3. 同サイズのセルを持つ複数のプールを作成することができます。これは、同じようなサイズの割り振りを数多く実行する、マルチスレッド・アプリケーションの場合に役立つことがあります。競合がまった

く発生していない場合、要求されたサイズの最初のプールが使用されます。最初のプールで競合が発生した場合、高速プール・メモリー・マネージャーは、競合を最小化するために、他の同サイズのプールからセルを割り振ります。

関連情報

- 71 ページの『_C_Quickpool_Init() ― 高速プール・メモリー・マネージャーの初期化』
- 69 ページの『_C_Quickpool_Debug() ― 高速プール・メモリー・マネージャー特性の変更』
- 73 ページの『_C_Quickpool_Report() ― 高速プール・メモリー・マネージャー・レポートの生成』

デバッグ・メモリー・マネージャー

デバッグ・メモリー・マネージャーは、主に、アプリケーションによるヒープの不正使用の検出に使用されます。デバッグ・メモリー・マネージャーは、パフォーマンス用に最適化されておらず、アプリケーションのパフォーマンスに悪影響を及ぼす場合もあります。ただし、ヒープの不正使用の判別には役立ちます。

メモリー管理エラーは、割り振られたバッファーの終端を超えて書き込みを行うことによって起こることがあります。ずっと後に、上書きされた (通常は他に割り振られた) メモリーが参照され、そのメモリーに予想されたデータが含まれていなくなるまで、症状は現れません。

デバッグ・メモリー・マネージャーを使用すると、メモリーの上書き、メモリーのオーバーリード、重複解放、および解放済みメモリーの再利用を検出することができます。デバッグ・メモリー・マネージャーによって検出されるメモリー問題は、次の 2 つのうちいずれかになります。

- 不正使用が発生した時点で問題が検出された場合、MCH 例外メッセージ (通常は、MCH0601、MCH3402、または MCH6801) が生成されます。この場合、通常、エラー・メッセージが表示された後、アプリケーションは停止します。
- 不正使用が既に発生した後になるまで問題が検出されなかった場合、C2M1212 メッセージが生成されます。この場合、通常、メッセージが表示されても、アプリケーションは停止しません。

デバッグ・メモリー・マネージャーは、次の 2 つの方法で、メモリーの上書きおよびメモリーのオーバーリードを検出します。

- 最初の方法として、デバッグ・メモリー・マネージャーは、制限付きアクセス・メモリー・ページを使用します。制限付きアクセス・メモリー・ページは、各割り振りの前後に置かれます。各メモリー・ブロックは、16 バイト境界に位置合わせされ、できるだけページ終端に近い位置に置かれます。メモリー保護が可能なのはページ境界上のみなので、この位置合わせを行うことにより、メモリー上書きおよびメモリー・オーバーリードの最適な検出が行えるようになります。制限付きアクセス・メモリー・ページの 1 つに対して読み取りまたは書き込みが行われると、即時に MCH 例外が出されることになります。
- 2 番目の方法として、デバッグ・メモリー・マネージャーは、各割り振りの前後に埋め込みバイトを使用します。各割り振りの直前のいくつかのバイトは、割り振り時に、事前設定されたバイト・パターンに初期化されます。割り振りサイズを 16 バイトの倍数に丸めるために必要な、割り振りの直後の埋め込みバイトはすべて、割り振り時に、事前設定済みのバイト・パターンに初期化されます。割り振りが解放された場合、すべての埋め込みバイトは検査され、予想された事前設定済みのバイト・パターンを含んだままであることが確認されます。埋め込みバイトのいずれかが変更されている場合、デバッグ・メモリー・マネージャーは C2M1212 メッセージと理由コード X'80000000' を生成して、この事実を示します。

割り振り

各割り振り要求には、大量の追加メモリーが必要です。追加メモリーは、以下のために必要です。

- 割り振りの前のメモリー・ページ (単一レベル・ストア・バージョンのみ)
- 割り振りの後のメモリー・ページ
- 各割り振りのヘッダー
- 各メモリー・ブロックの 16 バイト境界への配置

各割り振りのヘッダー・サイズは、16 バイトです。各ブロックは、16 バイト境界に位置合わせされている必要があります。単一レベル・ストア・バージョンでのサイズ *n* の割り振りに必要なメモリーの総量は、次のようになります。

```
size = ROUND((PAGESIZE * 2) + n + 16, PAGESIZE)
```

例えば、4096 バイトのページ・サイズを使用してサイズ 37 を割り振ると、ROUND(8192 + 37 + 16, 4096) のサイズ、すなわち 12,288 バイトが必要になります。

テラスペース・バージョンでのサイズ *n* の割り振りに必要なメモリーの総量は、次のようになります。

```
size = ROUND(PAGESIZE + n + 16, PAGESIZE)
```

例えば、4096 バイトのページ・サイズを使用してサイズ 37 を割り振ると、ROUND(4096 + 37 + 16, 4096) のサイズ、すなわち 8,192 バイトが必要になります。

割り振り解除

free 操作によって割り振り解除されたメモリー・ブロックは、システムに戻されます。ページ保護属性は、そのメモリー・ブロックに対する後続の読み取りまたは書き込みアクセスはすべて MCH 例外を生成するように設定されています。

再割り振り

いずれの場合も、次の処理が発生します。

- 要求されたサイズの新規ブロックが割り振られます。
- データが元のブロックから新しいブロックに移動されます。
- 元のブロックが free 操作によって戻されます。
- 新しいブロックが呼び出し元に戻されます。

デバッグ・メモリー・マネージャーの使用可能化

デバッグ・メモリー・マネージャーは、デフォルトでは使用可能になりませんが、以下の環境変数を設定することによって使用可能になり構成されます。

```
QIBM_MALLOC_TYPE=DEBUG
QIBM_MALLOC_DEBUG_OPTIONS=options
```

デフォルトの設定値を使用してデバッグ・メモリー・マネージャーを使用可能にするには、QIBM_MALLOC_TYPE=DEBUG を指定する必要があります。ユーザー指定の構成オプションを使用してデバッグ・メモリー・マネージャーを使用可能にするには、QIBM_MALLOC_DEBUG_OPTIONS=*options* を設定します。ここで、*options* は、1 つ以上の構成オプションをブランクで区切ったリストです。

QIBM_MALLOC_TYPE=DEBUG 環境変数が指定されているときに _C_Quickpool_Init() 関数が呼び出された場合は、環境変数の設定が _C_Quickpool_Init() 関数より優先され、_C_Quickpool_Init() 関数は、代替ヒープ・マネージャーが使用可能にされていることを示す値 -1 を戻します。

構成オプション

以下の構成オプションを使用できます。

MALLOC_INIT:N

このオプションを使用して、割り振られたメモリーの各バイトを指定値に初期化することを指定できます。値 *N* は、0 から 255 までの範囲の整数を表します。

このオプションは、デフォルトでは使用可能になりません。

FREE_INIT:N

このオプションを使用して、解放されたメモリーの各バイトを指定値に初期化することを指定できます。値 *N* は、0 から 255 までの範囲の整数を表します。

このオプションは、デフォルトでは使用可能になりません。

任意の数のオプションを任意の順序で指定することができます。ブランクが、構成オプションを区切るために有効な唯一の区切り文字です。各構成オプションは、1 回だけ指定するようにしてください。1 つの構成オプションが複数回指定された場合は、最後のインスタンスのみが適用されることになります。構成オプションが無効な値で指定された場合、その構成オプションは無視されます。

例

```
ADDENVVAR ENVVAR(QIBM_MALLOC_DEBUG_OPTIONS) LEVEL(*JOB) REPLACE(*YES) VALUE('')

ADDENVVAR ENVVAR(QIBM_MALLOC_DEBUG_OPTIONS) LEVEL(*JOB) REPLACE(*YES)
VALUE('MALLOC_INIT:255 FREE_INIT:0')
```

1 番目の例は、デフォルトの構成値を示します。2 番目の例は、指定されているすべてのオプションを示します。

関連機能

デバッグ・メモリー・マネージャー用の構成オプションを有効にするために、または指定するために使用可能な関数はありません。構成オプションを有効にする、または指定するには、環境変数サポートを使用します。

注:

1. デバッグ・メモリー・マネージャーは、一度に 1 つのアプリケーション、あるいは小さなアプリケーション・グループをデバッグする場合に使用します。

   デバッグ・メモリー・マネージャーは、常時、持続的に使用したり、システム全体で使用したりするのには適していません。デバッグ・メモリー・マネージャーは、デバッグするアプリケーションへのパフォーマンスの影響を最小限に押さえるように設計されていますが、システム全体で使用すると、システム全体のスループットに重大な悪影響を及ぼす場合があります。システム補助記憶域プール (ASP) の使用超過などの重大なシステムの問題を引き起こすことがあります。
2. デバッグ・メモリー・マネージャーは、デフォルト・メモリー・マネージャーに比べてはるかに多くのメモリーを消費します。その結果、デバッグ・メモリー・マネージャーは、デバッグする状態によっては使用するのが適切ではない場合があります。

   割り振りごとに 2 メモリー・ページ以上の追加メモリーを必要とするので、小さな割り振り要求を数多く出すアプリケーションでは、メモリー使用量が急激に増加します。これらのプログラムでは、メモリー不足によってメモリー割り振り要求が拒否され、別の障害が発生する場合があります。この障害は、必ずしもデバッグ中のアプリケーションで起こるエラーとは限らず、またデバッグ・メモリー・マネージャーで発生するエラーでもありません。

   単一レベル・ストア・バージョンには、割り振り済みヒープ・ストレージの最大量について、4 GB をわずかに下回る制限があります。デバッグ・メモリー・マネージャーは、割り振りごとに少なくとも 3 ページを割り振ります。これにより、350,000 より少ない未解決のヒープ割り振りが可能になります (4096 バイトのページ・サイズを使用した場合)。

3. デバッグ・メモリー・マネージャーの単一レベル・ストア・バージョンは、各割り振りを別個の 16 MB セグメント内で行います。このため、システムが一時アドレスをより頻繁に使用することになる場合があります。

関連情報

- 58 ページの『calloc() — ストレージの予約と初期化』
- 134 ページの『free() — ストレージ・ブロックの解放』
- 203 ページの『malloc() — ストレージ・ブロックの予約』
- 276 ページの『realloc() — 予約ストレージ・ブロック・サイズの変更』
- 71 ページの『_C_Quickpool_Init() — 高速プール・メモリー・マネージャーの初期化』

環境変数

以下の表では、ヒープ・メモリー・マネージャーを使用可能にして構成するために使用できる環境変数を説明します。

以下の環境変数を使用して、使用すべきメモリー・マネージャーを示すことができます。

表 32. 使用するメモリー・マネージャーを示す環境変数

環境変数	値	説明
QIBM_MALLOC_TYPE	DEFAULT	デフォルト・メモリー・マネージャーを使用することを示します。
	QUICKPOOL	高速プール・メモリー・マネージャーを使用することを示します。
	DEBUG	デバッグ・メモリー・マネージャーを使用することを示します。

QIBM_MALLOC_TYPE 環境変数が設定されていない場合、または QIBM_MALLOC_TYPE 環境変数の値が上記の値の 1 つではない場合、デフォルト・メモリー・マネージャーが使用され、以下の環境変数はすべて無視されます。

QIBM_MALLOC_TYPE が DEFAULT に設定されている場合、以下の環境変数を使用して、デフォルト・メモリー・マネージャー・オプションを示すことができます。そうでない場合、この環境変数は無視されます。

表 33. デフォルト・メモリー・マネージャー・オプション

環境変数	値	説明
QIBM_MALLOC_DEFAULT_OPTIONS	MALLOC_INIT:N	割り振られたメモリーの各バイトは、この値に初期化されます。
	FREE_INIT:N	解放されたメモリーの各バイトは、この値に初期化されます。

デフォルトでは、割り振られたメモリーと解放されたメモリーのどちらも初期化されません。

QIBM_MALLOC_TYPE が QUICKPOOL に設定されている場合、以下の環境変数を使用して、高速プール・メモリー・マネージャー・オプションを示すことができます。そうでない場合、この環境変数は無視されます。

表 34. 高速プール・メモリー・マネージャー・オプション

環境変数	値	説明
QIBM_MALLOC_QUICKPOOL_OPTIONS	POOLS:(C_1 E_1) (C_2 E_2) ... (C_n E_n)	各プールのセル・サイズおよびエクステント・セル数を定義します。(C_n E_n) ペアの数は、プールの数を示します。
	MALLOC_INIT:N	割り振られたメモリーの各バイトは、この値に初期化されます。
	FREE_INIT:N	解放されたメモリーの各バイトは、この値に初期化されます。
	COLLECT_STATS	統計を収集してアプリケーション終了時にレポートを生成することを示します。

デフォルトでは、割り振られたメモリーと解放されたメモリーのどちらも初期化されません。デフォルトの振る舞いは、統計収集を行わないことです。セル・サイズおよびエクステント・セル数が、指定されていないか、あるいは正しく指定されていない場合、本セクションで前述したように、デフォルトの構成値が使用されます。

QIBM_MALLOC_TYPE が DEBUG に設定されている場合、以下の環境変数を使用して、デバッグ・メモリー・マネージャー・オプションを示すことができます。そうでない場合、この環境変数は無視されます。

表 35. デバッグ・メモリー・マネージャー・オプション

環境変数	値	説明
QIBM_MALLOC_DEBUG_OPTIONS	MALLOC_INIT: N	割り振られたメモリーの各バイトは、この値に初期化されます。
	FREE_INIT:N	解放されたメモリーの各バイトは、この値に初期化されます。

デフォルトでは、割り振られたメモリーと解放されたメモリーのどちらも初期化されません。

C2M1211/C2M1212 メッセージの問題の診断

このセクションでは、ジョブ・ログ内の C2M1211 メッセージまたは C2M1212 メッセージで示される問題の診断に役立つ可能性がある情報を提供します。

C2M1211 メッセージ

C2M1211 メッセージは、テラスペース・バージョンのヒープ・メモリー・マネージャーが、ヒープ制御構造の破損を検出したことを示します。

C2M1211 メッセージは、さまざまなことが原因で表示されます。最も一般的な原因としては、次のようなものがあります。

- スペースを 2 回解放する。
- 割り振り済みストレージの境界外に書き込みする。
- 解放されているストレージへ書き込みする。

CM1211 メッセージは、多くの場合、アプリケーション・ヒープ問題を示します。残念ながら、多くの場合、この問題の追跡は困難です。このタイプの問題をデバッグするための最善のアプローチは、デバッグ・メモリー・マネージャーを使用可能に設定することです。

C2M1212 メッセージ

C2M1212 メッセージは、メモリーの破損およびその他の問題の原因となる可能性がある、あるタイプのメモリー問題を示します。メモリーの破損は、アプリケーション・コード内またはオペレーティング・システム・コード内で発生する可能性があります。このメッセージは、診断メッセージにすぎませんが、実際の問題の標識の場合があります。C2M1212 メッセージが、他の問題のソースである場合と、そうでない場合があります。可能な場合は、メモリー問題をクリーンアップしてください。

C2M1212 メッセージが生成される場合、free() 関数に渡されたポインターの 16 進値がメッセージ記述の一部として組み込まれています。この 16 進値は、問題の発生元への手掛かりを与えてくれます。malloc() 関数は、16 進値 0 で終了するポインターのみを戻します。16 進値 0 で終了しないすべてのポインターは、malloc() 関数で予約されないストレージを指すように設定されなかったか、または malloc() 関数によって予約されたストレージを指すように設定されたために変更されました。ポインターが 16 進数 0 で終了する場合、C2M1212 メッセージの原因が不確定であり、free() を呼び出すプログラム・コードを検査する必要があります。

ほとんどの場合、単一レベル・ストア・ヒープ・メモリー・マネージャーからの C2M1212 メッセージの前に、MCH6902 メッセージが出されます。MCH6902 メッセージは、問題が何であるかを示すエラー・コードを持っています。最も一般的なエラー・コードは 2 です。これは、現在割り振りされていないメモリーが解放されようとしていることを示します。このエラー・コードは、次のいずれかを意味しています。

- 割り振りされていないメモリーが解放されようとしている。
- メモリーが、再度、解放されようとしている。

場合によっては、メモリー・リークが原因で単一レベル・ストア・ヒープがフラグメント化され、その結果ヒープ制御セグメントがいっぱいになり、割り振り解除ができなくなってしまっている場合があります。この問題は、MCH6906 メッセージによって示されます。この場合、唯一の解決策は、アプリケーションをデバッグしてメモリー・リークを修正することです。

スタックのトレースバック (『スタックのトレースバック』を参照) を使用して、問題を引き起こしている原因となるコードを検出することができます。コードが検出された後、難しい部分は、ヒープ・ストレージへのポインターでの問題は何かを判別することです。考えられるいくつかの原因は、次のとおりです。

1. ポインターが初期化されておらず、予期しない値を含んでいる。C2M1212 メッセージは、ポインターの 16 進値をダンプします。
2. ポインターは、malloc() から取得されたものではない。このポインターは、自動 (ローカル) 変数または静的 (グローバル) 変数へのポインターであり、malloc() からのヒープ・ストレージへのポインターではないことが考えられます。
3. ポインターは、malloc() から戻された後に変更された。例えば、malloc() から戻されたポインターが、若干増分された後に free() に渡された場合、このポインターは無効となり、C2M1212 メッセージが出されます。
4. ポインターは、再度、free() に渡されようとしている。free() がポインターで呼び出されると、そのポインターによって指されたスペースは割り振り解除され、free() が再び呼び出された場合は、C2M1212 メッセージが出されます。
5. ヒープ割り振りを追跡するためにヒープ・マネージャーが保守するヒープ構造が破壊されている。この場合、ポインターが有効であっても、ヒープ・マネージャーはそれを判別できず、その結果として C2M1212 メッセージが出されます。ヒープ構造が破壊された場合、通常、ヒープ破壊が発生したことを示す C2M1211 メッセージが、ジョブ・ログに少なくとも 1 つ表示されます。
6. デバッグ・メモリー・マネージャーが使用中であり、C2M1212 メッセージ上の理由コードが X'8000000000' である場合は、指定された割り振りに関して埋め込みバイトが上書きされています。詳しくは、 573 ページの『デバッグ・メモリー・マネージャー』を参照してください。

スタックのトレースバック

単一レベル・ストア・ヒープ・メモリー・マネージャーの使用可能化

C2M1211 メッセージまたは C2M1212 メッセージが単一レベル・ストア・ヒープ・ルーチンから生成された場合、コードが QGPL/QC2M1211 または QGPL/QC2M1212 という名前の付いた *DTAARA を検査します。データ域が存在する場合、プログラム・スタックがダンプされます。データ域が存在しない場合、ダンプは実行されません。

テラスペース・ヒープ・メモリー・マネージャーの使用可能化

C2M1211 メッセージまたは C2M1212 メッセージがテラスペース・ヒープ・ルーチンから生成された場合、コードが QGPL/QC2M1211 または QGPL/QC2M1212 という名前の付いた *DTAARA を検査します。データ域が存在し、そこに少なくとも 50 文字のデータが含まれている場合、データ域から 50 文字ストリングがリトリーブされます。データ域内のストリングが以下のストリングのいずれかと一致する場合、特殊な振る舞いがトリガーされます。

```
_C_TS_dump_stack
_C_TS_dump_stack_vfy_heap
_C_TS_dump_stack_vfy_heap_wabort
_C_TS_dump_stack_vry_heap_wsleep
```

データ域が存在しない場合、ダンプまたはヒープ検査は実行されません。

次の場合、振る舞いは、デフォルトによって _C_TS_dump_stack の振る舞いになります。

- データ域は存在するが、そこに文字データが含まれていない。
- データ域の長さが、50 文字より短い。

| • データ域には、リストされているストリングが 1 つも含まれていない。

| データ域のストリングには、以下の意味があります。

| **_C_TS_dump_stack**

| スタックをダンプするデフォルトの振る舞いが実行されます。ヒープ検査は実行されません。

| **_C_TS_dump_stack_vfy_heap**

| スタックがダンプされた後、ヒープ制御構造を検査するために `_C_TS_malloc_debug()` 関数が呼び出されます。ヒープ制御構造内で破壊が検出された場合、ヒープ・エラーおよびすべてのヒープ制御情報がダンプされます。ダンプされたヒープ情報はすべて、スタック・ダンプと同じファイル内に収容されます。ヒープ破壊が検出されない場合、ヒープ情報はダンプされません。

| 検査が実行された後、C2M1211 メッセージまたは C2M1212 メッセージを生成する元のプログラムに制御が戻り、実行が継続されます。

| **_C_TS_dump_stack_vfy_heap_wabort**

| `_C_TS_dump_stack_vfy_heap_wabort` は、`_C_TS_dump_stack_vfy_heap` と同様の検査の振る舞いをします。

| ヒープ破壊が検出された場合、元のプログラムに制御を戻す代わりに、`abort()` 関数を呼び出して実行を停止します。

| **_C_TS_dump_stack_vfy_heap_wsleep**

| `_C_TS_dump_stack_vfy_heap_wsleep` は、`_C_TS_dump_stack_vfy_heap` と同様の検査の振る舞いをします。

| ヒープ破壊が検出された場合、元のプログラムに制御を戻す代わりに、`sleep()` 関数を呼び出して無期限にスリープさせ、実行を休止してアプリケーションのデバッグを可能にします。アプリケーションは手動で終了する必要があります。

| 次の例は、C2M1212 メッセージが生成されるたびに、`_C_TS_malloc_debug` を呼び出してヒープを検査することを指示するデータ域の作成方法を示します。

```
CRTDTAARA DTAARA(QGPL/QC2M1212) TYPE(*CHAR) LEN(50)
     VALUE('_C_TS_dump_stack_vfy_heap')
```

| 分析

| データ域が存在する場合、各 C2M1211 メッセージまたは C2M1212 メッセージに関連したダンプ情報を使用して QPRINT という名前のスプール・ファイルが作成されます。スプール・ファイルは、メッセージを受け取るジョブを実行しているユーザー用に作成されます。例えば、C2M1211 メッセージまたは C2M1212 メッセージを受け取るジョブが、ユーザー ID ABC123 の下で実行されているサーバー・ジョブまたはバッチ・ジョブである場合、スプール・ファイルは、ユーザー ID ABC123 用の出力キューに作成されます。スタックのトレースバックを含むスプール・ファイルが取得された場合、データ域は除去することができます。そして、トレースバックを分析することができます。

| スタックのトレースバックを使用して、問題を引き起こしている原因となるコードを検出することができます。以下は、スタックのトレースバックのサンプルです。

```
PROGRAM NAME  PROGRAM LIB  MODULE NAME  MODULE LIB  INST#   PROCEDURE        STATEMENT#
QC2UTIL1      QSYS         QC2ALLOC     QBUILDSS1   000000  dump_stack__Fv   0000001019
QC2UTIL1      QSYS         QC2ALLOC     QBUILDSS1   000000  free             0000001128
QYPPRT370     QSYS         DLSCTODF37   QBUILDSS1   000000  __dl__FPv        0000000007
FSOSA         ABCSYS       OSAACTS      FSTESTOSA   000000  FS_FinalizeDoc   0000000110
ABCKRNL       ABCSYS       A2PDFUTILS   ABMOD_8     000000  PRT_EndDoc_Adb   0000000625
```

```
| ABCKRNL      ABCSYS      A2PDFUTILS  ABMOD_8    000000 PRT_EndDoc       0000000003
| ABCKRNL      ABCSYS      A2ENGINE    ABMOD_8    000000 ABCReport_Start 0000000087
| ABCKRNL      ABCSYS      A2ENTRYPNT  ABMOD_8    000000 ABCReport_Run    0000000056
| ABCKRNL      ABCSYS      A2ENTRYPNT  ABMOD_8    000000 ABCReport_Entry 0000000155
| PRINTABC     ABCSYS      RUNBATCH    ABMOD_6    000000 main             0000000040
| PRINTABC     ABCSYS      RUNBATCH    ABMOD_6    000000 _C_pep
| QCMD         QSYS                               000422
```

| 先頭行は、プログラム名、プログラム・ライブラリー、モジュール名、モジュール・ライブラリー、命令番
| 号、プロシージャー名、およびステートメント番号を示すヘッダー行です。

| ヘッダーの下の最初の行は、常に、dump_stack プロシージャーです。このプロシージャーが、C2M1211
| メッセージまたは C2M1212 メッセージを生成しています。次の行は、dump_stack プロシージャーを呼び
| 出しているプロシージャーです。このプロシージャーは、ほとんどの場合、free プロシージャーですが、
| realloc またはそれ例外のプロシージャーの場合もあります。次の行は、__dl__FPv プロシージャーです。
| これは、C++ 削除演算子を処理するプロシージャーです。C++ コードの場合、このプロシージャーがスタ
| ックの中にあることがよくありますが、C コードの場合は、そうではありません。

| free 関数および delete 関数は、呼び出し元の代わりにメモリーを解放しているライブラリー・ルーチン
| です。これらの関数は、メモリー問題の原因の判別においては重要ではありません。

| __dl__FPv プロシージャーの次の行は、興味をひきつける行です。この例では、プロシージャーは
| FS_FinalizeDoc と呼ばれます。このコードは、不正な削除呼び出しを含んでいます (既に削除済み/解放済
| みのオブジェクトを削除しようとしています)。アプリケーションの所有者は、このプロシージャーのソー
| ス・コードの指定されたステートメント番号を調べて、何が削除または解放されようとしているかを判別す
| る必要があります。場合によっては、このオブジェクトはなんらかのタイプのローカル・オブジェクトであ
| り、問題の判別が容易であることもあります。別の場合では、オブジェクトはパラメーターとしてプロシー
| ジャーに渡されることがあり、そのプロシージャーの呼び出し元を検査することが必要です。この例では、
| PRT_EndDoc_Adb プロシージャーが、FS_FinalizeDoc の呼び出し元です。

| この例の場合、問題は、ABCSYS ライブラリー内のコードにあります。

付録 A. ライブラリー関数および拡張機能

この章では、すべての標準 C ライブラリー関数および ILE C ライブラリー拡張機能についてまとめています。

標準 C ライブラリー関数表 (名前順)

この表では、C ライブラリーをアルファベット順にリストして、簡単な説明を行っています。この表には、各関数のインクルード・ファイル名および関数プロトタイプが記載されています。

表 36. 標準 C ライブラリー関数

関数	システム・インクルード・ファイル	関数プロトタイプ	説明
abort	stdlib.h	void abort(void);	プログラムを異常終了します。
abs	stdlib.h	int abs(int *n*);	整数の引数 *n* の絶対値を計算します。
acos	math.h	double acos(double *x*);	*x* のアークコサインを計算します。
asctime	time.h	char *asctime(const struct tm **time*);	構造体として保管されている *time* を、文字ストリングに変換します。
asctime_r	time.h	char *asctime_r(const struct tm *tm, char *buf);	構造体として保管されている *tm* を、文字ストリングに変換します。(asctime の再始動可能バージョン。)
asin	math.h	double asin(double *x*);	*x* のアークサインを計算します。
assert	assert.h	void assert(int *expression*);	式が false の場合に、診断メッセージを出力してプログラムを終了します。
atan	math.h	double atan(double *x*);	*x* のアークタンジェントを計算します。
atan2	math.h	double atan2(double *y*, double *x*);	*y/x* のアークタンジェントを計算します。
atexit	stdlib.h	int atexit(void (*func*)(void));	正常終了時に呼び出される関数を登録します。
atof	stdlib.h	double atof(const char *string*);	*string* を、倍精度の浮動小数点値に変換します。
atoi	stdlib.h	int atoi(const char *string*);	*string* を整数に変換します。
atol	stdlib.h	long int atol(const char *string*);	*string* を long 型整数に変換します。
bsearch	stdlib.h	void *bsearch(const void *key*, const void *base*, size_t *num*, size_t *size*, int (*compare*) (const void *element1*, const void *element2*));	*num* エレメント (それぞれ *size* バイト) の配列のバイナリー・サーチを行います。この配列は、*compare* で示される関数によって、昇順でソートする必要があります。
btowc	stdio.h wchar.h	wint_t btowc(int *c*);	初期シフト状態時に、*c* が有効なマルチバイト文字で構成されているかどうかを判別します。

© Copyright IBM Corp. 1999, 2010

表 36. 標準 C ライブラリー関数 (続き)

関数	システム・インクルード・ファイル	関数プロトタイプ	説明
calloc	stdlib.h	void *calloc(size_t *num*, size_t *size*);	*num* エレメント (それぞれのサイズが *size* である) 用のストレージ・スペースを予約して、すべてのエレメントの値を 0 に初期化します。
catclose[6]	nl_types.h	int catclose (nl_catd catd);	すでにオープンされているメッセージ・カタログをクローズします。
catgets[6]	nl_types.h	char *catgets(nl_catd catd, int set_id, int msg_id, const char *s);	オープンされているメッセージ・カタログから、メッセージを取得します。
catopen[6]	nl_types.h	nl_catd catopen(const char *name, int oflag);	メッセージ・カタログをオープンします (メッセージが取得される前に行う必要があります)。
ceil	math.h	double ceil(double *x*);	*x* 以上の最小整数を表す double 値を計算します。
clearerr	stdio.h	void clearerr(FILE **stream*);	*stream* のエラー標識およびファイル終了標識をリセットします。
clock	time.h	clock_t clock(void);	ジョブ開始後に経過したプロセッサー時間を戻します。
cos	math.h	double cos(double *x*);	*x* のコサインを計算します。
cosh	math.h	double cosh(double *x*);	*x* の双曲線コサインを計算します。
ctime	time.h	char *ctime(const time_t **time*);	*time* を文字ストリングに変換します。
ctime64	time.h	char *ctime64(const time64_t **time*);	*time* を文字ストリングに変換します。
ctime_r	time.h	char *ctime_r(const time_t *time, char *buf);	*time* を文字ストリングに変換します。(ctime の再始動可能バージョン。)
ctime64_r	time.h	char *ctime64_r(const time64_t *time, char *buf);	*time* を文字ストリングに変換します。(ctime64 の再始動可能バージョン。)
difftime	time.h	double difftime(time_t *time2*, time_t *time1*);	*time2* と *time1* との差を計算します。
difftime64	time.h	double difftime64(time64_t *time2*, time64_t *time1*);	*time2* と *time1* との差を計算します。
div	stdlib.h	div_t div(int *numerator*, int *denominator*);	*numerator* を *denominator* で割った商および剰余を計算します。
erf	math.h	double erf(double *x*);	*x* の誤差関数を計算します。
erfc	math.h	double erfc(double *x*);	*x* のラージ値の誤差関数を計算します。
exit	stdlib.h	void exit(int *status*);	プログラムを正常に終了します。
exp	math.h	double exp(double *x*);	浮動小数点引数 *x* の指数関数を計算します。
fabs	math.h	double fabs(double *x*);	浮動小数点引数 *x* の絶対値を計算します。
fclose	stdio.h	int fclose(FILE **stream*);	指定された *stream* をクローズします。

表 36. 標準 C ライブラリー関数 (続き)

関数	システム・インクルード・ファイル	関数プロトタイプ	説明
fdopen[5]	stdio.h	FILE *fdopen(int *handle*, const char **type*);	入力ストリームまたは出力ストリームを、ハンドルで判別されたファイルに関連付けます。
feof	stdio.h	int feof(FILE **stream*);	指定された *stream* にファイル終了フラグが設定されているかどうか、テストします。
ferror	stdio.h	int ferror(FILE **stream*);	*stream* からの読み取り中または書き込み中でのエラー標識をテストします。
fflush[1]	stdio.h	int fflush(FILE **stream*);	出力 *stream* に関連するバッファーの内容を書き込みます。
fgetc[1]	stdio.h	int fgetc(FILE **stream*);	入力 *stream* から単一の符号なし文字を読み取ります。
fgetpos[1]	stdio.h	int fgetpos(FILE **stream*, fpos_t **pos*);	*stream* に関連するファイル・ポインターの現在位置を、*pos* が示すオブジェクトに保管します。
fgets[1]	stdio.h	char *fgets(char **string*, int *n*, FILE **stream*);	入力 *stream* からストリングを読み取ります。
fgetwc[6]	stdio.h wchar.h	wint_t fgetwc(FILE **stream*);	*stream* が指す入力ストリームから、次に来るマルチバイト文字を読み取ります。
fgetws[6]	stdio.h wchar.h	wchar_t *fgetws(wchar_t **wcs*, int *n*, FILE **stream*);	ストリームからのワイド文字を読み取って、*wcs* が示す配列にします。
fileno[5]	stdio.h	int fileno(FILE **stream*);	*stream* に現在関連付けられているファイル・ハンドルを判別します。
floor	math.h	double floor(double *x*);	*x* 以下の最大整数を表す浮動小数点値を計算します。
fmod	math.h	double fmod(double *x*, double *y*);	*x/y* の剰余の浮動小数点を計算します。
fopen	stdio.h	FILE *fopen(const char **filename*, const char **mode*);	指定されたファイルをオープンします。
fprintf	stdio.h	int fprintf(FILE **stream*, const char **format-string*, *arg-list*);	文字および値をフォーマット設定して、出力 *stream* にプリントします。
fputc[1]	stdio.h	int fputc(int *c*, FILE **stream*);	出力 *stream* に文字をプリントします。
fputs[1]	stdio.h	int fputs(const char **string*, FILE **stream*);	出力 *stream* にストリングをコピーします。
fputwc[6]	stdio.h wchar.h	wint_t fputwc(wchar_t *wc*, FILE **stream*);	ワイド文字 *wc* をマルチバイト文字に変換して、それを現在位置の *stream* が指した出力ストリームに書き込みます。
fputws[6]	stdio.h wchar.h	int fputws(const wchar_t **wcs*, FILE **stream*);	ワイド文字ストリング *wcs* をマルチバイト文字ストリングに変換し、マルチバイト文字ストリングとして *stream* に書き込みます。

表 36. 標準 C ライブラリー関数 (続き)

関数	システム・インクルード・ファイル	関数プロトタイプ	説明
fread	stdio.h	size_t fread(void **buffer*, size_t *size*, size_t *count*, FILE **stream*);	入力 *stream* から、最大 *count* 項目 (長さは *size*) までを読み取って、*buffer* に保管します。
free	stdlib.h	void free(void **ptr*);	ストレージのブロックを解放します。
freopen	stdio.h	FILE *freopen(const char **filename*, const char **mode*, FILE **stream*);	*stream* をクローズし、指定されたファイルに再割り当てします。
frexp	math.h	double frexp(double *x*, int **expptr*);	浮動小数点数を小数部と指数とに分離します。
fscanf	stdio.h	int fscanf(FILE **stream*, const char **format-string*, *arg-list*);	*stream* からデータを読み取って、*arg-list* が指定した位置に置きます。
fseek[1]	stdio.h	int fseek(FILE **stream*, long int *offset*, int *origin*);	*stream* に関連付けられた現行のファイル位置を、新しい位置に変更します。
fsetpos[1]	stdio.h	int fsetpos(FILE **stream*, const fpos_t **pos*);	現行のファイル位置を、*pos* により判別された新しい位置に移動します。
ftell[1]	stdio.h	long int ftell(FILE **stream*);	ファイル・ポインターの現在位置を取得します。
fwide[6]	stdio.h wchar.h	int fwide(FILE **stream*, int *mode*);	*stream* が示すストリームの方向を判別します。
fwprintf[6]	stdio.h wchar.h	int fwprintf(FILE **stream*, const wchar_t **format*, *arg-list*);	*stream* が示すストリームへの出力を書き込みます。
fwrite	stdio.h	size_t fwrite(const void **buffer*, size_t *size*,size_t *count*, FILE **stream*);	*buffer* から *stream* まで、最大 *count* 項目 (長さは *size*) までを書き込みます。
fwscanf[6]	stdio.h wchar.h	int fwscanf(FILE **stream*, const wchar_t **format*, *arg-list*)	*stream* が示すストリームから、入力を読み取ります。
gamma	math.h	double gamma(double *x*);	ガンマ関数を計算します
getc[1]	stdio.h	int getc(FILE **stream*);	入力 *stream* から、単一の文字を読み取ります。
getchar[1]	stdio.h	int getchar(void);	*stdin.* から、単一の文字を読み取ります。
getenv	stdlib.h	char *getenv(const char **varname*);	*varname* の環境変数を検索します。
gets	stdio.h	char *gets(char **buffer*);	*stdin* からストリングを読み取り、*buffer* に保管します。
getwc[6]	stdio.h wchar.h	wint_t getwc(FILE **stream*);	次に来るマルチバイト文字を *stream* から読み取り、ワイド文字に変換して、*stream* の関連するファイル位置標識を前に進めます。
getwchar[6]	wchar.h	wint_t getwchar(void);	次のマルチバイト文字を stdin から読み取り、それをワイド文字に変換し、stdin の関連するファイル位置標識を前に進めます。

表 36. 標準 C ライブラリー関数 (続き)

関数	システム・インクルード・ファイル	関数プロトタイプ	説明
gmtime	time.h	struct tm *gmtime(const time_t **time*);	*time* 値を tm 型の構造体に変換します。
gmtime64	time.h	struct tm *gmtime64(const time64_t **time*);	*time* 値を tm 型の構造体に変換します。
gmtime_r	time.h	struct tm *gmtime_r(const time_t *time, struct tm *result);	*time* 値を tm 型の構造体に変換します。 (gmtime の再始動可能バージョン。)
gmtime64_r	time.h	struct tm *gmtime64_r(const time64_t *time, struct tm *result);	*time* 値を tm 型の構造体に変換します。 (gmtime64 の再始動可能バージョン。)
hypot	math.h	double hypot(double *side1*, double *side2*);	長さ *side1* および *side2* の二辺を持つ直角三角形の、斜辺の長さを計算します。
isalnum	ctype.h	int isalnum(int *c*);	*c* が英数字かどうかをテストします。
isalpha	ctype.h	int isalpha(int *c*);	*c* が英字かどうかをテストします。
isascii	ctype.h	int isascii(int *c*);	7 ビットの US-ASCII の範囲内に *c* があるかどうか、テストします。
iscntrl	ctype.h	int iscntrl(int *c*);	*c* が制御文字であるかどうかをテストします。
isdigit	ctype.h	int isdigit(int *c*);	*c* が 10 進数であるかどうかをテストします。
isgraph	ctype.h	int isgraph(int *c*);	*c* が印刷可能文字 (スペース以外) であるかどうかをテストします。
islower	ctype.h	int islower(int *c*);	*c* が小文字であるかどうかをテストします。
isprint	ctype.h	int isprint(int *c*);	*c* が印刷可能文字 (スペースも含む) であるかどうかをテストします。
ispunct	ctype.h	int ispunct(int *c*);	*c* が句読文字であるかどうかをテストします。
isspace	ctype.h	int isspace(int *c*);	*c* が空白文字であるかどうかをテストします。
isupper	ctype.h	int isupper(int *c*);	*c* が大文字であるかどうかをテストします。
iswalnum[4]	wctype.h	int iswalnum (wint_t wc);	英数字のワイド文字の有無をチェックします。
iswalpha[4]	wctype.h	int iswalpha(wint_t wc);	英字のワイド文字の有無をチェックします。
iswcntrl[4]	wctype.h	int iswcntrl(wint_t wc);	制御ワイド文字の有無をテストします。
iswctype[4]	wctype.h	int iswctype(wint_t wc, wctype_t wc_prop);	ワイド文字 wc にプロパティー wc_prop があるかどうかを判別します。

表 36. 標準 C ライブラリー関数 (続き)

関数	システム・インクルード・ファイル	関数プロトタイプ	説明
iswdigit[4]	wctype.h	int iswdigit(wint_t wc);	10 進数のワイド文字の有無をチェックします。
iswgraph[4]	wctype.h	int iswgraph(wint_t wc);	印刷ワイド文字 (ワイド文字スペースを除く) の有無をチェックします。
iswlower[4]	wctype.h	int iswlower (wint_t wc);	小文字のワイド文字の有無をチェックします。
iswprint[4]	wctype.h	int iswprint(wint_t wc);	印刷ワイド文字の有無をチェックします。
iswpunct[4]	wctype.h	int iswpunct(wint_t wc);	非英数字で非スペースのワイド文字の有無をテストします。
iswspace[4]	wctype.h	int iswspace(wint_t wc);	iswalnum が false で、インプリメンテーションで定義された一連のワイド文字に対応する、ワイド文字の有無をチェックします。
iswupper[4]	wctype.h	int iswupper(wint_t wc);	大文字のワイド文字の有無をチェックします。
iswxdigit[4]	wctype.h	int iswxdigit(wint_t wc);	16 進数の文字の有無をチェックします。
isxdigit[4]	wctype.h	int isxdigit(int *c*);	*c* が 16 進数であるかどうかをテストします。
j0	math.h	double j0(double *x*);	最初の順序の種類が 0 のベッセル関数値を計算します。
j1	math.h	double j1(double *x*);	最初の順序の種類が 1 のベッセル関数値を計算します。
jn	math.h	double jn(int *n*, double *x*);	最初の順序の種類が *n* のベッセル関数値を計算します。
labs	stdlib.h	long int labs(long int *n*);	*n* の絶対値を計算します。
ldexp	math.h	double ldexp(double *x*, int *exp*);	乗算された *x* (2 の *exp* 乗) の値を戻します。
ldiv	stdlib.h	ldiv_t ldiv(long int *numerator*, long int *denominator*);	*numerator* を *denominator* で割った商および余りを計算します。
localeconv	locale.h	struct lconv *localeconv(void);	現行のロケールに従って、struct lconv の数量をフォーマット設定します。
localtime	time.h	struct tm *localtime(const time_t **timeval*);	*timeval* を tm 型の構造体に変換します。
localtime64	time.h	struct tm *localtime64(const time64_t **timeval*);	*timeval* を tm 型の構造体に変換します。
localtime_r	time.h	struct tm *localtime_r(const time_t *timeval, struct tm *result);	*time* 値を *tm* 型の構造体に変換します。 (localtime の再始動可能バージョン。)

表 36. 標準 C ライブラリー関数 (続き)

関数	システム・インクルード・ファイル	関数プロトタイプ	説明
localtime64_r	time.h	struct tm *localtime64_r(const time64_t *timeval, struct tm *result);	*time* 値を *tm* 型の構造体に変換します。 (localtime64 の再始動可能バージョン。)
log	math.h	double log(double *x*);	*x* の自然対数を計算します。
log10	math.h	double log10(double *x*);	10 を底とする *x* の対数を計算します。
longjmp	setjmp.h	void longjmp(jmp_buf *env*, int *value*);	以前に setjmp 関数によって *env* に設定されたスタック環境を復元します。
malloc	stdlib.h	void *malloc(size_t *size*);	ストレージのブロックを予約します。
mblen	stdlib.h	int mblen(const char **string*, size_t *n*);	マルチバイト文字 *string* の長さを判別します。
mbrlen[4]	wchar.h	int mbrlen (const char *s, size_t n, mbstate_t *ps);	マルチバイト文字の長さを判別します。 (mblen の再始動可能バージョン。)
mbrtowc[4]	wchar.h	int mbrtowc (wchar_t *pwc, const char *s, size_t n, mbstate_t *ps);	マルチバイト文字をワイド文字に変換します。(mbtowc の再始動可能バージョン。)
mbsinit[4]	wchar.h	int mbsinit (const mbstate_t *ps);	状態オブジェクト **ps* が初期状態であるかどうかをテストします。
mbsrtowcs[4]	wchar.h	size_t mbsrtowc (wchar_t *dst, const char **src, size_t len, mbstate_t *ps);	マルチバイト・ストリングをワイド文字ストリングに変換します。 (mbstowcs の再始動可能バージョン。)
mbstowcs	stdlib.h	size_t mbstowcs(wchar_t **pwc*, const char **string*, size_t *n*);	*string* のマルチバイト文字を、対応する wchar_t コードに変換し、*n* 以下のコードを *pwc* に保管します。
mbtowc	stdlib.h	int mbtowc(wchar_t **pwc*, const char **string*, size_t *n*);	最初の *n* バイトのマルチバイト文字 *string* に対応する wchar_t コードを、wchar_t 文字 *pwc* に保管します。
memchr	string.h	void *memchr(const void **buf*, int *c*, size_t *count*);	*buf* の最初の *count* バイトを検索して、符号なし文字に変換される *c* が最初に現れる位置を調べます。
memcmp	string.h	int memcmp(const void **buf1*, const void **buf2*, size_t *count*);	*buf1* と *buf2* を、最大 *count* バイトまで比較します。
memcpy	string.h	void *memcpy(void **dest*, const void **src*, size_t *count*);	*count* バイト分の *src* を、*dest* にコピーします。
memmove	string.h	void *memmove(void **dest*, const void **src*, size_t *count*);	*count* バイト分の *src* を、*dest* にコピーします。オーバーラップするオブジェクト間でのコピーを許可します。
memset	string.h	void *memset(void **dest*, int *c*, size_t *count*);	*count* バイト分の *dest* を、値 *c* に設定します。
mktime	time.h	time_t mktime(struct tm **time*);	ローカル *time* を、カレンダー時間に変換します。

表 36. 標準 C ライブラリー関数 (続き)

関数	システム・インクルード・ファイル	関数プロトタイプ	説明
mktime64	time.h	time64_t mktime64(struct tm *time);	ローカル time を、カレンダー時間に変換します。
modf	math.h	double modf(double x, double *intptr);	浮動小数点値の x を小数と整数の部分に分けます。
nextafter	math.h	double nextafter(double x, double y);	y の方向で x の後にくる、次の表現可能な値を計算します。
nextafterl	math.h	long double nextafterl(long double x, long double y);	y の方向で x の後にくる、次の表現可能な値を計算します。
nexttoward	math.h	double nexttoward(double x, long double y);	y の方向で x の後にくる、次の表現可能な値を計算します。
nexttowardl	math.h	long double nexttowardl(long double x, long double y);	y の方向で x の後にくる、次の表現可能な値を計算します。
nl_langinfo[4]	langinfo.h	char *nl_langinfo(nl_item item);	item が指定した要求情報が記述されているストリングを、現行ロケールから検索します。
perror	stdio.h	void perror(const char *string);	エラー・メッセージを stderr に出力します。
pow	math.h	double pow(double x, double y);	x の y 乗の値を計算します。
printf	stdio.h	int printf(const char *format-string, arg-list);	文字および値をフォーマット設定し、stdout にプリントします。
putc[1]	stdio.h	int putc(int c, FILE *stream);	出力 stream に c をプリントします。
putchar[1]	stdio.h	int putchar(int c);	stdout に c をプリントします。
putenv	stdlib.h	int *putenv(const char *varname);	既存の変数を変更するか、新しい変数を作成することにより、環境変数の値を設定します。
puts	stdio.h	int puts(const char *string);	ストリングを stdout に出力します。
putwc[6]	stdio.h wchar.h	wint_t putwchar(wchar_t wc, FILE *stream);	ワイド文字 wc をマルチバイト文字に変換し、現在位置のストリームに書き込みます。
putwchar[6]	wchar.h	wint_t putwchar(wchar_t wc);	ワイド文字 wc をマルチバイト文字に変換し、stdout に書き込みます。
qsort	stdlib.h	void qsort(void *base, size_t num, size_t width, int(*compare)(const void *element1, const void *element2));	num エレメント (それぞれのサイズは width バイト) の配列のクイック・ソートを実行します。
raise	signal.h	int raise(int sig);	実行中のプログラムに、シグナル sig を送信します。
rand	stdlib.h	int rand(void);	疑似乱数整数を戻します。
rand_r	stdlib.h	int rand_r(void);	疑似乱数整数を戻します。 (再始動可能バージョン)

表 36. 標準 C ライブラリー関数 (続き)

関数	システム・インクルード・ファイル	関数プロトタイプ	説明
realloc	stdlib.h	void *realloc(void **ptr*, size_t *size*);	直前に予約されていたストレージ・ブロックの *size* を変更します。
regcomp	regex.h	int regcomp(regex_t **preg*, const char **pattern*, int *cflags*);	*pattern* が示す送信元正規表現を、実行可能バージョンにコンパイルして、*preg* が示す場所に保管します。
regerror	regex.h	size_t regerror(int *errcode*, const regex_t **preg*, char **errbuf*, size_t *errbuf_size*);	正規表現 *preg* 用のエラー・コード *errcode* の記述を検索します。
regexec	regex.h	int regexec(const regex_t **preg*, const char **string*, size_t *nmatch*, regmatch_t **pmatch*, int *eflags*);	ヌル終了ストリング *string* とコンパイル済み正規表現 *preg* とを比較して、両者の間に一致箇所を見つけます。
regfree	regex.h	void regfree(regex_t *preg);	regcomp により割り当てられたメモリーをすべて解放し、正規表現 *preg* をインプリメントします。
remove	stdio.h	int remove(const char **filename*);	*filename* で指定したファイルを削除します。
rename	stdio.h	int rename(const char **oldname*, const char **newname*);	指定されたファイルを名前変更します。
rewind[1]	stdio.h	void rewind(FILE **stream*);	*stream* に関連付けられたファイル・ポインターを、ファイルの先頭に位置変更します。
scanf	stdio.h	int scanf(const char **format-string*, *arg-list*);	stdin から *arg-list* が指定した場所へ、データを読み取ります。
setbuf	stdio.h	void setbuf(FILE **stream*, char **buffer*);	*stream* のバッファリングを制御します。
setjmp	setjmp.h	int setjmp(jmp_buf *env*);	今後 longjmp によって復元できるスタック環境を保存します。
setlocale	locale.h	char *setlocale(int *category*, const char **locale*);	*locale* で定義された変数について、変更または照会を行います。
setvbuf	stdio.h	int setvbuf(FILE **stream*, char **buf*, int *type*, size_t *size*);	*stream* のバッファリングおよびバッファーの *size* を制御します。
signal	signal.h	void(*signal(int *sig*, void(**func*)(int)))(int);	シグナル sig のシグナル・ハンドラーとして、func を登録します。
sin	math.h	double sin(double *x*);	*x* のサインを計算します。
sinh	math.h	double sinh(double *x*);	*x* の双曲線サインを計算します。
snprintf	stdio.h	int snprintf(char *outbuf, size_t n, const char*, ...)	sprintf と同じです。ただし、outbuf に n 文字書き込まれると、関数が停止します。
sprintf	stdio.h	int sprintf(char **buffer*, const char **format-string*, *arg-list*);	文字および値をフォーマット設定し、*buffer* に保管します。
sqrt	math.h	double sqrt(double *x*);	*x* の平方根を計算します。

表 36. 標準 C ライブラリー関数 (続き)

関数	システム・インクルード・ファイル	関数プロトタイプ	説明
srand	stdlib.h	void srand(unsigned int *seed*);	疑似乱数生成プログラムの *seed* を設定します。
sscanf	stdio.h	int sscanf(const char **buffer*, const char **format*, *arg-list*);	*buffer* から *arg-list* が指定した場所に、データを読み取ります。
strcasecmp	strings.h	int srtcasecmp(const char *string1, const char *string2);	大/小文字を区別しないで、ストリングを比較します。
strcat	string.h	char *strcat(char **string1*, const char **string2*);	*string2* を *string1* に連結します。
strchr	string.h	char *strchr(const char **string*, int *c*);	*string* 内で *c* が最初に現れる位置を見つけます。
strcmp	string.h	int strcmp(const char **string1*, const char **string2*);	*string1* の値を *string2* と比較します。
strcoll	string.h	int strcoll(const char **string1*, const char **string2*);	現行ロケールで照合シーケンスを使用して、2 つのストリングを比較します。
strcpy	string.h	char *strcpy(char **string1*, const char **string2*);	*string2* を *string1* にコピーします。
strcspn	string.h	size_t strcspn(const char **string1*, const char **string2*);	*string2* に含まれていない文字から構成される、*string1* の初期サブストリングの長さを戻します。
strerror	string.h	char *strerror(int *errnum*);	*errnum* のエラー番号を、エラー・メッセージ・ストリングにマップします。
strfmon[4]	wchar.h	int strfmon (char *s, size_t maxsize, const char *format, ...);	通貨の値をストリングに変換します。
strftime	time.h	size_t strftime (char **dest*, size_t *maxsize*, const char **format*, const struct tm **timeptr*);	*format* が判別したストリングに従って、*dest* が示す配列で文字を保管します。
strlen	string.h	size_t strlen(const char **string*);	*string* の長さを計算します。
strncasecmp	strings.h	int strncasecmp(const char *string1, const char *string2, size_t count);	大/小文字を区別しないで、ストリングを比較します。
strncat	string.h	char *strncat(char **string1*, const char **string2*, size_t *count*);	最大 *count* 文字までの *string2* を、*string1* に連結します。
strncmp	string.h	int strncmp(const char **string1*, const char **string2*, size_t *count*);	*string1* と *string2* を、最大 *count* 文字まで比較します。
strncpy	string.h	char *strncpy(char **string1*, const char **string2*, size_t *count*);	最大 *count* 文字までの *string2* を、*string1* にコピーします。
strpbrk	string.h	char *strpbrk(const char **string1*, const char **string2*);	*string2* 内の文字が *string1* 内で最初に現れる位置を見つけます。

表 36. 標準 C ライブラリー関数 (続き)

関数	システム・インクルード・ファイル	関数プロトタイプ	説明
strptime[4]	time.h	char *strptime(const char *buf, const char *format, struct tm *tm);	日付および時刻の変換
strrchr	string.h	char *strrchr(const char **string*, int *c*);	*string* 内で *c* が最後に現れる位置を見つけます。
strspn	string.h	size_t strspn(const char **string1*, const char **string2*);	*string2* に含まれている文字から構成される、*string1* の初期サブストリングの長さを戻します。
strstr	string.h	char *strstr(const char **string1*, const char **string2*);	*string1* において *string2* が最初に現れる位置へのポインターを戻します。
strtod	stdlib.h	double strtod(const char **nptr*, char ***endptr*);	*nptr* を倍精度の値に変換します。
strtod32	stdlib.h	_Decimal32 strtod32(const char **nptr*, char ***endptr*);	*nptr* を、単精度の 10 進浮動小数点値に変換します。
strtod64	stdlib.h	_Decimal64 strtod64(const char **nptr*, char ***endptr*);	*nptr* を、倍精度の 10 進浮動小数点値に変換します。
strtod128	stdlib.h	_Decimal128 strtod128(const char **nptr*, char ***endptr*);	*nptr* を、4 倍精度の 10 進浮動小数点値に変換します。
strtof	stdlib.h	float strtof(const char **nptr*, char ***endptr*);	*nptr* を float 型の値に変換します。
strtok	string.h	char *strtok(char **string1*, const char **string2*);	*string2* 内の次の区切り文字で区切られている、*string1* の次のトークンの位置を見つけます。
strtok_r	string.h	char *strtok_r(char **string*, const char **seps*, char ***lasts*);	*seps* 内の次の区切り文字で区切られている、*string* 内の次のトークンの位置を見つけます。 (strtok の再始動可能バージョン。)
strtol	stdlib.h	long int strtol(const char **nptr*, char ***endptr*, int *base*);	*nptr* を、符号付き long 型整数に変換します。
strtold	stdlib.h	long double strtold(const char **nptr*, char ***endptr*);	*nptr* を long double 型の値に変換します。
strtoul	stdlib.h	unsigned long int strtoul(const char **string1*, char ***string2*, int *base*);	*string1* を、符号なしの long 型整数に変換します。
strxfrm	string.h	size_t strxfrm(char **string1*, const char **string2*, size_t *count*);	*string2* を変換して、結果を *string1* 内に配置します。プログラムの現行ロケールによって、変換の判別が行われます。
swprintf	wchar.h	int swprintf(wchar_t **wcsbuffer*, size_t *n*, const wchar_t **format*, *arg-list*);	一連のワイド文字と値をフォーマット設定して、ワイド文字バッファー *wcsbuffer* に保管します。

表 36. 標準 C ライブラリー関数 (続き)

関数	システム・インクルード・ファイル	関数プロトタイプ	説明
swscanf	wchar.h	int swscanf (const wchar_t **buffer*, const wchar_t **format*, *arg-list*)	*buffer* から *arg-list* が指定した場所に、データを読み取ります。
system	stdlib.h	int system(const char **string*);	システム・コマンド分析プログラムに *string* を受け渡します。
tan	math.h	double tan(double *x*);	*x* のタンジェントを計算します。
tanh	math.h	double tanh(double *x*);	*x* の双曲線タンジェントを計算します。
time	time.h	time_t time(time_t **timeptr*);	現在のカレンダー時間を戻します。
time64	time.h	time64_t time64(time64_t **timeptr*);	現在のカレンダー時間を戻します。
tmpfile	stdio.h	FILE *tmpfile(void);	一時バイナリー・ファイルを作成して、オープンします。
tmpnam	stdio.h	char *tmpnam(char **string*);	一時ファイル名を生成します。
toascii	ctype.h	int toascii(int *c*);	7 ビットの US-ASCII 文字セットの文字に *c* を変換します。
tolower	ctype.h	int tolower(int *c*);	*c* を小文字に変換します。
toupper	ctype.h	int toupper(int *c*);	*c* を大文字に変換します。
towctrans	wctype.h	wint_t towctrans(wint_t *wc*, wctrans_t *desc*);	*desc* で記述されたマッピングを基に、ワイド文字 *wc* を変換します。
towlower[4]	wctype.h	wint_t towlower(wint_t wc);	大文字を小文字に変換します。
towupper[4]	wctype.h	wint_t towupper(wint_t wc);	小文字を大文字に変換します。
ungetc[1]	stdio.h	int ungetc(int *c*, FILE **stream*);	*c* を入力 *stream* にプッシュ・バックします。
ungetwc[6]	stdio.h wchar.h	wint_t ungetwc(wint_t *wc*, FILE **stream*);	ワイド文字 *wc* を入力ストリームにプッシュ・バックします。
va_arg	stdarg.h	*var_type* va_arg(va_list *arg_ptr*, var_type);	ある引数の値を戻し、その次の引数を指すように *arg_ptr* を変更します。
va_end	stdarg.h	void *va_end*(va_list *arg_ptr*);	変数の引数リスト処理から正常に戻るようにします。
va_start	stdarg.h	void *va_start*(va_list *arg_ptr, variable_name*);	*arg_ptr* を初期化して、以降 *va_arg* および *va_end* によって使用できるようにします。
vfprintf	stdio.h stdarg.h	int vfprintf(FILE **stream*, const char **format*, *va_list arg_ptr*);	文字のフォーマット設定を行い、引数の可変値を使用して出力 *stream* に出力します。
vfscanf	stdio.h stdarg.h	int vfscanf(FILE **stream*, const char **format*, va_list *arg_ptr*);	指定されたストリームから、引数の可変値により指定されたロケーションに、データを読み取ります。
vfwprintf[6]	stdarg.h stdio.h wchar.h	int vfwprintf(FILE **stream*, const wchar_t **format*, *va_list arg*);	fwprintf と同じです。ただし、変数の引数リストが *arg* によって置き換えられる点が異なります。

表 36. 標準 C ライブラリー関数 (続き)

関数	システム・インクルード・ファイル	関数プロトタイプ	説明
vfwscanf	stdio.h stdarg.h	int vfwscanf(FILE **stream*, const wchar_t **format*, *va_list arg_ptr*);	指定されたストリームから、引数の可変値により指定されたロケーションに、ワイド・データを読み取ります。
vprintf	stdio.h stdarg.h	int vprintf(const char **format*, va_list *arg_ptr*);	引数の可変値を使用して、文字をフォーマット設定し、stdout にプリントします。
vscanf	stdio.h stdarg.h	int vscanf(const char **format*, *va_list arg_ptr*);	stdin から引数の可変値が指定した場所に、データを読み取ります。
vsprintf	stdio.h stdarg.h	int vsprintf(char **target-string*, const char **format*, va_list *arg_ptr*);	引数の可変値を使用して、文字をフォーマット設定し、バッファーに保管します。
vsnprintf	stdio.h	int vsnprintf(char *outbuf, size_t n, const char*, va_list);	vsprintf と同じです。ただし、outbuf に n 文字書き込まれた後に関数が停止する点が異なります。
vsscanf	stdio.h stdarg.h	int vsscanf(const char**buffer*, const char **format*, *va_list arg_ptr*);	バッファーから、引数の可変値により指定されたロケーションに、データを書き込みます。
vswprintf	stdarg.h wchar.h	int vswprintf(wchar_t **wcsbuffer*, size_t *n*, const wchar_t **format*, *va_list arg*);	一連のワイド文字および値をフォーマット設定し、バッファー *wcsbuffer* に保管します。
vswscanf	stdio.h wchar.h	int vswscanf(const wchar_t **buffer*, const wchar_t **format*, *va_list arg_ptr*);	バッファーから、引数の可変値により指定されたロケーションに、ワイド・データを読み取ります。
vwprintf[6]	stdarg.h wchar.h	int vwprintf(const wchar_t **format*, *va_list arg*);	wprintf と同じです。ただし、変数の引数リストが *arg* により置き換えられる点が異なります。
vwscanf	stdio.h wchar.h	int vwscanf(const wchar_t **format*, *va_list arg_ptr*);	stdin から引数の可変値が指定した場所に、ワイド・データを読み取ります。
wcrtomb[4]	wchar.h	int wcrtomb (char *s, wchar_t wchar, mbstate_t *pss);	ワイド文字をマルチバイト文字に変換します。 (wctomb の再始動可能バージョン)
wcscat	wchar.h	wchar_t *wcscat(wchar_t **string1*, const wchar_t **string2*);	*string2* が指すストリングのコピーを *string1* が指すストリングの終わりに付加します。
wcschr	wchar.h	wchar_t *wcschr(const wchar_t **string*, wchar_t *character*);	*string* が示すワイド文字ストリングを検索して、*character* が出現しているかどうかを調べます。
wcscmp	wchar.h	int wcscmp(const wchar_t **string1*, const wchar_t **string2*);	2 つのワイド文字ストリング **string1* と **string2* を比較します。
wcscoll[4]	wchar.h	int wcscoll (const wchar_t *wcs1, const wchar_t *wcs2);	現行ロケールで照合シーケンスを使用して、2 つのワイド文字ストリングを比較します。

表 36. 標準 C ライブラリー関数 (続き)

関数	システム・インクルード・ファイル	関数プロトタイプ	説明
wcscpy	wchar.h	wchar_t *wcscpy(wchar_t *string1*, const wchar_t *string2*);	**string2* の内容 (最後の wchar_t null 文字を含む) を **string1* にコピーします。
wcscspn	wchar.h	size_t wcscspn(const wchar_t *string1*, const wchar_t *string2*);	**string2* が示すストリングに現れない、**string1* が示すストリングの初期セグメントにおける、wchar_t の文字数を判別します。
wcsftime	wchar.h	size_t wcsftime(wchar_t *wdest*, size_t *maxsize*, const wchar_t *format*, const struct tm *timeptr*);	*timeptr* 構造体の時間および日付の仕様を、ワイド文字ストリングに変換します。
wcslen	wchar.h	size_t wcslen(const wchar_t *string*);	*string* が示すストリングのワイド文字数を計算します。
wcslocaleconv	locale.h	struct wcslconv *wcslocaleconv(void);	現行ロケールに従って struct wcslconv の数量のフォーマット設定を行います。
wcsncat	wchar.h	wchar_t *wcsncat(wchar_t *string1*, const wchar_t *string2*, size_t *count*);	*string2* のワイド文字を、最長 *count* 文字分 *string1* の最後に追加し、その結果に wchar_t null 文字を追加します。
wcsncmp	wchar.h	int wcsncmp(const wchar_t *string1*, const wchar_t *string2*, size_t *count*);	*string1* から *string2* 間の最大の *count* ワイド文字までを比較します。
wcsncpy	wchar.h	wchar_t *wcsncpy(wchar_t *string1*, const wchar_t *string2*, size_t *count*);	*string2* から *string1* 間の最大 *count* ワイド文字までをコピーします。
wcspbrk	wchar.h	wchar_t *wcspbrk(const wchar_t *string1*, const wchar_t *string2*);	*string2* が示すストリングの任意のワイド文字が、*string1* が指したストリング内で最初に出現する位置を見つけます。
wcsptime	wchar.h	wchar_t *wcsptime(const wchar_t *buf, const wchar_t *format, struct tm *tm);	日付および時刻の変換。strptime() と同じですが、ワイド文字を使用する点が異なります。
wcsrchr	wchar.h	wchar_t *wcsrchr(const wchar_t *string*, wchar_t *character*);	*string* が指すストリング内の *character* の最後のオカレンスの位置を指定します。
wcsrtombs[4]	wchar.h	size_t wcsrtombs (char *dst, const wchar_t **src, size_t len, mbstate_t *ps);	ワイド文字ストリングをマルチバイト・ストリングに変換します。(wcstombs の再始動可能バージョン)
wcsspn	wchar.h	size_t wcsspn(const wchar_t *string1*, const wchar_t *string2*);	*string1* が指すストリングの初期セグメントにあって、すべて *string2* が指すストリングのワイド文字から成るワイド文字数を計算します。
wcsstr	wchar.h	wchar_t *wcsstr(const wchar_t *wcs1*, const wchar_t *wcs2*);	*wcs1* 内で *wcs2* が最初に現れる位置を見つけます。

表 36. 標準 C ライブラリー関数 (続き)

関数	システム・インクルード・ファイル	関数プロトタイプ	説明
wcstod	wchar.h	double wcstod(const wchar_t **nptr*, wchar_t ***endptr*);	*nptr* が指すワイド文字ストリングの初期部分を、double 型の値に変換します。
wcstod32	wchar.h	_Decimal32 wcstod32(const wchar_t **nptr*, wchar_t ***endptr*);	*nptr* が指すワイド文字ストリングの初期部分を、単精度の 10 進浮動小数点値に変換します。
wcstod64	wchar.h	_Decimal64 wcstod64(const wchar_t **nptr*, wchar_t ***endptr*);	*nptr* が指すワイド文字ストリングの初期部分を、倍精度の 10 進浮動小数点値に変換します。
wcstod128	wchar.h	_Decimal128 wcstod128(const wchar_t **nptr*, wchar_t ***endptr*);	*nptr* が指すワイド文字ストリングの初期部分を、4 倍精度の 10 進浮動小数点値に変換します。
wcstok	wchar.h	wchar_t *wcstok(wchar_t **wcs1*, const wchar_t **wcs2*, wchar_t ***ptr*)	*wcs1* を一連のトークンに分割し、*wcs2* が指すワイド・ストリングのワイド文字によって、各トークンを区切ります。
wcstol	wchar.h	long int wcstol(const wchar_t **nptr*, wchar_t ***endptr*, int *base*);	*nptr* が指すワイド文字ストリングの初期部分を、long 型の整数値に変換します。
wcstombs	stdlib.h	size_t wcstombs(char *dest, const wchar_t **string*, size_t *count*);	wchar_t の *string* を、マルチバイト・ストリングの *dest* に変換します。
wcstoul	wchar.h	unsigned long int wcstoul(const wchar_t **nptr*, wchar_t ***endptr*, int *base*);	*nptr* が指すワイド文字ストリングの初期部分を、符号なしの long 型整数値に変換します。
wcsxfrm[4]	wchar.h	size_t wcsxfrm (wchar_t **wcs1*, const wchar_t **wcs2*, size_t *n*);	ワイド文字ストリングを、文字照合重みを表す値に変換し、結果のワイド文字ストリングを配列に配置します。
wctob	stdarg.h wchar.h	int wctob(wint_t *wc*);	初期シフト状態時にマルチバイト文字表現が単一バイトである拡張文字セットのメンバーに、*wc* が対応しているかどうかを判別します。
wctomb	stdlib.h	int wctomb(char **string*, wchar_t *character*);	*character* の wchar_t 値を、マルチバイトの *string* に変換します。
wctrans	wctype.h	wctrans_t wctrans(const char **property*);	ストリング引数プロパティーによって識別されるワイド文字間のマッピングを表す、wctrans_t 型の値を構成します。
wctype[4]	wchar.h	wctype_t wctype (const char **property*);	文字プロパティー分類のハンドルを取得します。
wcwidth	wchar.h	int wcswidth(const wchar_t **pwcs*, size_t *n*);	ワイド文字ストリングの表示幅を判別します。
wmemchr	wchar.h	wchar_t *wmemchr(const wchar_t **s*, wchar_t *c*, size_t *n*);	*s* が示すオブジェクトの最初の *n* ワイド文字内で、*c* が最初に現れる位置を見つけます。

表 36. 標準 C ライブラリー関数 (続き)

関数	システム・インクルード・ファイル	関数プロトタイプ	説明
wmemcmp	wchar.h	int wmemcmp(const wchar_t *s1, const wchar_t *s2, size_t n);	s1 が示すオブジェクトの最初の n ワイド文字と、s2 が示すオブジェクトの最初の n 文字とを比較します。
wmemcpy	wchar.h	wchar_t *wmemcpy(wchar_t *s1, const wchar_t *s2, size_t n);	n ワイド文字を、s2 が示すオブジェクトから s1 が示すオブジェクトにコピーします。
wmemmove	wchar.h	wchar_t *wmemmove(wchar_t *s1, const wchar_t *s2, size_t n);	n ワイド文字を、s2 が示すオブジェクトから s1 が示すオブジェクトにコピーします。
wmemset	wchar.h	wchar_t *wmemset(wchar_t *s, wchar_t c, size_t n);	c の値を、s が示すオブジェクトの最初の n ワイド文字それぞれにコピーします。
wprintf[6]	wchar.h	int wprintf(const wchar_t *format, arg-list);	wprintf の引数の前に挿入された引数 stdout を持つ fwprintf に相当します。
wscanf[6]	wchar.h	int wscanf(const wchar_t *format, arg-list);	wscanf の引数の前に挿入された引数 stdin を持つ fwscanf に相当します。
y0	math.h	double y0(double x);	2 番目の順序の種類が 0 のベッセル関数値を計算します。
y1	math.h	double y1(double x);	2 番目の順序の種類が 1 のベッセル関数値を計算します。
yn	math.h	double yn(int n, double x);	2 番目の順序の種類が n のベッセル関数値を計算します。

注: [1] この関数は、type=record でオープンされたファイルではサポートされません。
注: [2] この関数は、type=record かつ mode=ab+、rb+、wb+ のいずれかでオープンされたファイルではサポートされません。
注: [3] ILE C コンパイラーは、完全にバッファーに入れられたストリームおよび行バッファリングされたストリームのみをサポートします。ブロックおよび行が、オープンされたファイルのレコード長に等しくなるため、完全にバッファーに入れられたストリームおよび行バッファリングされたストリングも、同様にサポートされます。 `setbuf()` および `setvbuf()` 関数には、影響はありません。
注: [4] コンパイル・コマンドで LOCALETYPE(*CLD) が指定されている場合、この関数は使用できません。
注: [5] CRTCMOD コマンドまたは CRTBNDC コマンドで SYSIFCOPT(*IFSIO) が指定されている場合のみ、この関数が使用できます。
注: [6] コンパイル・コマンドで LOCALETYPE(*CLD) または SYSIFCOPT(*NOIFSIO) のいずれかが指定されている場合、この関数は使用できません。

ILE C ライブラリーの C ライブラリー関数への拡張機能に関する表

この表では、ILE C ライブラリーの拡張機能をアルファベット順に掲載して、簡単に説明します。この表には、インクルード・ファイル名、および各関数の関数プロトタイプが記載されています。

表 37. ILE C ライブラリー拡張機能

関数	システム・インクルード・ファイル	関数プロトタイプ	説明
_C_Get _Ssn_Handle	stdio.h	_SSN_Handle_T _C_Get_Ssn_Handle (void);	DSM API で使用するために、C セッションにハンドルを戻します。
_C_Quickpool _Debug	stdio.h	_C_Quickpool_Debug_T _C_Quickpool_Debug (_C_Quickpool_Debug_T *newval);	高速プールのメモリー特性を変更します。
_C_Quickpool _Init	stdio.h	int _C_Quickpool_Init(unsigned int numpools, unsigned int *cell_sizes, unsigned int *num_cells);	高速プールのメモリー管理アルゴリズムの使用法を初期設定します。
_C_Quickpool _Report	stdio.h	void _C_Quickpool_Report(void);	現行活動化グループ内で、高速プール・メモリー管理アルゴリズムにより使用されるメモリーのスナップショットを含んだ、スプール・ファイルを生成します。
_C_TS _malloc64	stdlib.h	void *_C_TS_malloc64(unsigned long long int);	_C_TS_malloc と同じです。ただし、ユーザーが単一要求で 2 GB より多くのストレージを求めることができるように、符号なしの long long 型整数を受け取ります。
_C_TS _malloc_info	mallocinfo.h	int _C_TS_malloc_info(struct _C_mallinfo_t *output_record, size_t sizeofoutput);	現在のメモリー使用情報を戻します。
_C_TS _malloc_debug	mallocinfo.h	int _C_TS_malloc_debug(unsigned int dump_level, unsigned int verify_level, struct _C_mallinfo_t *output_record, size_t sizeofoutput);	_C_TS_malloc_info と同じ情報を戻すと共に、C_TS_malloc 関数が使用するメモリー構造体に関する詳細情報のスプール・ファイルを作成します。
_GetExcData	signal.h	void _GetExcData(_INTRPT_Hndlr_Parms_T *parms);	シグナル・ハンドラー内から、例外に関する情報を取得します。
QXXCHGDA	xxdtaa.h	void QXXCHGDA(_DTAA_NAME_T *dtaname*, short int *offset*, short int *len*, char **dtaptr*);	*dtaptr* が指すデータを使用して、*dtaname* で指定された i5/OS データ域を変更します。
QXXDTOP	xxcvt.h	void QXXDTOP(unsigned char **pptr*, int *digits*, int *fraction*, double *value*);	double 型の値を、桁数全体が *digits* で小数桁数が *fraction* のパック 10 進数値に変換します。
QXXDTOZ	xxcvt.h	void QXXDTOZ(unsigned char **zptr*, int *digits*, int *fraction*, double *value*);	double 型の値を、桁数全体が *digits* で小数桁数が *fraction* のゾーン 10 進数値に変換します。
QXXITOP	xxcvt.h	void QXXITOP(unsigned char **pptr*, int *digits*, int *fraction*, int *value*);	整数値をパック 10 進値に変換します。
QXXITOZ	xxcvt.h	void QXXITOZ(unsigned char **zptr*, int *digits*, int *fraction*, int *value*);	整数値をゾーン 10 進値に変換します。
QXXPTOD	xxcvt.h	double QXXPTOD(unsigned char **pptr*, int *digits*, int *fraction*);	パック 10 進数を、桁数全体が *digits* で小数桁数が *fraction* の double 型の値に変換します。
QXXPTOI	xxcvt.h	int QXXPTOI(unsigned char **pptr*, int *digits*, int *fraction*);	パック 10 進数を、桁数全体が *digits* で小数桁数が *fraction* の整数値に変換します。
QXXRTVDA	xxdtaa.h	void QXXRTVDA(_DTAA_NAME_T *dtaname*, short int *offset*, short int *len*, char **dtaptr*);	*dtaname* で指定された i5/OS データ域のコピーを取得します。
QXXZTOD	xxcvt.h	double QXXZTOD(unsigned char **zptr*, int *digits*, int *fraction*);	ゾーン 10 進数を、桁数全体が *digits* で小数桁数が *fraction* の double 型の値に変換します。

表 37. ILE C ライブラリー拡張機能 (続き)

関数	システム・インクルード・ファイル	関数プロトタイプ	説明
QXXZTOI	xxcvt.h	int QXXZTOI(unsigned char *zptr, int digits, int fraction);	ゾーン 10 進数を、桁数全体が *digits* で小数桁数が *fraction* の整数値に変換します。
_Racquire	recio.h	int _Racquire(_RFILE *fp, char *dev);	レコード入出力操作用のデバイスを準備します。
_Rclose	recio.h	int _Rclose(_RFILE *fp);	レコード入出力操作用にオープンしたファイルをクローズします。
_Rcommit	recio.h	int _Rcommit(char *cmtid);	現行トランザクションを完了し、新規コミットメント境界を確立します。
_Rdelete	recio.h	_RIOFB_T *_Rdelete(_RFILE *fp);	現在ロックされているレコードを削除します。
_Rdevatr	xxfdbk.h recio.h	_XXDEV_ATR_T *_Rdevatr(_RFILE *fp, char *pgmdev);	*fp* が参照するファイル、および *pgmdev* が参照する装置について、装置属性フィードバック域のコピーを示すポインターを戻します。
_Rfeod	recio.h	int _Rfeod(_RFILE *fp);	*fp* が参照するファイルについて、強制的にファイル終了状態にします。
_Rfeov	recio.h	int _Rfeov(_RFILE *fp);	*fp* が参照するテープ・ファイルについて、強制的にボリューム終了状態にします。
_Rformat	recio.h	void Rformat(_RFILE *fp, char *fmt);	*fp* が参照するファイルについて、レコード・フォーマットを *fmt* に設定します。
_Rindara	recio.h	void _Rindara (_RFILE *fp, char *indic_buf);	後続のレコード入出力操作で使用する、別の標識領域を設定します。
_Riofbk	recio.h xxfdbk.h	_XXIOFB_T *_Riofbk(_RFILE *fp);	*fp* が参照するファイルについて、入出力フィードバック域のコピーを示すポインターを戻します。
_Rlocate	recio.h	_RIOFB_T *_Rlocate(_RFILE *fp, void *key, int klen_rrn, int opts);	*fp* に関連付けられていて、*key*、*klen_rrn* および *opt* の各パラメーターで指定されたファイルに、レコードを配置します。
_Ropen	recio.h	_RFILE *_Ropen(const char *filename, const char *mode ...);	レコード入出力操作用のファイルをオープンします。
_Ropnfbk	recio.h xxfdbk.h	_XXOPFB_T *_Ropnfbk(_RFILE *fp);	*fp* が参照するファイルについて、オープン・フィードバック域のコピーを示すポインターを戻します。
_Rpgmdev	recio.h	int _Rpgmdev(_RFILE *fp, char *dev);	デフォルトのプログラム装置を設定します。
_Rreadd	recio.h	_RIOFB_T *_Rreadd(_RFILE *fp, void *buf, size_t size, int opts, long rrn);	相対レコード番号でレコードを読み取ります。
_Rreadf	recio.h	_RIOFB_T *_Rreadf(_RFILE *fp, void *buf, size_t size, int opts);	最初のレコードを読み取ります。
_Rreadindv	recio.h	_RIOFB_T *_Rreadindv(_RFILE *fp, void *buf, size_t size, int opts);	送信勧誘された装置からレコードを読み取ります。
_Rreadk	recio.h	_RIOFB_T *_Rreadk(_RFILE *fp, void *buf, size_t size, int opts, void *key, int klen);	*key* を使ってレコードを読み取ります。
_Rreadl	recio.h	_RIOFB_T *_Rreadl(_RFILE *fp, void *buf, size_t size, int opts);	最終レコードを読み取ります。

表 37. ILE C ライブラリー拡張機能 (続き)

関数	システム・インクルード・ファイル	関数プロトタイプ	説明
_Rreadn	recio.h	_RIOFB_T *_Rreadn(_RFILE *fp*, void **buf*, size_t *size*, int *opts*);	次のレコードを読み取ります。
_Rreadnc	recio.h	_RIOFB_T *_Rreadnc(_RFILE *fp*, void **buf*, size_t *size*);	サブファイル内にある、次の変更済みレコードを読み取ります。
_Rreadp	recio.h	_RIOFB_T *_Rreadp(_RFILE *fp*, void **buf*, size_t *size*, int *opts*);	前のレコードを読み取ります。
_Rreads	recio.h	_RIOFB_T *_Rreads(_RFILE *fp*, void **buf*, size_t *size*, int *opts*);	同じレコードを読み取ります。
_Rrelease	recio.h	int _Rrelease(_RFILE *fp*, char **dev*);	指定したデバイスにおいて、レコード入出力操作を行えなくします。
_Rrlslck	recio.h	int _Rrlslck(_RFILE *fp*);	現在ロックされているレコードを解放します。
_Rrollbck	recio.h	int _Rrollbck(void);	最新のコミットメント境界を、現行のコミットメント境界として再確立します。
_Rupdate	recio.h	_RIOFB_T *_Rupdate(_RFILE *fp*, void **buf*, size_t *size*);	現在ロックされているレコードに、更新用の書き込みを行います。
_Rupfb	recio.h	_RIOFB_T *_Rupfb(_RFILE *fp*);	最新のレコード入出力操作に関する情報を使って、フィードバック構造体を更新します。
_Rwrite	recio.h	_RIOFB_T *_Rwrite(_RFILE *fp*, void **buf*, size_t *size*);	ファイルの終わりにレコードを書き込みます。
_Rwrited	recio.h	_RIOFB_T *_Rwrited(_RFILE *fp*, void **buf*, size_t *size*, unsigned long *rrn*);	相対レコード番号でレコードを書き込みます。削除されたレコードにのみ、書き込みを行います。
_Rwriterd	recio.h	_RIOFB_T *_Rwriterd(_RFILE *fp*, void **buf*, size_t *size*);	レコードの読み取りと書き込みを行います。
_Rwrread	recio.h	_RIOFB_T *_Rwrread(_RFILE *fp*, void **inbuf*, size_t in_buf_size, void **out_buf*, size_t out_buf_size);	_Rwriterd と同じ働きをしますが、分離バッファーを入出力データに指定できる点が異なります。
__wcsicmp	wchar.h	int __wcsicmp(const wchar_t *string1, const wchar_t *string2);	大/小文字を区別しないで、ワイド文字ストリングを比較します。
__wcsnicmp	wchar.h	int __wcsnicmp(const wchar_t *string1, const wchar_t *string2, size_t count);	大/小文字を区別しないで、ワイド文字ストリングを比較します。

付録 B. 特記事項

本書は米国 IBM が提供する製品およびサービスについて作成したものです。

本書に記載の製品、サービス、または機能が日本においては提供されていない場合があります。日本で利用可能な製品、サービス、および機能については、日本 IBM の営業担当員にお尋ねください。本書で IBM 製品、プログラム、またはサービスに言及していても、その IBM 製品、プログラム、またはサービスのみが使用可能であることを意味するものではありません。これらに代えて、IBM の知的所有権を侵害することのない、機能的に同等の製品、プログラム、またはサービスを使用することができます。ただし、IBM 以外の製品とプログラムの操作またはサービスの評価および検証は、お客様の責任で行っていただきます。

IBM は、本書に記載されている内容に関して特許権 (特許出願中のものを含む) を保有している場合があります。本書の提供は、お客様にこれらの特許権について実施権を許諾することを意味するものではありません。実施権についてのお問い合わせは、書面にて下記宛先にお送りください。

〒242-8502
神奈川県大和市下鶴間1623番14号
日本アイ・ビー・エム株式会社
法務・知的財産
知的財産権ライセンス渉外

以下の保証は、国または地域の法律に沿わない場合は、適用されません。 IBM およびその直接または間接の子会社は、本書を特定物として現存するままの状態で提供し、商品性の保証、特定目的適合性の保証および法律上の瑕疵担保責任を含むすべての明示もしくは黙示の保証責任を負わないものとします。国または地域によっては、法律の強行規定により、保証責任の制限が禁じられる場合、強行規定の制限を受けるものとします。

この情報には、技術的に不適切な記述や誤植を含む場合があります。本書は定期的に見直され、必要な変更は本書の次版に組み込まれます。 IBM は予告なしに、随時、この文書に記載されている製品またはプログラムに対して、改良または変更を行うことがあります。

本書において IBM 以外の Web サイトに言及している場合がありますが、便宜のため記載しただけであり、決してそれらの Web サイトを推奨するものではありません。それらの Web サイトにある資料は、この IBM 製品の資料の一部ではありません。それらの Web サイトは、お客様の責任でご使用ください。

IBM は、お客様が提供するいかなる情報も、お客様に対してなんら義務も負うことのない、自ら適切と信ずる方法で、使用もしくは配布することができるものとします。

本プログラムのライセンス保持者で、(i) 独自に作成したプログラムとその他のプログラム（本プログラムを含む）との間での情報交換、および (ii) 交換された情報の相互利用を可能にすることを目的として、本プログラムに関する情報を必要とする方は、下記に連絡してください。

IBM Corporation
Software Interoperability Coordinator, Department YBWA
3605 Highway 52 N
Rochester, MN 55901
U.S.A.

© Copyright IBM Corp. 1999, 2010

本プログラムに関する上記の情報は、適切な使用条件の下で使用することができますが、有償の場合もあります。

本書で説明されているライセンス・プログラムまたはその他のライセンス資料は、IBM 所定のプログラム契約の契約条項、IBM プログラムのご使用条件、IBM 機械コードのご使用条件、またはそれと同等の条項に基づいて、IBM より提供されます。

この文書に含まれるいかなるパフォーマンス・データも、管理環境下で決定されたものです。そのため、他の操作環境で得られた結果は、異なる可能性があります。一部の測定が、開発レベルのシステムで行われた可能性がありますが、その測定値が、一般に利用可能なシステムのものと同じである保証はありません。さらに、一部の測定値が、推定値である可能性があります。実際の結果は、異なる可能性があります。お客様は、お客様の特定の環境に適したデータを確かめる必要があります。

IBM 以外の製品に関する情報は、その製品の供給者、出版物、もしくはその他の公に利用可能なソースから入手したものです。IBM は、それらの製品のテストは行っておりません。したがって、他社製品に関する実行性、互換性、またはその他の要求については確証できません。 IBM 以外の製品の性能に関する質問は、それらの製品の供給者にお願いします。

IBM の将来の方向または意向に関する記述については、予告なしに変更または撤回される場合があり、単に目標を示しているものです。

本書には、日常の業務処理で用いられるデータや報告書の例が含まれています。より具体性を与えるために、それらの例には、個人、企業、ブランド、あるいは製品などの名前が含まれている場合があります。これらの名称はすべて架空のものであり、名称や住所が類似する企業が実在しているとしても、それは偶然にすぎません。

著作権使用許諾:

本書には、様々なオペレーティング・プラットフォームでのプログラミング手法を例示するサンプル・アプリケーション・プログラムがソース言語で掲載されています。お客様は、サンプル・プログラムが書かれているオペレーティング・プラットフォームのアプリケーション・プログラミング・インターフェースに準拠したアプリケーション・プログラムの開発、使用、販売、配布を目的として、いかなる形式においても、IBM に対価を支払うことなくこれを複製し、改変し、配布することができます。このサンプル・プログラムは、あらゆる条件下における完全なテストを経ていません。従って IBM は、これらのサンプル・プログラムについて信頼性、利便性もしくは機能性があることをほのめかしたり、保証することはできません。サンプル・プログラムは特定物として現存するままの状態で提供されるものであり、いかなる保証も提供されません。IBM は、当該サンプル・プログラムの使用から生ずるいかなる損害に対しても一切の責任を負いません。

それぞれの複製物、サンプル・プログラムのいかなる部分、またはすべての派生的創作物にも、次のように、著作権表示を入れていただく必要があります。

© (お客様の会社名) (西暦年). このコードの一部は、IBM Corp. のサンプル・プログラムから取られています。 © Copyright IBM Corp. _年を入れる_. .

この情報をソフトコピーでご覧になっている場合は、写真やカラーの図表は表示されない場合があります。

プログラミング・インターフェース情報

この「ILE C/C++ ランタイム・ライブラリー関数」資料には、プログラムを作成するユーザーが IBMi のサービスを使用するためのプログラミング・インターフェースが記述されています。

商標

IBM、IBM ロゴおよび ibm.com は、世界の多くの国で登録された International Business Machines Corp. の商標です。他の製品名およびサービス名は、IBM または各社の商標です。現時点での IBM の商標リストについては、www.ibm.com/legal/copytrade.shtml の「Copyright and trademark information」をご覧ください。

Linux は、Linus Torvalds の米国およびその他の国における商標です。

Java およびすべての Java 関連の商標およびロゴは Sun Microsystems, Inc.の米国およびその他の国における商標です。

UNIX は The Open Group の米国およびその他の国における登録商標です。

他の会社名、製品名およびサービス名等はそれぞれ各社の商標です。

参考文献

IBM i プラットフォームの ILE C/C++ プログラミングに関連したトピックについての詳細は、次の IBM i の資料および IBM i Information Center のトピックを参照してください。

(http://www.ibm.com/systems/i/infocenter/)

- IBM i Information Center の『プログラミング』カテゴリーにあるトピック『アプリケーション・プログラミング・インターフェース』には、アプリケーション・プログラミング・インターフェース (API) を使用する経験豊富なアプリケーション・プログラマーおよびシステム・プログラマー向けの情報が記載されています。
- 「*System i アプリケーション表示プログラミング*, SC88-4031-01」には、DDS を使用したディスプレイの作成および保守、ディスプレイ・ファイルの作成および処理、オンライン・ヘルプ情報の作成、UIM を使用したディスプレイの定義、およびパネル・グループ、レコード、および資料の使用についての説明が記載されています。
- IBM i Information Center の『システム管理』カテゴリーにあるトピック『バックアップおよび回復』には、バックアップおよび回復に関する方針の計画方法、システムのバックアップ方法、テープ・ライブラリーの管理方法、およびデータに対するディスク保護のセットアップ方法についての説明が記載されています。このトピックには、IBM i ナビゲーターへの Backup, Recovery and Media Services プラグインに関する情報、システムの回復についての情報、バックアップおよび回復についてのよくある質問に対する回答も記載されています。
- 「*Recovering your system*, SC41-5304-10」には、IBM i プラットフォームの回復オプションおよび可用性オプションについての一般情報が記載されています。システムで利用可能なオプションについての説明、オプションの比較および対比、オプションに関する詳細情報の記載場所の紹介、などを行っています。
- IBM i Information Center の『プログラミング』カテゴリーにあるトピック『制御言語』には、制御言語コマンドの説明が記載されています。また、このトピックには、オブジェクトおよびライブラリー、CL プログラミング、フロー制御およびプログラム間通信、CL プログラムのオブジェクト処理、および CL プログラム作成に関する一般的な説明を含む、プログラミング上のトピックについての広範囲の説明も記載されています。その他のトピックとしては、事前定義メッセージと即時メッセージおよびメッセージの処理、ユーザー定義のコマンドとメニューの定義と作成、デバッグ・モード、ブレークポイント、追跡、および表示機能を含むアプリケーションのテストなどが入っています。
- 「*Communications Management*, SC41-5406-02」には、通信環境における実行管理機能、通信状況、通信問題のトレースと診断、エラー処理とリカバリー、パフォーマンス、および特定の回線速度とサブシステム・ストレージ情報に関する説明が記載されています。
- IBM i Information Center の『ファイルおよびファイル・システム』カテゴリーには、アプリケーション・プログラムでのファイルの使用に関する情報が記載されています。
- IBM i Information Center の『プログラミング』カテゴリーにあるトピック『グローバリゼーション』には、IBM i 製品のグローバリゼーションおよびマルチリンガル・サポート機能の計画、インストール、構成、および使用に関する情報が記載されています。マルチリンガル・データのデータベース管理、およびマルチリンガル・システムのアプリケーションに関する考慮事項についても説明しています。
- 「*ICF Programming*, SC41-5442-00」のマニュアルには、通信およびシステム間通信機能 (IBM i -ICF) を使用するアプリケーション・プログラムの作成に必要となる情報が記載されています。データ記述仕様 (DDS) キーワード、システム提供の書式、戻りコード、ファイル転送サポート、プログラム例、などに関する情報も記載されています。
- 「*ILE Concepts*, SC41-5606-09」では、IBM i ライセンス・プログラムの統合化言語環境 (Integrated Language Environment) アーキテクチ

© Copyright IBM Corp. 1999, 2010

ャーに関する概念および用語について説明しています。モジュールの作成、バインド、プログラムの実行、プログラムのデバッグ、および例外処理などをカバーするトピックが含まれています。

- IBM i Information Center の『印刷』情報カテゴリーには、印刷の基本情報に加えて、印刷機能の計画方法および構成方法に関する情報も記載されています。
- トピック『印刷の基本』には、IBM i 製品の印刷の要素と概念、プリンター・ファイルと印刷スプーリングのサポート、およびプリンターの接続性に関する具体的な情報が記載されています。
- IBM i Information Center の『セキュリティー』カテゴリーには、システム・セキュリティーのセットアップおよび計画を行う方法、ネットワーク・アプリケーションおよび通信アプリケーションをセキュアにする方法、およびご使用の製品に高度な機密保護機能のある暗号処理機能を追加する方法に関する情報が記載されています。オブジェクト署名および署名検査、識別マッピング、インターネットのセキュリティー・リスクに対するソリューション、などについての情報も紹介しています。
- 「*Security reference*, SC41-5302-11」では、システム・セキュリティー・サポートを使用して、システムおよびデータが適切な権限のないユーザーによって使用されるのを防ぐ方法、データを故意または偶発的な損傷または破壊から保護する方法、セキュリティー情報を最新の状態に保つ方法、およびシステムにセキュリティーをセットアップする方法について説明します。
- IBM i Information Center の『システム管理』カテゴリーには、問題処理に関する情報に加えて、システム装置制御パネル、システムの開始および終了、テープおよびディスケットの使用、プログラム一時修正の処理に関する情報も記載されています。
- 「ILE C/C++ 解説書」には、C/C++ 言語の参照情報が記載されています。
- 「ILE C/C++ コンパイラー参照」には、プリプロセッサー・ステートメントの使用、ILE C/C++ コンパイラーによるマクロ定義およびプラグマ認識、IBM i および QShell 稼働環境でのコマンド行オプション、および IBM i 環境の入出力に関する考慮事項などについての参照情報が記載されています。
- 「*IBM Rational Development Studio for i: ILE C/C++ Programmer's Guide*, SC09-2712-07」には、ILE C 言語を使用したアプリケーションの開発方法に関する情報が記載されています。これには、プログラムの作成、実行、およびデバッグについての説明があります。 また、言語をまたがるプログラムとプロシージャー呼び出し、ロケール、例外処理、データベース、外部記述ファイル、および装置ファイルのプログラミング上の考慮事項が解説されています。 パフォーマンス上のヒントもいくつか説明されています。 付録には、EPM C またはシステム C から ILE C へのソース・コードの移行に関する情報が記載されています。

プログラミング・ユーティリティーについて詳しくは、IBM Publications Center で、以下の書籍を参照してください。

- *AS/400 プログラム開発管理機能 (PDM)*, SC88-5197-00
- *AS/400 適用業務開発ツールセット AS/400 用 画面設計機能 (SDA)*, SD88-5046-00
- *AS/400 適用業務開発ツールセット AS/400 用 原始ステートメント入力ユーティリティー 使用者の手引きと参照*, SD88-5047-00

索引

日本語, 数字, 英字, 特殊文字の順に配列されています。なお, 濁音と半濁音は清音と同等に扱われています。

［ア行］

アクセス・モード　96, 114
インクルード・ファイル
  assert.h　3
  ctype.h　3
  decimal.h　4
  errno.h　4
  except.h　4
  float.h　7
  inttypes.h　7
  limits.h　8
  locale.h　8
  math.h　9
  pointer.h　10
  recio.h　10
  regex.h　13
  setjmp.h　14
  signal.h　15
  stdarg.h　15
  stddef.h　15
  stdint.h　15
  stdio.h　17
  stdlib.h　18
  string.h　19
  time.h　19
  xxcvt.h　20
  xxdtaa.h　21
  xxenv.h　21
  xxfdbk.h　21
インバリアント文字
  16 進表記　549
受け渡し
  定数　547
  変数　548
エラー処理
  関数　23
  ストリーム入出力　100
  assert　46
  clearerr　65
  ferror　100
  perror　236
  strerror　384
エラー標識　100
エラー・マクロ、ストリーム入出力例外のマッピング　535
エラー・メッセージ
  出力　236
オープン
  メッセージ・カタログ　63

［カ行］

書き込み操作
  出力　152
  ストリームからデータ項目　152
  ストリームへの行　251
  ストリングをストリームへ　127
  フォーマット設定　122, 238, 368
  文字を stdout へ　124, 249
  文字をストリームへ　124, 249, 439
各種関数
  assert　46
  getenv　160
  longjmp　201
  perror　236
  putenv　250
  rand　268
  rand_r　268
  setjmp　352
  srand　370
獲得、プログラム装置の　269
型変換
  atof　49
  atoi　50
  atol　52
  strtod　409
  strtol　418
  strtoul　420
  toascii　433
  wcstod　498
  wcstol　503
  wcstoul　508
可変引数関数　32
環境
  関数　34
  情報の検索　187
  対話　34
  表　160
  変数　160, 250
環境変数
  検索　160
  追加　250
  変更　250
疑似乱数
  rand　268
  rand_r　268
  srand　370
疑似ランダム整数　268
逆正弦　45
逆正接　47
逆余弦　40
記録、プログラム終了の　48
クイック・ソート　256
組み込み
  __EXBDY　6
  __VBDY　6
クリア、エラー標識の　65
クローズ
  ストリーム　95
  ファイル　270
  メッセージ・カタログ　60
ケース・マッピング関数　36
計算
  基数 10 の対数　199
  逆正接　47
  逆余弦　40
  計算、次の表現可能な浮動小数点値の　232
  誤差関数　91
  時差　86, 88
  指数関数　93
  自然対数　198
  斜辺　175
  商と剰余　90
  正弦　365
  絶対値　39
  絶対値、long 型整数の　184
  双曲線正弦　366
  双曲線余弦　69
  対数　198
  浮動小数点絶対値　94
  浮動小数点の剰余　113
  余弦　68
計算関数
  ベッセル　53
  abs　39
  acos　40
  asin　45
  atan　47
  atan2　47
  div　90
  erf　91
  erfc　91
  exp　93

© Copyright IBM Corp. 1999, 2010

計算関数 *(続き)*
fabs 94
floor 112
fmod 113
frexp 137
gamma 156
hypot 175
labs 184
ldexp 185
ldiv 186
log 198
log10 199
modf 232
pow 237
sin 365
sinh 366
sqrt 369
tan 427
tanh 428
決定
長さ、マルチバイト文字の 207
表示幅、ワイド文字ストリングの 511, 512
言語照合ストリングの比較 475
検索
環境変数 160
ストリング 375, 401, 407
トークンのストリング 415, 417
bsearch 関数 54
検索、データ域の 264
検索、ロケール情報の 234
検索関数とソート関数 24
検証、条件の 46
現地時間の訂正 192, 194
現地時間の訂正 (再始動可能バージョン) 195, 197
更新ファイル 96
国際化対応 8
コピー
ストリング 380, 397
バイト 223, 226
コンパイル、正規表現の 279

［サ行］

最大
一時ファイル名 17
オープン・ファイル 17
ファイル名 17
再割り振り 276
削除
ファイル 286
レコード 273, 286
作成
一時ファイル 432, 433
ストリームからデータ項目 152
作成 *(続き)*
ストリング 127, 251
フォーマット済みデータ、ストリームへの 122
文字 124, 249
ワイド文字 128, 252, 254
ワイド文字ストリング 130
ワイド文字をストリームへ 149
三角関数
acos 40
asin 45
atan 47
atan2 47
cos 68
cosh 69
sin 365
sinh 366
tan 427
tanh 428
時間
関数 25, 192, 194, 195, 197
現地時間 192
現地時間の訂正 192, 194, 195, 197
時刻 429
変換、構造体からストリングへ 41
変換、構造体からストリングへ (再始動可能バージョン) 43
asctime 41
asctime_r 43
ctime 75
ctime64 77
ctime64_r 80
ctime_r 78
difftime 86
difftime64 88
gmtime 167
gmtime64 169
gmtime64_r 173
gmtime_r 171
localtime64 194
localtime64_r 197
localtime_r 195
mktime 228
mktime64 230
strftime 387
time64 430
シグナル処理 536
指数関数
exp 93
frexp 137
ldexp 185
log 198
log10 199
pow 237
sqrt 369
斜辺 175
終了、プログラムの 92
出力
エラー・メッセージ 236
取得
ハンドル、文字特性種別の 517
ハンドル、文字マッピングの 516
ワイド文字、STDIN から 165
初期ストリング 397
処理、割り込みシグナルの 362
数学関数 24
ストリーム
アクセス・モード 136
オープン 114
書き込み、文字の 124, 249
書き込み行 251
現行のファイル位置を変更 140, 144
更新 114, 136
再オープン 136
ストリングの書き込み 127
データ項目の書き込み 152
定様式 I/O 138, 238, 344, 368, 371
テキスト・モード 136
バイナリー・モード 136
バッファリング 351
付加 114, 136
変換モード 136
変更、ファイル位置の 288
巻き戻し 288
文字の取得不可 439
文字の読み取り 102, 158
読み取り、行の 105, 162
読み取り、データ項目の 132
ストリーム指向 146
ストリーム入出力
fclose 95
feof 99
ferror 100
fflush 101
fgetc 102
fgets 105
fopen 114
fprintf 122
fputc 124
fputs 127
fputwc 128
fputws 130
fread 132
freopen 136
fscanf 138
fseek 140
ftell 144
fwrite 152
getc 158
getchar 158
gets 162
printf 238

ストリーム入出力 *(続き)*
putc 249
putchar 249
puts 251
rewind 288
scanf 344
setbuf 351
setvbuf 360
snprintf 367
sprintf 368
sscanf 371
swprintf 423
swscanf 425
tmpfile 432
ungetc 439
ungetwc 440
va_arg 442
va_end 442
va_start 442
vfprintf 444
vfscanf 446
vfwprintf 447
vfwscanf 449
vprintf 452
vscanf 453
vsnprintf 455
vsprintf 456
vsscanf 458
vswprintf 459
vswscanf 461
vwprintf 463
vwscanf 465
wprintf 527
wscanf 528
ストリーム入出力関数 29
ストリームの再オープン 136
ストリング
検索 375, 401, 407
コピー 380
作成 127
初期設定 397
大/小文字の無視 376, 379, 381
トークンの検索 415, 417
長さ 392
比較 381, 395
変換
整数への 50
浮動小数点に 52
long 型整数に 52
読み取り 105
連結 374
strstr 408
ストリング処理
strcasecmp 373
strcat 374
strchr 375
ストリング処理 *(続き)*
strcmp 376
strcoll 379
strcpy 380
strcspn 381
strlen 392
strncasecmp 393
strncat 394
strncmp 395
strncpy 397
strpbrk 401
strrchr 406
strspn 407
strstr 408
strtod 409
strtok 415
strtok_r 417
strtol 418
strxfrm 422
wcsstr 497
wcstok 502
ストレージの位置決め 134
ストレージ割り振り 58
正弦 365
整数
疑似ランダム 268
正接 427
絶対値
abs() 関数 39
fabs 94
labs 184
設定
値へのバイト 227
ソート
クイック・ソート 256
送信、シグナルの 267

[タ行]

対数関数
log 198
log10 199
追加、データのストリームへの 96
追加、データのファイルへの 96
通貨関数 25
データ型の互換性
CL 544, 547
COBOL 545
ILE COBOL 543
RPG 542, 545
データ型の制限 8
データ項目 132
データ変換
atof() 関数 49
atoi() 関数 50
atol() 関数 52
停止
プログラム 38
定様式 I/O 122
テスト
状態オブジェクト、初期状態であるかどうかの 213
文字プロパティー 179, 182
ワイド 10 進数字 180
ワイド 16 進数字 180
ワイド印字文字 180
ワイド英字 180
ワイド英数字 180
ワイド大文字 180
ワイド空白文字 180
ワイド小文字 180
ワイド制御文字 180
ワイド非英数字 180
ワイド非スペース文字 180
ASCII 値 177
isalnum 176
isalpha 176
iscntrl 176
isdigit 176
isgraph 176
islower 176
isprint 176
ispunct 176
isspace 176
isupper 176
isxdigit 176
テスト、状態オブジェクト、初期状態であるかどうかの 213
デフォルト・メモリー・マネージャー 566
トークン
トークン化、ストリングの 415
strtok 415
strtok_r 417
統合ファイル・システムの errno 値 533
取り消し、ハンドラー理由コード 539

[ナ行]

長さ、変数の 542
長さ関数 392
二分検索 54
入出力エラー 65

[ハ行]

バイナリー・ファイル 116
バッファー
検索 221
コピー 223, 226
比較 222

バッファー *(続き)*
フラッシュ　101
文字の設定　227
割り当て　351
判別、ワイド文字の表示幅の　520
ヒープ・メモリー　564, 573
ヒープ・メモリー・マネージャー　565
比較
ストリング　376, 379, 381, 395
バッファー　222
比較、ストリングの　373, 393
引数リスト関数　32
日付と時刻の変換　402, 491
微分式　25
標識、エラー　65
標準タイプ
FILE　18
非ローカル goto　201, 352
ファイル
位置決め　288
更新　96
最大、オープンされた　17
追加先　96
名前の長さ　17
名前変更　287
ハンドル　111
include　3
ファイル終了標識　65, 99
ファイル処理
remove　286
rename　287
tmpnam　433
ファイルの位置決め　104, 140, 142, 144
ファイル名、一時　17
ファイル名の長さ　17
ファイル・エラー　65
フォーマット設定、データの、ワイド文字としての　149
付加モード
使用、fopen() 関数の　114
プッシュ、文字の　439
プラグマ・プリプロセッサー・ディレクティブ
環境変数　577
高速プール・メモリー・マネージャー　569
デフォルト・メモリー・マネージャー　566
ヒープ・メモリー　564, 573
ヒープ・メモリー・マネージャー　565
メッセージの問題　579
フラッシュ、バッファーの　101
プリプロセッサー・ディレクティブ
環境変数　577
高速プール・メモリー・マネージャー　569
プリプロセッサー・ディレクティブ *(続き)*
デフォルト・メモリー・マネージャー　566
ヒープ・メモリー　564, 573
ヒープ・メモリー・マネージャー　565
メッセージの問題　579
プログラム終了
終了　92
abort　38
atexit　48
プログラムの異常終了　38
プログラムの終了　38
プロセス制御
signal　362
ブロック・サイズ　117
分離、浮動小数点値の　137
ベッセル関数　25, 53
変換
英大/小文字　434
現地時間　228, 230
構造体からストリングへ　41
構造体からストリングへ (再始動可能バージョン)　43
時間　167, 169, 171, 173, 192, 194, 195, 197
時間から文字ストリング　75, 77, 78, 80
時刻　402, 491
ストリングからフォーマット済み日時へ　479
ストリングから浮動小数点へ　49
ストリングを long 型整数値に　52
ストリング・セグメントから 符号なし整数へ　420
整数値へのストリング　50
整数から ASCII 文字セットの文字へ　433
整数からゾーン 10 進数へ　262
整数からパック 10 進数へ　261
ゾーン 10 進数から double へ　265
ゾーン 10 進数から整数へ　266
通貨値からストリングへ　384
パック 10 進数から double へ　262
パック 10 進数から整数へ　263
日付　402, 491
浮動小数点からパック 10 進数へ　259
浮動小数点数から整数と小数部へ　232
マルチバイト文字から wchar_t へ　220
マルチバイト文字からワイド文字へ　210
マルチバイト・ストリングからワイド文字ストリングへ　214
文字ストリングから 10 進浮動小数点へ　412
変換 *(続き)*
文字ストリングから double へ　409
文字ストリングから long 型整数へ　418
ワイド文字から long 型整数へ　503
ワイド文字からバイトへ　513
ワイド文字からマルチバイト文字へ　515
ワイド文字からマルチバイト文字文字へ　467
ワイド文字ストリングから 符号なし long へ　508
ワイド文字ストリングから 10 進浮動小数点へ　499
ワイド文字ストリングから double 型へ　498
ワイド文字ストリングからマルチバイト・ストリングへ　494
ワイド文字の英大/小文字　436
1 バイトからワイド文字に　56
double からゾーン 10 進数へ　260
変換関数
QXXDTOP　259
QXXDTOZ　260
QXXITOP()　261
QXXITOZ　262
QXXPTOD　262
QXXPTOI　263
QXXZTOD　265
QXXZTOI　266
変更
環境変数　250
データ域　258
ファイル位置　140
予約ストレージ・ブロック・サイズ　276

[マ行]

マルチバイト関数
btowc　56
mblen　205
mbrlen　207
mbrtowc　210
mbsinit　213
mbsrtowcs　214
mbstowcs　216
mbtowc　220
towctrans　435
wcrtomb　467
wcscat　471
wcschr　472
wcscmp　474
wcscoll　475
wcscpy　476
wcscspn　477

マルチバイト関数 *(続き)*
wcsicmp 480
wcslen 482
wcslocaleconv 483
wcsncat 484
wcsncmp 485
wcsncpy 487
wcsnicmp 488
wcspbrk 490
wcsrchr 493
wcsrtombs 494
wcsspn 496
wcstombs 505
wcswcs 510
wcswidth 511
wcsxfrm 512
wctob 513
wctomb 515
wctrans 516
wctype 517
wcwidth 520
_wcsicmp 480
_wcsnicmp 488
メッセージの問題 579
メモリー演算
memchr 221
memcmp 222
memcpy 223
memmove 226
memset 227
wmemchr 521
wmemcmp 522
wmemcpy 524
wmemmove 525
wmemset 526
メモリー管理
解放 134
calloc 58
malloc 203
realloc 276
_C_TS_malloc_debug 82
_C_TS_malloc_info 84
メモリー割り振り
解放 134
calloc 58
malloc 203
realloc 276
_C_TS_malloc_debug 82
_C_TS_malloc_info 84
メモリー・オブジェクト関数 33
文字
作成 124, 249
取得不可 439
設定 227
変換
整数への 50
文字 *(続き)*
変換 *(続き)*
浮動小数点に 52
long 型整数に 52
読み取り 102, 158
文字ケースのマッピング
tolower 434
toupper 434
towlower 436
towupper 436
文字種テスト関数 35
文字テスト
文字プロパティー 179, 182
ワイド 10 進数字 180
ワイド 16 進数字 180
ワイド印字文字 180
ワイド英字 180
ワイド英数字 180
ワイド大文字 180
ワイド空白文字 180
ワイド小文字 180
ワイド制御文字 180
ワイド非英数字 180
ワイド非スペース文字 180
ASCII 値 177
isalnum 176
isalpha 176
iscntrl 176
isdigit 176
isgraph 176
islower 176
isprint 176
ispunct 176
isspace 176
isupper 176
isxdigit 176

[ヤ行]

読み取り
行 162
項目 132
ストリーム 105
データ 344
データ、ストリームから、ワイド文字を使用した 153
データ、ワイド文字書式ストリングを使用した 528
フォーマット済みデータ 138
メッセージ 61
文字 158
ワイド文字、ストリームから 107, 163
ワイド文字ストリング、ストリームから 109
読み取り操作
ストリームからデータ項目 132
ストリームからの行 105
ストリームから文字を 158
走査 138
フォーマット設定 138, 344, 371
読み取り、文字の 102
stdin から行 162
stdin からの文字 158
予約ストレージ
malloc 203
realloc 276
_C_TS_malloc_debug 82
_C_TS_malloc_info 84

[ラ行]

ライブラリー関数
エラー処理
clearerr 65
raise 267
strerror 384
_GetExcData 161
各種
assert 46
getenv 160
longjmp 201
perror 236
putenv 250
rand 268
rand_r 268
setjmp 352
srand 370
型変換
atof 49
atoi 50
atol 52
strol 418
strtod 409
strtoul 420
toascii 433
wcstod 498
wcstol 503
wcstoul 508
可変引数処理
va_arg 442
va_end 442
va_start 442
vfprintf 444
vfscanf 446
vfwscanf 449
vprintf 452
vscanf 453
vsnprintf 455
vsprintf 456
vsscanf 458

ライブラリー関数 *(続き)*
  可変引数処理 *(続き)*
    vswscanf 461
    vwscanf 465
  検索
    bsearch 54
    qsort 256
  三角法
    acos 40
    asin 45
    atan 47
    atan2 47
    cos 68
    cosh 69
    sin 365
    sinh 366
    tan 427
    tanh 428
  時間
    現地時間 192
    asctime 41
    asctime_r 43
    clock 67
    ctime 75
    ctime64 77
    ctime64_r 80
    ctime_r 78
    difftime 86
    difftime64 88
    gmtime 167
    gmtime64 169
    gmtime64_r 173
    gmtime_r 171
    localtime64 194
    localtime64_r 197
    localtime_r 195
    mktime 228
    mktime64 230
    strftime 387
    wcsftime 479
  指数
    exp 93
    frexp 137
    ldexp 185
    log 198
    log10 199
    pow 237
  ストリーム入出力
    fclose 95
    feof 99
    ferror 100
    fflush 101
    fgetc 102
    fgetpos 104
    fgets 105
    fgetwc 107
ライブラリー関数 *(続き)*
  ストリーム入出力 *(続き)*
    fgetws 109
    fprintf 122
    fputc 124
    fputs 127
    fputwc 128
    fputws 130
    fread 132
    freopen 136
    fscanf 138
    fseek 140
    fsetpos 142
    ftell 144
    fwide 146
    fwprintf 149
    fwrite 152
    fwscanf 153
    getc 158
    getchar 158
    gets 162
    getwc 163
    getwchar 165
    printf 238
    putc 249
    putchar 249
    puts 251
    putwc 252
    putwchar 254
    scanf 344
    setbuf 351
    setvbuf 360
    sprintf 368
    sscanf 371
    swprintf 423
    swscanf 425
    ungetc 439
    ungetwc 440
    vfprintf 444
    vfscanf 446
    vfwprintf 447
    vfwscanf 449
    vprintf 452
    vscanf 453
    vsnprintf 455
    vsprintf 456
    vsscanf 458
    vswprintf 459
    vswscanf 461
    vwprintf 463
    vwscanf 465
    wprintf 527
    wscanf 528
  ストリング処理
    strcat 374
    strchr 375
ライブラリー関数 *(続き)*
  ストリング処理 *(続き)*
    strcmp 376
    strcoll 379
    strcpy 380
    strcspn 381
    strlen 392
    strncmp 395
    strncpy 397
    strpbrk 401
    strrchr 406
    strspn 407
    strstr 408
    strtod 409
    strtok 415
    strtok_r 417
    strtol 418
    strtoul 420
    strxfrm 422
    wcsstr 497
    wcstok 502
  正規表現
    regcomp 279
    regerror 281
    regexec 283
    regfree 285
  絶対値
    abs 39
    fabs 94
    labs 184
  データ域
    QXXCHGDA 258
    QXXRTVDA 264
  ファイル処理
    fileno 111
    remove 286
    rename 287
    tmpfile 432
    tmpnam 433
  プログラム
    終了 92
    abort 38
    atexit 48
    signal 362
  変換
    QXXDTOP 259
    QXXDTOZ 260
    QXXITOP 261
    QXXITOZ 262
    QXXPTOD 262
    QXXPTOI 263
    QXXZTOD 265
    QXXZTOI 266
    strfmon 384
    strptime 402
    wcsftime 479

ライブラリー関数 *(続き)*
変換 *(続き)*
wcsptime 491
マルチバイト
btowc 56
mblen 205
mbrlen 207
mbrtowc 210
mbsinit 213
mbsrtowcs 214
mbstowcs 216
mbtowc 220
towctrans 435
wcrtomb 467
wcscat 471
wcschr 472
wcscmp 474
wcscoll 475
wcscpy 476
wcscspn 477
wcslen 482
wcslocaleconv 483
wcsncat 484
wcsncmp 485
wcsncpy 487
wcspbrk 490
wcsrchr 493
wcsrtombs 494
wcsspn 496
wcstombs 505
wcswcs 510
wcswidth 511
wcsxfrm 512
wctob 513
wctomb 515
wctrans 516
wctype 517
wcwidth 520
_wcsicmp 480
_wcsnicmp 488
メッセージ・カタログ
catclose 60
catgets 61
catopen 63
メモリー演算
memchr 221
memcmp 222
memcpy 223
memmove 226
memset 227
wmemchr 521
wmemcmp 522
wmemcpy 524
wmemmove 525
wmemset 526
ライブラリー関数 *(続き)*
メモリー管理
解放 134
calloc 58
malloc 203
realloc 276
_C_TS_malloc_debug 82
_C_TS_malloc_info 84
文字ケースのマッピング
tolower 434
toupper 434
towlower 436
towupper 436
文字テスト
isalnum 176
isalpha 176
isascii 177
iscntrl 176
isdigit 176
isgraph 176
islower 176
isprint 176
ispunct 176
isspace 176
isupper 176
iswalnum 180
iswalpha 180
iswcntrl 180
iswctype 182
iswdigit 180
iswgraph 180
iswlower 180
iswprint 180
iswpunct 180
iswspace 180
iswupper 180
iswxdigit 180
isxdigit 176
ロケール
localeconv 187
nl_langinfo 234
setlocale 354
strxfrm 422
math
ベッセル 53
acos 40
asin 45
atan 47
atan2 47
ceil 64
cos 68
cosh 69
div 90
erf 91
erfc 91
floor 112
ライブラリー関数 *(続き)*
math *(続き)*
fmod 113
frexp 137
gamma 156
hypot 175
ldiv 186
log 198
log10 199
modf 232
sin 365
sinh 366
sqrt 369
tan 427
tanh 428
ライブラリーの概要 23
乱数発生ルーチン 268, 370
ランダム・アクセス 140, 144
リダイレクト 136
例外クラス
マッピング 538
リスト 540
レコード形式 117
レコード入出力
_Racquire 269
_Rclose 270
_Rcommit 271
_Rdelete 273
_Rdevatr 275
_Rfeod 290
_Rfeov 291
_Rformat 292
_Rindara 294
_Riofbk 296
_Rlocate 298
_Ropen 301
_Ropnfbk 305
_Rpgmdev 306
_Rreadd 308
_Rreadf 310
_Rreadindv 312
_Rreadk 314
_Rreadl 318
_Rreadn 319
_Rreadnc 322
_Rreadp 324
_Rreads 326
_Rrelease 328
_Rrlslck 329
_Rrollbck 331
_Rupdate 332
_Rupfb 334
_Rwrite 336
_Rwrited 338
_Rwriterd 341
_Rwrread 342

連結、ストリングの　374, 394
ロケール
　情報の検索　234
　設定　354
ロケール関数
　localeconv　187
　setlocale　354
　strxfrm　422
論理エラー　46
論理レコード長　117

［ワ行］

ワイド文字ストリング機能　36
割り込みシグナル　362

A

abort() 関数　38
abs() 関数　39
acos() 関数　40
asctime() 関数　41
asctime_r() 関数　43
asin() 関数　45
assert() 関数　46
assert.h インクルード・ファイル　3
atan2() 関数　47
atan() 関数　47
atexit() 関数　48
atof() 関数　49
atoi() 関数　50
atoll() 関数
　ストリングを long long 型整数値に　52
atol() 関数　52

B

blksize　117
bsearch() 関数　54
btowc() 関数　56
bufsiz 定数　17

C

calloc() 関数　58
catclose() 関数　60
catgets() 関数　61
catopen() 関数　63
ceiling 関数　64
ceil() 関数　64
clearerr　65
CLOCKS_PER_SEC　67
clock() 関数　67
cosh() 関数　69
cos() 関数　68
ctime64() 関数　77
ctime64_r() 関数　80
ctime() 関数　75
ctime_r() 関数　78
ctype 関数　176
ctype.h インクルード・ファイル　3

D

decimal.h インクルード・ファイル　4
difftime64() 関数　88
difftime() 関数　86
divf() 関数　90

E

eofile
　クリア　288
　マクロ　17
　リセット、エラー標識の　65
erfc() 関数　91
erf() 関数　91
errno　4
errno 値、統合ファイル・システムの　533
errno 変数　236
errno マクロ　531
errno.h インクルード・ファイル　4
except.h インクルード・ファイル　4
exit() 関数　92
EXIT_FAILURE　18, 92
EXIT_SUCCESS　18, 92
exp() 関数　93

F

fabs() 関数　94
fclose() 関数　95
fdopen() 関数　96
feof() 関数　99
ferror() 関数　100
fflush() 関数　101
fgetc() 関数　102
fgetpos() 関数　104
fgets() 関数　105
fgetwc() 関数　107
fgetws() 関数　109
FILE 型　18
fileno() 関数　111
float.h インクルード・ファイル　7
floor() 関数　112
fmod() 関数　113
fopen、最大同時ファイル　17
fopen() 関数　114
fpos_t　18
fprintf() 関数　122
fputc() 関数　124
fputs() 関数　127
fputwc() 関数　128
fputws() 関数　130
fread() 関数　132
free() 関数　134
freopen() 関数　136
frexp() 関数　137
fscanf() 関数　138
fseeko() 関数　140
fseek() 関数　140
fsetpos() 関数　142
ftell() 関数　144
fwide() 関数　146
fwprintf() 関数　149
fwrite() 関数　152
fwscanf() 関数　153

G

gamma() 関数　156
getchar() 関数　158
getc() 関数　158
getenv() 関数　160
gets() 関数　162
getwchar() 関数　165
getwc() 関数　163
gmtime64() 関数　169
gmtime64_r() 関数　173
gmtime() 関数　167
gmtime_r() 関数　171

H

HUGE_VAL　9
hypot() 関数　175

I

idate
　関数　25
　現地時間の訂正　192, 194, 195, 197
inttypes.h インクルード・ファイル　7
isalnum() 関数　176
isalpha() 関数　176
isascii() 関数　177
isblank() 関数　179
iscntrl() 関数　176
isdigit() 関数　176
isgraph() 関数　176
islower() 関数　176
isprint() 関数　176
ispunct() 関数　176

isspace() 関数　176
isupper() 関数　176
iswalnu() 関数　180
iswcntrl() 関数　180
iswctype() 関数　182
iswdigit() 関数　180
iswgraph() 関数　180
iswlower() 関数　180
iswprint() 関数　180
iswpunct() 関数　180
iswspace() 関数　180
iswupper() 関数　180
iswxdigit() 関数　180
isxdigit() 関数　176

L

labs() 関数　184
langinfo.h インクルード・ファイル　8
ldexp() 関数　185
ldiv() 関数　186
limits.h インクルード・ファイル　8
llabs() サブルーチン
　絶対値、long long 型整数の　184
lldiv() サブルーチン
　実行、long long 型整数の除算　186
localeconv() 関数　187
locale.h インクルード・ファイル　8
localtime64() 関数　194
localtime64_r() 関数　197
localtime() 関数　192
localtime_r() 関数　195
log10() 関数　199
log() 関数　198
longjmp() 関数　201
lrecl　117

M

malloc() 関数　203
math.h インクルード・ファイル　9
mblen() 関数　205
mbrlen() 関数　207
mbrtowc() 関数　210
mbsinit() 関数　213
mbsrtowcs() 関数　214
mbstowcs() 関数　216
mbtowc() 関数　220
MB_CUR_MAX　18
memchr() 関数　221
memcmp() 関数　222
memcpy() 関数　223
memicmp() 関数　224
memmove() 関数　226
memset() 関数　227
mktime64() 関数　230
mktime() 関数　228
modf() 関数　232
monetary.h インクルード・ファイル　9

N

NDEBUG　3, 46
nextafterl() 関数　232
nextafter() 関数　232
nexttowardl() 関数　232
nexttoward() 関数　232
nltypes.h インクルード・ファイル　9
nl_langinfo() 関数　234
NULL ポインター　15, 17, 18

O

offsetof マクロ　15

P

perror() 関数　236
pointer.h インクルード・ファイル　10
pow() 関数　237
printf() 関数　238
ptrdiff_t　15
putchar() 関数　249
putc() 関数　249
putenv() 関数　250
puts() 関数　251
putwchar() 関数　254
putwc() 関数　252

Q

qsort() 関数　256
QXXCHGDA() 関数　258
QXXDTOP() 関数　259
QXXDTOZ() 関数　260
QXXITOP() 関数　261
QXXITOZ() 関数　262
QXXPTOD() 関数　262
QXXPTOI() 関数　263
QXXRTVDA() 関数　264
QXXZTOD() 関数　265
QXXZTOI() 関数　266

R

raise() 関数　267
rand() 関数　268
RAND_MAX　18
rand_r() 関数　268
realloc() 関数　276
recfm　117
recio.h インクルード・ファイル　10
regcomp() 関数　279
regerror() 関数　281
regexec() 関数　283
regex.h インクルード・ファイル　13
regfree() 関数　285
remove() 関数　286
rename() 関数　287
rewind() 関数　288

S

scanf() 関数　344
seed　370
setbuf() 関数　351
setjmp() 関数　352
setjmp.h インクルード・ファイル　14
setlocale() 関数　354
setvbuf() 関数　360
signal() 関数　362
signal.h インクルード・ファイル　15
sinh() 関数　366
sin() 関数　365
size_t　15
snprintf() 関数　367
sprintf() 関数　368
sqrt() 関数　369
srand() 関数　370
sscanf() 関数　371
stdarg.h インクルード・ファイル　15
stddef.h インクルード・ファイル　15
stdint.h インクルード・ファイル　15
stdio.h インクルード・ファイル　17
stdlib.h インクルード・ファイル　18
strcasecmp() 関数　373
strcat() 関数　374
strchr() 関数　375
strcmpi() 関数　378
strcmp() 関数　376
strcoll() 関数　379
strcpy() 関数　380
strcspn() 関数　381
strdup() 関数　383
strerror() 関数　384
strfmon() 関数　384
strftime() 関数　387
stricmp() 関数　390
string.h インクルード・ファイル　19
strlen() 関数　392
strncasecmp() 関数　393
strncat() 関数　394
strncmp() 関数　395
strncpy() 関数　397
strnicmp() 関数　398

strnset() 関数 400
strpbrk() 関数 401
strptime() 関数 402
strrchr() 関数 406
strset() 関数 400
strspn() 関数 407
strstr() 関数 408
strtod128() 関数 412
strtod32() 関数 412
strtod64() 関数 412
strtod() 関数 409
strtok() 関数 415
strtok_r() 関数 417
strtoll() サブルーチン
 文字ストリングから long long 型整数へ 418
strtol() 関数 418
strtoull() サブルーチン
 文字ストリングから 符号なし long long 型整数へ 420
strtoul() 関数 420
strxfrm() 関数 422
swprintf() 関数 423
swscanf() 関数 425
system() 関数 426

T

tanh() 関数 428
tan() 関数 427
time64() 関数 430
time() 関数 429
time.h インクルード・ファイル 19
tm 構造体 167, 169, 171, 173
tmpfile() 関数
 数 17
 名前 17
tmpnam() 関数
 ファイル名 17
 tmpnam() 433
TMP_MAX 433
toascii() 関数 433
tolower() 関数 434
toupper() 関数 434
towctrans() 関数 435
towlower() 関数 436
towupper() 関数 436

U

ungetc() 関数 439
ungetwc() 関数 440

V

va_arg() 関数 442
va_end() 関数 442
va_start() 関数 442
vfprintf() 関数 444
vfscanf() 関数 446
vfwprintf() 関数 447
vfwscanf() 関数 449
vprintf() 関数 452
vscanf() 関数 453
vsnprintf() 関数 455
vsprintf() 関数 456
vsscanf() 関数 458
vswprintf() 関数 459
vswscanf() 関数 461
vwprintf() 関数 463
vwscanf() 関数 465

W

wchar.h インクルード・ファイル 20
wcrtomb() 関数 467
wcscat() 関数 471
wcschr() 関数 472
wcscmp() 関数 474
wcscoll() 関数 475
wcscpy() 関数 476
wcscspn() 関数 477
wcsftime() 関数 479
wcslen() 関数 482
wcslocaleconv() 関数 483
wcsncat() 関数 484
wcsncmp() 関数 485
wcsncpy() 関数 487
wcspbrk() 関数 490
wcsptime() 関数 491
wcsrchr() 関数 493
wcsrtombs() 関数 494
wcsspn() 関数 496
wcsstr() 関数 497
wcstod128() 関数 499
wcstod32() 関数 499
wcstod64() 関数 499
wcstod() 関数 498
wcstok() 関数 502
wcstoll() サブルーチン
 ワイド文字から long long 型整数へ 503
wcstol() 関数 503
wcstombs() 関数 505
wcstoull() サブルーチン
 ワイド文字ストリングから 符号なし long long へ 508
wcstoul() 関数 508
wcswcs() 関数 510
wcswidth() 関数 511
wcsxfrm() 関数 512
wctob() 関数 513
wctomb() 関数 515
wctrans() 関数 516
wctype() 関数 517
wctype.h インクルード・ファイル 20
wcwidth() 関数 520
wmemchr() 関数 521
wmemcmp() 関数 522
wmemcpy() 関数 524
wmemmove() 関数 525
wmemset() 関数 526
wprintf() 関数 527
wscanf() 関数 528

X

xxcvt.h インクルード・ファイル 20
xxdtaa.h インクルード・ファイル 21
xxenv.h インクルード・ファイル 21
xxfdbk.h インクルード・ファイル 21

[特殊文字]

_C_Get_Ssn_Handle() 関数 57
_C_Quickpool_Debug() 関数 69
 高速プール・メモリー・マネージャーのデバッグ 69
_C_Quickpool_Init() 関数 71
 初期化、高速プール・メモリー・マネージャーの 71
_C_Quickpool_Report() 関数 73
 生成、高速プール・メモリー・マネージャー・レポートの 73
_C_TS_malloc 204, 599
_C_TS_malloc64 204, 599
_C_TS_malloc_debug 599
_C_TS_malloc_debug() 関数 82
_C_TS_malloc_info 599
_C_TS_malloc_info() 関数 84
_EXBDY マクロ 6
_fputchar() 関数 126
_gcvt() 関数 157
_GetExcData() 関数 161
_INTRPT_Hndlr_Parms_T 4
_itoa() 関数 183
_ltoa() 関数 200
_Racquire() 関数 269
_Rclose() 関数 270
_Rcommit() 関数 271
_Rdelete() 関数 273
_Rdevatr() 関数 275
_Rfeod() 関数 290
_Rfeov() 関数 291

_Rformat() 関数　292
_Rindara() 関数　294
_Riofbk() 関数　296
_Rlocate() 関数　298
_Ropen() 関数　301
_Ropnfbk() 関数　305
_Rpgmdev() 関数　306
_Rreadd() 関数　308
_Rreadf() 関数　310
_Rreadindv() 関数　312
_Rreadk() 関数　314
_Rreadl() 関数　318
_Rreadnc() 関数　322
_Rreadn() 関数　319
_Rreadp() 関数　324
_Rreads() 関数　326
_Rrelease() 関数　328
_Rrlslck() 関数　329
_Rrollbck() 関数　331
_Rupdate() 関数　332
_Rupfb() 関数　334
_Rwrited() 関数　338
_Rwriterd() 関数　341
_Rwrite() 関数　336
_Rwrread() 関数　342
_ultoa() 関数　438
_VBDY マクロ　6
_wcsicmp() 関数　480
_wcsnicmp() 関数　488
__EXBDY 組み込み　6
__VBDY 組み込み　6

Printed in USA

SC88-4701-01

日本アイ·ビー·エム株式会社

〒103-8510 東京都中央区日本橋箱崎町19-21