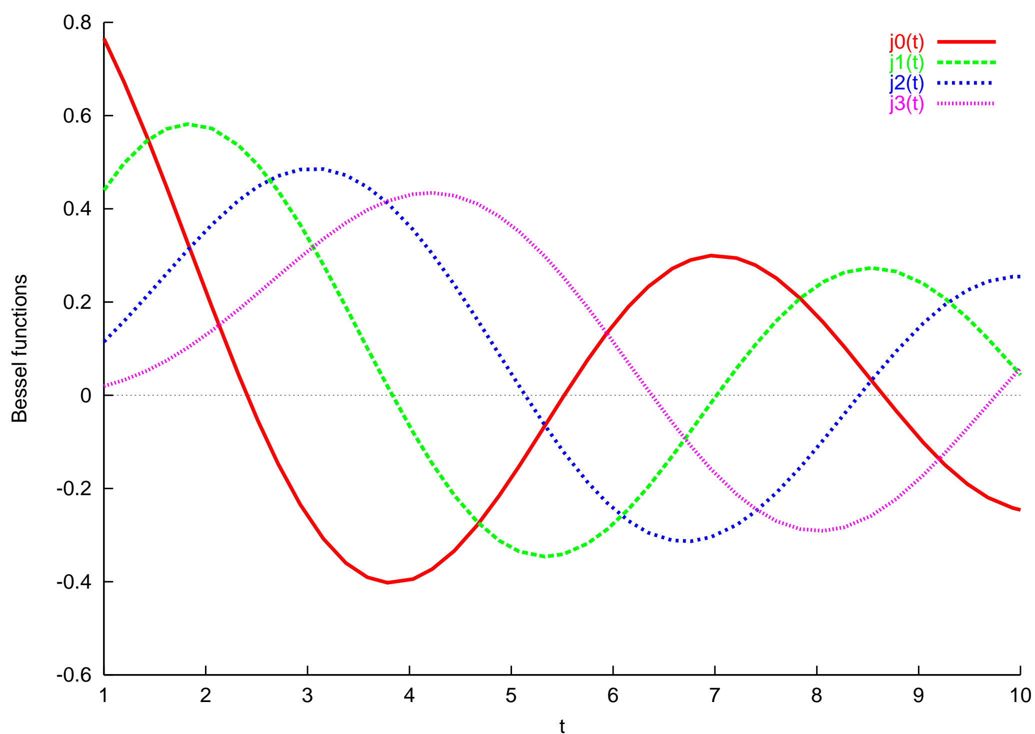
SoftIntegration®

Ch 言語環境

バージョン 7.0

ユーザーズ ガイド



Copyright ©2012 by SoftIntegration, Inc. All rights reserved.
2012 年 9 月 日本語版 7.0

SoftIntegration, Inc. is the holder of the copyright to the Ch language environment described in this document, including without limitation such aspects of the system as its code, structure, sequence, organization, programming language, header files, function and command files, object modules, static and dynamic loaded libraries of object modules, compilation of command and library names, interface with other languages and object modules of static and dynamic libraries. Use of the system unless pursuant to the terms of a license granted by SoftIntegration or as otherwise authorized by law is an infringement of the copyright.

SoftIntegration, Inc. makes no representations, expressed or implied, with respect to this documentation, or the software it describes, including without limitations, any implied warranty merchantability or fitness for a particular purpose, all of which are expressly disclaimed. Users should be aware that included in the terms and conditions under which SoftIntegration is willing to license the Ch language environment as a provision that SoftIntegration, and their distribution licensees, distributors and dealers shall in no event be liable for any indirect, incidental or consequential damages in connection with, or arising out of, the furnishing, performance, or use of the Ch language environment, and that liability for direct damages shall be limited to the amount of purchase price paid for the Ch language environment.

In addition to the foregoing, users should recognize that all complex software systems and their documentation contain errors and omissions. SoftIntegration shall not be responsible under any circumstances for providing information on or corrections to errors and omissions discovered at any time in this documentation or the software it describes, even if SoftIntegration has been advised of the errors or omissions. The Ch language environment is not designed or licensed for use in the on-line control of aircraft, air traffic, or navigation or aircraft communications; or for use in the design, construction, operation or maintenance of any nuclear facility.

Ch、SoftIntegration、および One Language for All は、米国または他の国々における SoftIntegration, Inc. の登録商標または商標です。Microsoft、MS-DOS、Windows、Windows 2000、Windows XP、Windows Vista および Windows 7 は Microsoft Corporation の商標です。

Solaris および Sun は、Sun Microsystems, Inc. の商標です。Unix は、Open Group の商標です。HP-UX は、Hewlett-Packard Co. の登録商標または商標です。Linux は、Linus Torvalds の商標です。Mac OS X と Darwin は、Apple Computers, Inc. の商標です。QNX は、QNX Software Systems の商標です。AIX は、IBM の商標です。本書に記載されている他のすべての名称は、各社の商標です。
本書は、株式会社ラネクシーが、米国 SoftIntegration, Inc. の許可を得て作成した日本語ドキュメントです。

本製品または本製品の派生物の配布は、事前に著作権者の書面による許可を受けない限り、いかなる形式であっても禁止されています。

まえがき

Ch (C H と発音) は、組み込み可能な C/C++インタープリタです。Ch 言語環境は、アプリケーションの設計、およびメカトロニクスと計算運動学における調査のための C 言語でのコンピュータ プログラミングの入門教育を行う目的で、元々は Dr. Harry H. Cheng によって設計、実装されました。ユーザー層が増大するにつれ、Ch は特殊なアプリケーションプログラムから、広範囲に応用可能な一般的な目的をもった言語環境へと進化しました。自然言語を使う場合と同様に、Ch は、*One Language for All* **（あらゆる目的に対する一つの言語）**プログラミングを使用するためのものです。

C 言語は、そのコンパクトな文法、表現力、および幅広い有用性のために選択されてきましたが、多くのアプリケーションにとっては、C インタープリタがさらに望まれています。その結果、C インタープリタが数十年にわたって開発されてきました。この C インタープリタと既存の C コンパイラとを区別するため、これを Ch と呼んでいます。しかしながら、Ch は決して新たな言語というわけではありません。C 標準への準拠は、Ch に対する一般的な設計上のゴールです。私たちは、Ch が現存するもっとも完全な C インタープリタであると信じています。完全な C インタープリタとしての Ch は、1990 年に承認された ISO C90 標準の全機能をサポートしています。Ch は、オブジェクト指向プログラミングのための、C++におけるいくつかの主要な機能をサポートしています。C は元来、システムプログラミング用に設計されました。C は、工学と科学用アプリケーション設計に対しては、多くの欠点があります。Ch は C を、非常に高度な数値計算とグラフィカルのプロット、シェルプログラミングおよび組み込みスクリプト用に拡張したものです。

既存の C を、数値計算に対して拡張するという作業は、C 標準における改訂での ANSI C Standard Committee の取り組みと重なり合っています。私たちが、1993 年以来 ANSI X3J11 および ISO S22/WG14 C Standard Committees へ参加していることにより、Ch は大きな利益を受けました。複素数、可変長配列 (VLA)、バイナリ定数、および関数名 `__func__` のような、最初に Ch に実装された多くの新機能が、C99 と呼ばれる最新の C 標準に追加されてきました。この現在の実装において、Ch は ISO C99 に追加された、ほとんどの既存 C コンパイラよりさらに新しい機能をサポートしています。C プログラマは、本書で記載されている複素数、可変長配列 (VLA)、IEEE 浮動小数点演算、type generic な数学関数のような新規機能を使うよう勇気付けられています。なぜならば、これにより、多くのプログラミングの仕事がかなり簡略化されるからです。C99 におけるこれらの数値機能に加えて、Ch は、Fortran 90 および MATLAB における、線形代数と行列計算のためのファーストクラスオブジェクトである計算配列もサポートしています。

C++でのクラスを C99 標準に追加するような提案が C Standard Committee に提出されました。しかし、時間の制約などの理由によりこの提案は C99 には採用されませんでした。それにも関わらず、主としてこの提案に基づく C++におけるクラスを、Ch は追加しました。これに加えて Ch は、C++における参照を、（Fortran のような）関数への参照による値渡しに便宜をはかるために、サポートしています。

他の多くのソフトウェア パッケージと異なり、Ch は、低水準言語と超高水準言語との橋渡しを行っています。C のスーパーセットとして、Ch は、ハードウェア インターフェース用のメモリ アクセス

のような、C の低水準機能を維持しています。とはいえ、Ch は超高水準言語 (VHLL) 環境です。Ch は、string 型をビルトインしたシェル プログラミングをサポートしています。数千行もの C コードが必要な問題は、Ch のほんの数行を用いて容易に解決できます。Ch を使うと、プログラマは生産性を飛躍的に向上させることができます。

さらに、Ch はプラットフォームに依存しないように設計されています。Ch は、異なるコンピュータ ハードウェアと、Windows、Linux、Mac OS X、および Unix を含む オペレーティング システムを用いた異質のコンピューティング環境上で動作することができます。一つのプラットフォーム上で開発されたプログラムは、他のいかなるプラットフォーム上でも動作可能です。

このドキュメントは、C99 における新機能を用いた C/Ch 上でのプログラミング方法を習得したい読者のために記述されています。プログラミング言語の知識が前もって必要とされないとはいえ、Ch の基礎を理解することは有益です。このドキュメントでは、C90 に対する C99 の拡張が強調されています。C に対する Ch の拡張に関しては、特に強調されています。このドキュメントで説明されていない機能については、ISO C99 Standard の解釈に従います。サンプル プログラムは、Ch の開発中に使用された多くのテストコードを含んでいます。コンピュータの初心者ユーザーであろうと、経験のあるプロフェッショナルプログラマであろうと、Ch を使用すると、プログラミング作業がさらに楽しくなり、そして Ch が気に入ってくれることを期待します。

Ch と C/C++の違いにあまり関心がない場合は、付録 B および C を読みとばしても構いません。これらの個所では、C/C++に対する Ch の新機能の概要について記載されています。伝統的なコンパイル、リンク、実行、デバッグのサイクルなしに、Ch ですばやくプログラミングを開始することができます。また Ch では、`hello.c` または `hello.cpp` のように、プログラム名を入力するだけで、Ch コマンド シェル上で C/C++プログラムを実行することができます。さらに Ch では、統合化開発環境 (IDE) から C/C++を実行することも可能です。

表記規則

以下のリストは、このドキュメント全体にわたる、テキストの特定の要素に対する表示形式として使用されている、表記規則について定義し、説明しています。

- インターフェース コンポーネント は、モニタ画面または表示上に現れるウィンドウ タイトル、ボタン、アイコン名、メニュー名、選択項目、およびその他のオプションです。これらは、太字で表されています。マウスでの一連のポイントとクリックは、一連の太字の単語で表されています。

  例: **OK** をクリックします

  例: シーケンス **[スタート]**–>**[プログラム]**–>**[Ch7.0]**–>**[Ch]** は、最初に **[スタート]** を選択し、次にマウスで **[プログラム]** をポイントして、**[プログラム]** サブメニューを選択して、**[Ch7.0]** を選び、最後に **[Ch]** を選択することを示しています。

- キーボードのキー上にあるラベル、キーキャップ は、山かっこで囲まれています。キーキャップのラベルは、タイプライターのような書体で表されています。

  例: `<Enter>` を押します

- キー組み合わせ とは、(特に指定がない場合) 一つの機能を実行するために、一連のキーが同時に押されることです。キーのラベルは、タイプライターのような書体で表されています。

例: <Ctrl><Alt><Enter> を押します

- コマンド　は、小文字の太字で表されていて、参照のためのみにあり、議論の特定のポイントで意図して入力するものではないことを示しています。

  例: “... をインストールするには、**[インストール]** コマンドを使用します。”

  これに対して、タイプライターのような書体で表されたコマンドは、命令の一部として意図的に入力されます。

  例: “ソフトウェアを現在のディレクトリ上にインストールするには、`[install]` を入力します。”

- コマンド書式行　は、コマンドとそのすべてのパラメータから構成されます。コマンドは小文字の太字で表示され、（ある値で置き換える）変数パラメータは、小文字のイタリック体で表示され、定数パラメータは小文字の太字で表示されます。かっこは、オプションの項目を示しています。

  例: **ls** [**-aAbcCdfFgilLmnopqrRstux1**] [*file* ...]

- コマンド行　は、コマンドと場合によっては、いくつかのコマンド パラメータで構成されます。コマンド行は、タイプライターのような書体で表されます。

  例: `ls /home/username`

- 画面テキスト　は、画面または外部モニタ上に表示されます。これはたとえば、システム メッセージ、または（コマンド行のように参照される）コマンドの一部として入力する可能性があるテキストです。画面テキストは、タイプライターのような書体で表されます。

  例: 以下のメッセージが画面上に表示されます

  ```
  usage:  rm [-fiRr] file ...


  ls [-aAbcCdfFgilLmnopqrRstux1] [file ... ]
  ```

- 関数プロトタイプ　は、返値、関数名、およびデータ型とパラメータをもった引数で構成されます。Ch 言語のキーワード、typedef 名、および関数名は太字体で表されます。関数引数のパラメータは、イタリック体で表されます。かっこは、オプション項目を示しています。

  例: **double derivative**(**double** (**func*)(**double**), **double** *x*, ... [**double** **err*, **double** *h*]);

- プログラムの ソース コード　は、タイプライターのような書体で表されます。

  例: 以下のコードのプログラム **hello.ch**

  ```
  int main() {
      printf("Hello, world!\n");
  }
  ```

  は、画面上に、 `Hello, world!` を表示します。

- 変数 は、ある値で置き換えるべきシンボルです。これらはイタリック体で表されます。

  例: module *n* (ここで、*n* はメモリ モジュール番号を表します)

- システム変数とシステム ファイル名 は、太字体で表されます。

  例: Unix 上のスタートアップ ファイル **/home/username/.chrc** または `/home/username` ディレクトリの **.chrc** および、Windows 上の **C:_chrc** または C:\ ディレクトリの **_chrc**

- プログラム内で宣言された 識別子 は、テキスト内で使用される場合、タイプライターのような書体で表されます。

  例: 変数 `var` がプログラム内で宣言されます。

- ディレクトリ は、テキスト内で使用される場合、タイプライターのような書体で表されます。

  例: Ch は、Unix ではディレクトリ `/usr/local/ch` に、Windows ではディレクトリ `C:\Ch` にインストールされます。

- 環境変数 は、システム レベルの変数です。これらは太字体で表されます。

  例: 環境変数 **PATH** はディレクトリ /usr/ch を含みます。

関連ドキュメント

Ch のマニュアル構成は、以下のとおりとなっています。これらのマニュアル（PDF 形式）は、製品 CD に同梱され、CHHOME/docs （ここで CHHOME は Ch のホーム ディレクトリ）にインストールされます。

- *The Ch Language Environment — Installation and System Administration Guide*, version 7.0, SoftIntegration, Inc., 2011.

  このマニュアルは、ウェブサーバ用の Ch のスタートアップ方法だけでなく、システムインストールと構築について述べています。
  （注）このマニュアルは日本語化されていません。製品のインストール方法（Windows 版）については、製品パッケージ同梱の『補足説明書』を参照してください。

- ***Ch* 言語環境、— ユーザーズ ガイド**, version 7.0, SoftIntegration, Inc., 2011.

  このマニュアル（本書です）は、様ざまなアプリケーションに対する Ch の言語機能について記載しています。

- ***Ch* 言語環境、— リファレンス ガイド**, version 7.0, SoftIntegration, Inc., 2011

  このマニュアルは、サンプル ソースコードを伴った、関数、クラスおよびコマンドの詳細なリファレンス情報を提供しています。

- *The Ch Language Environment, — SDK User's Guide*, version 7.0, SoftIntegration, Inc., 2011.

  このマニュアルは、静的または動的ライブラリにおいて C/C++関数とのインターフェース用のソフトウェア開発キットについてのべています。
  （注）このマニュアルは日本語化されていません。

- *The Ch Language Environment CGI Toolkit User's Guide*, version 3.5, SoftIntegration, Inc., 2003.

  このマニュアルは、CGI クラスにおける Common Gateway Interface と、そのクラスの各メンバ関数に対する詳細なリファレンス情報を提供します。
  （注）このマニュアルは日本語化されていません。

目次

まえがき **2**

Ch グラフィック ギャラリー **20**

序論 **22**

第 I 部　言語の機能 **27**

第 1 章　はじめに **28**

1.1 起動 28

1.1.1 Unix での起動 28

1.1.2 Windows での起動 30

1.2 コマンドモード 30

1.3 プログラムモード 33

1.3.1 コマンドファイル 33

1.3.2 スクリプトファイル 34

1.3.3 関数ファイル 35

1.4 複素数 35

1.5 計算配列 37

1.6 プロット 39

第 2 章　字句要素 **41**

2.1 文字セット 41

2.1.1 三連文字 41

2.2 キーワード 42

2.2.1 キーワード 42

2.2.2 予約済みのシンボル 44

2.3 識別子 44

2.3.1 事前定義済みの識別子 44

2.4 区切り子 50

2.5 コメント 50

第 3 章　プログラムの構造 **52**

3.1 Ch ホームディレクトリ内のディレクトリとファイル 52

3.2 起動 52

3.2.1 スタートアップファイルのサンプル 54

3.2.2 コマンドラインオプション . . . 58
3.3 Ch プログラム . . . 59
3.3.1 コマンドファイル . . . 59
3.3.2 スクリプトファイル . . . 61
3.3.3 関数ファイル . . . 61
3.4 プログラムの実行 . . . 64
3.4.1 コマンドモードでのプログラミングステートメントの実行 . . . 64
3.4.2 プログラムの起動 . . . 65
3.4.3 プログラムの終了 . . . 66
3.4.4 検索順序 . . . 66
3.4.5 複数のファイルを使用するプログラムの実行 . . . 67
3.4.6 プログラムのデバッグ . . . 70
3.5 スコープの規則 . . . 72
3.5.1 識別子のスコープ . . . 72
3.5.2 識別子のリンケージ . . . 73
3.5.3 識別子の名前空間 . . . 74
3.5.4 オブジェクトの記憶期間 . . . 74

第 4 章 移植可能な対話型コマンドシェルとシェルプログラミング 76
4.1 シェルプロンプト . . . 76
4.2 コマンドの対話的な実行 . . . 77
4.2.1 現在のシェル . . . 78
4.2.2 バックグラウンドジョブ . . . 79
4.3 プログラムステートメントの対話的な実行 . . . 80
4.4 組み込みコマンド . . . 83
4.4.1 対話型シェル専用のコマンド . . . 84
4.5 プロンプトでのコマンドの繰り返し . . . 86
4.5.1 履歴置換 . . . 86
4.5.2 簡易置換 . . . 89
4.5.3 ファイルの補完 . . . 90
4.5.4 コマンドの補完 . . . 91
4.6 別名 . . . 92
4.7 変数置換 . . . 96
4.7.1 式置換 . . . 97
4.7.2 コマンド名置換 . . . 98
4.8 ファイル名置換 . . . 99
4.9 コマンド置換 . . . 101
4.10 入出力のリダイレクト . . . 103
4.11 パイプライン . . . 106
4.12 バックグラウンドでのコマンドの実行 . . . 108
4.13 実行時の式の評価 . . . 108
4.14 環境変数の処理 . . . 109

4.15 汎用 Ch プログラム . . . 112
4.16 シェルプログラミング . . . 114
4.16.1 プログラムでのシェルコマンドの使用 . . . 114
4.16.2 シェルコマンドへの値渡し . . . 118

第 5 章 プリプロセッサディレクティブ 122
5.1 条件付き組み込み . . . 122
5.2 ソースファイルの組み込み . . . 123
5.3 マクロ置換 . . . 124
5.4 トークンから文字列への変換 . . . 126
5.5 マクロ拡張で結合するトークン . . . 127
5.6 行の制御 . . . 128
5.7 エラーディレクティブ . . . 129
5.8 NULL ディレクティブ . . . 129
5.9 プラグマディレクティブ . . . 129
5.10 事前定義済みのマクロ . . . 132

第 6 章 型と宣言 134
6.1 データ型 . . . 134
6.1.1 整数型のデータ . . . 134
6.1.2 浮動小数点型 . . . 137
6.1.3 集合体の浮動小数点型 . . . 140
6.1.4 ポインタのデータ型 . . . 141
6.1.5 配列型 . . . 141
6.1.6 構造体型 . . . 143
6.1.7 クラス型 . . . 144
6.1.8 ビットフィールド . . . 145
6.1.9 共用体型 . . . 145
6.1.10 列挙型 . . . 146
6.1.11 Void 型 . . . 146
6.1.12 参照型 . . . 146
6.1.13 文字列型 . . . 147
6.1.14 関数型 . . . 149
6.2 型修飾子 . . . 150
6.2.1 計算配列 . . . 151
6.2.2 制限付き関数 . . . 151
6.3 定数 . . . 151
6.3.1 文字定数 . . . 151
6.3.2 文字列リテラル . . . 154
6.3.3 整数定数 . . . 155
6.3.4 浮動小数点定数 . . . 157
6.4 初期化 . . . 158

第7章　演算子と式 **162**
7.1 算術演算子 165
7.2 関係演算子 165
7.3 論理演算子 168
7.4 ビットごとの演算子 168
7.5 代入演算子 169
7.6 条件演算子 169
7.7 キャスト演算子 172
7.7.1 キャスト演算子 172
7.7.2 関数型キャスト演算子 172
7.8 コンマ演算子 173
7.9 単項演算子 174
7.9.1 アドレスと間接演算子 174
7.9.2 増分演算子と減分演算子 175
7.9.3 コマンド置換演算子 176
7.10 メンバ演算子 176

第8章　ステートメントと制御フロー **178**
8.1 単純ステートメントと複合ステートメント 178
8.2 式ステートメントと空ステートメント 178
8.3 選択ステートメント 179
8.3.1 If ステートメント 179
8.3.2 If-Else ステートメント 180
8.3.3 Else-If ステートメント 180
8.3.4 Switch ステートメント 181
8.4 繰り返しステートメント 182
8.4.1 While ループ 182
8.4.2 Do-While ループ 183
8.4.3 For ループ 184
8.4.4 Foreach ループ 185
8.5 分岐ステートメント 186
8.5.1 Break ステートメント 186
8.5.2 Continue ステートメント 186
8.5.3 Return ステートメント 187
8.5.4 Goto ステートメント 187
8.6 ラベル付きステートメント 189

第9章　ポインタ **190**
9.1 ポインタ演算 190
9.2 メモリの動的な割り当て 193
9.3 ポインタ配列 195
9.4 ポインタへのポインタ 197

第 10 章 関数 200
10.1 値呼び出しと参照呼び出し . . . 201
10.2 関数の定義 . . . 201
10.3 関数プロトタイプ . . . 205
10.4 再帰関数 . . . 210
10.5 入れ子にされた関数 . . . 211
10.5.1 入れ子された関数のスコープとレキシカルレベル . . . 212
10.5.2 入れ子にされた関数のプロトタイプ . . . 215
10.5.3 入れ子にされた再帰関数 . . . 220
10.6 ポインタを使用した、関数の引数の参照渡し . . . 224
10.7 関数内の可変長の引数 . . . 225
10.8 関数へのポインタ . . . 234
10.9 関数間の通信 . . . 242
10.10 `main()` 関数とコマンドライン引数 . . . 243
10.11 関数ファイル . . . 247
10.12 汎用関数 . . . 249

第 11 章 参照型 250
11.1 ステートメント内の参照 . . . 251
11.2 関数の引数の参照渡し . . . 253
11.3 データ型の異なる変数を同じ参照に渡す方法 . . . 258

第 12 章 汎用数学関数を使用する科学計算 261
12.1 汎用数学関数の概要 . . . 262
12.2 プログラミング例 . . . 266
12.2.1 浮動小数点数の極値の計算 . . . 266
12.2.2 メタ数値を使用したプログラミング . . . 270

第 13 章 複素数を使用したプログラミング 272
13.1 複素数 . . . 272
13.1.1 複素定数と複素変数 . . . 272
13.2 複素平面と複素メタ数値 . . . 273
13.2.1 データ変換規則 . . . 275
13.3 複素数に対する入出力 . . . 278
13.4 複素演算 . . . 279
13.4.1 通常の複素数による複素演算 . . . 279
13.4.2 複素メタ数値による複素演算 . . . 279
13.5 複素関数 . . . 282
13.5.1 通常の複素数を使用する複素関数の結果 . . . 282
13.5.2 複素メタ数値を使用する複素関数の結果 . . . 286
13.6 複素数に関する lvalue . . . 290
13.7 ユーザーによる複素関数の作成 . . . 291

第 14 章 ポインタと配列 **293**
14.1 ポインタを使用した配列要素へのアクセス 293
14.2 配列の動的割り当て 294
14.2.1 1 次元配列の動的割り当て 294
14.2.2 2 次元配列の動的割り当て 295
14.3 固定長の 1 次元および多次元配列を渡す処理 298
14.3.1 1 次元配列 298
14.3.2 固定長の多次元配列 300

第 15 章 可変長配列 **304**
15.1 記憶期間と配列の宣言 305
15.1.1 オブジェクトの記憶期間 305
15.1.2 配列の宣言 306
15.2 形状無指定配列 308
15.2.1 制約と意味 308
15.2.2 switch ステートメントに関連する形状無指定配列 311
15.2.3 goto ステートメントに関連する形状無指定配列 311
15.2.4 構造体と共用体のメンバとしての形状無指定配列 313
15.2.5 Sizeof 315
15.2.6 Typedef 316
15.2.7 その他のデータ型とポインタ演算 317
15.3 形状引継ぎ配列 317
15.3.1 制約と意味 317
15.3.2 Sizeof 321
15.3.3 その他のデータ型とポインタ演算 321
15.4 形状引継ぎ配列へのポインタ 322
15.4.1 宣言 322
15.4.2 制約と意味 323
15.4.3 関数プロトタイプのスコープ 325
15.4.4 Typedef 326
15.4.5 メモリ割り当て関数による配列の動的割り当て 326
15.4.6 固定長配列へのポインタと形状引継ぎ配列へのポインタとの類似点 327
15.5 上限値と下限値を明示した配列 330
15.5.1 固定添字範囲の配列 330
15.5.2 可変添字範囲の配列 334
15.6 上限値と下限値を明示した配列を関数に渡す方法 336
15.6.1 固定添字範囲の配列を渡す方法 337
15.6.2 形状引継ぎ配列へのポインタを使用して可変長添字範囲の配列を渡す方法 . . . 339

第 16 章 計算配列と行列計算 **343**
16.1 計算配列の宣言と初期化 344
16.2 配列参照 345
16.2.1 配列全体 345

16.2.2 配列の要素 347
16.3 計算配列の代入形式と出力 348
16.4 計算配列の明示的データ型変換 350
16.5 配列演算 351
16.5.1 算術演算 351
16.5.2 代入演算 354
16.5.3 インクリメント演算とデクリメント演算 354
16.5.4 関係演算 355
16.5.5 論理演算子 357
16.5.6 条件演算 358
16.5.7 アドレス演算子 358
16.5.8 キャスト演算子 359
16.6 演算でのスカラから計算配列への上位変換 360
16.7 計算配列を関数に渡す方法 361
16.7.1 形状が完全指定された配列 362
16.7.2 形状引継ぎ配列 363
16.7.3 形状無指定配列 366
16.7.4 可変長の引数の配列 367
16.7.5 参照配列 369
16.8 値 NULL の計算配列 376
16.9 計算配列を返す関数 378
16.9.1 固定長の計算配列を返す関数 378
16.9.2 可変長の計算配列を返す関数 379
16.10 型の汎用配列関数 380
16.11 使用頻度の高い配列関数 384
16.12 計算配列へのポインタ 388
16.12.1 固定長の計算配列へのポインタ 388
16.12.2 形状引継ぎ計算配列へのポインタ 394
16.12.3 計算配列へのポインタを使用して配列を関数に渡す方法 396
16.13 計算配列と C 配列との関係 397

第 17 章 文字と文字列 399

17.1 string.h ヘッダーファイル内の関数の使用 399
17.1.1 コピー関数 400
17.1.2 連結関数 401
17.1.3 比較関数 401
17.1.4 検索関数 402
17.1.5 その他の関数 403
17.1.6 C 標準ライブラリでサポートされていない Ch の文字列関数 404
17.2 文字列型 string_t 404
17.3 foreach ループを使用した文字列トークンの処理 408
17.4 ワイド文字 409

17.5 ワイド文字列 . . . 409

第 18 章 構造体、共用体、ビットフィールド、および列挙体 410

18.1 構造体 . . . 410
18.2 共用体 . . . 411
18.3 ビットフィールド . . . 412
18.4 列挙体 . . . 413

第 19 章 クラスおよびオブジェクトベースのプログラミング 415

19.1 クラス定義とオブジェクト . . . 415
19.2 クラスのメンバ関数 . . . 415
19.3 クラスの public メンバと private メンバ . . . 417
19.4 クラスのコンストラクタとデストラクタ . . . 418
19.5 演算子 new および delete . . . 419
19.6 クラスの静的メンバ . . . 421
19.7 スコープ解決演算子 :: . . . 423
19.8 暗黙のポインタ this . . . 424
19.9 ポリモーフィズム . . . 424
19.9.1 ポリモーフィックな汎用数学関数 . . . 425
19.9.2 参照配列型のパラメータを含む関数 . . . 426
19.9.3 ポリモーフィックな関数 . . . 426
19.9.4 クラスのポリモーフィックなメンバ関数 . . . 433
19.10 入れ子にされたクラス . . . 434
19.11 メンバ関数内のクラス . . . 437
19.12 関数の引数としてのメンバ関数の受け渡し . . . 439
19.13 事前定義済みの識別子 __class__ と __class_func__ . . . 443

第 20 章 入出力 445

20.1 ストリーム . . . 445
20.2 入出力のバッファリングとノンバッファリング . . . 445
20.3 入出力形式 . . . 447
20.3.1 fprintf 出力関数ファミリの出力形式 . . . 447
20.3.2 fscanf 入力関数ファミリの入力形式 . . . 452
20.4 既定の入出力形式 . . . 457
20.4.1 fprintf 出力関数ファミリの既定の形式 . . . 457
20.4.2 fscanf 入力関数ファミリの既定の形式 . . . 458
20.4.3 cout、cin、cerr、および endl を使用する入出力 . . . 459
20.5 メタ数値の入出力 . . . 461
20.6 集合体データ型の入出力形式 . . . 463
20.7 fprintf による逐次出力ブロック . . . 464
20.8 ファイル操作 . . . 468
20.8.1 ファイルを開く/閉じる . . . 468
20.8.2 ファイルの読み込み/書き込み . . . 470

20.8.3 ランダムアクセス 472
20.9 ディレクトリ操作 473
20.9.1 ディレクトリを開く処理と閉じる処理 474
20.9.2 ディレクトリの読み込み 476

第 21 章 セーフ Ch **480**
21.1 セーフ Ch シェル 480
21.1.1 Windows での起動 480
21.2 サンドボックス内の無効な機能 480
21.3 制限付き関数 483
21.4 セーフ Ch プログラム 483
21.5 アプレットとネットワークコンピューティング 483

第 22 章 ライブラリ、ツールキット、およびパッケージ **484**
22.1 ライブラリ 484
22.2 ツールキット 488
22.3 パッケージ 490

第 II 部 科学計算用ライブラリ **494**

第 23 章 2 次元プロットと 3 次元プロット **495**
23.1 プロットのクラス 495
23.1.1 プロットのためのデータ 495
23.1.2 注釈 504
23.1.3 複数のデータセットと凡例 510
23.1.4 事前定義済みの幾何プリミティブ 517
23.1.5 サブプロット 518
23.1.6 プロットのエクスポートとズーム 521
23.1.7 プロットの出力 525
23.2 2 次元プロット 525
23.2.1 プロットの種類、線のスタイル、マーカー 525
23.2.2 極座標 534
23.2.3 2 次元プロット関数 536
23.3 3 次元プロット 539
23.3.1 プロットの種類 539
23.3.2 各種の座標系を使用したプロット 540
23.3.3 3 次元プロット関数 543
23.4 動的な Web プロット 545

第 24 章 数値解析 **550**
24.1 数学関数 551
24.1.1 外積 551

24.1.2 内積 . . . 555
24.1.3 一様乱数 . . . 555
24.1.4 符号関数 . . . 556
24.1.5 最大公約数 . . . 556
24.1.6 最小公倍数 . . . 557
24.1.7 複素方程式 . . . 557
24.2 データ解析と統計 . . . 558
24.2.1 コンソールからの数値の取得 . . . 558
24.2.2 配列へのデータの代入 . . . 559
24.2.3 最小値と最大値 . . . 560
24.2.4 合計 . . . 560
24.2.5 積 . . . 562
24.2.6 平均値 . . . 563
24.2.7 中央値 . . . 563
24.2.8 標準偏差 . . . 564
24.2.9 共分散と相関係数 . . . 564
24.2.10 ノルム . . . 566
24.2.11 階乗 . . . 567
24.2.12 組み合わせ . . . 568
24.2.13 データの並べ替え . . . 568
24.2.14 アンラップ . . . 569
24.2.15 配列の要素に適用する関数 . . . 571
24.2.16 ヒストグラム . . . 572
24.3 データ補間とカーブフィッティング . . . 572
24.3.1 1 次元補間 . . . 572
24.3.2 2 次元補間 . . . 574
24.3.3 一般的なカーブフィッティング . . . 576
24.3.4 多項式関数を使用するカーブフィッティング . . . 577
24.4 関数の最小化または最大化 . . . 579
24.4.1 1 つの変数を含む関数の最小化 . . . 579
24.4.2 複数の変数を含む関数の最小化 . . . 580
24.5 多項式 . . . 581
24.5.1 多項式の評価 . . . 582
24.5.2 多項式の導関数 . . . 583
24.5.3 代数方程式の根 . . . 585
24.5.4 代数方程式の係数 . . . 585
24.5.5 多項式の因数分解の剰余 . . . 586
24.5.6 行列の特性多項式 . . . 589
24.6 非線形方程式 . . . 589
24.6.1 非線形方程式の解法 . . . 589
24.6.2 非線形連立方程式の解法 . . . 590
24.7 導関数と常微分方程式 . . . 591

24.7.1 差分 591
24.7.2 導関数 591
24.8 常微分方程式の解法 592
24.9 数値積分 595
24.9.1 1 次元積分 595
24.9.2 2 次元積分 595
24.9.3 3 次元積分 597
24.10 行列関数 598
24.10.1 行列の特性 598
24.10.2 行列の操作 601
24.10.3 特殊な行列 602
24.10.4 行列解析 607
24.11 行列分解 609
24.11.1 LU 分解 609
24.11.2 特異値分解 610
24.11.3 Cholesky 分解 612
24.11.4 QR 分解 613
24.11.5 Hessenberg 分解 615
24.11.6 Schur 分解 616
24.12 線形方程式 616
24.12.1 連立一次方程式 616
24.12.2 過剰決定または過少決定の連立一次方程式 617
24.12.3 逆行列と疑似逆行列 619
24.12.4 線形空間 621
24.13 固有値と固有ベクトル 622
24.14 高速 Fourier 変換 623
24.15 畳み込みとフィルタリング 626
24.16 相互相関 635

第 25 章 参考文献 637

付 録 A 既知の問題とプラットフォーム特有の機能 639
A.1 プラットフォーム固有の機能 639
A.1.1 Solaris 639
A.1.2 Windows NT/2000/XP/Vista/Windows 7 639
A.2 特定のプラットフォームでサポートされていない関数 639

付 録 B C の動作と実装で定義される動作の比較 641
B.1 Ch でサポートされている C99 の新機能 641
B.2 C に対する拡張の要約 642
B.3 実装に関する補足 645
B.3.1 無制限のプロパティ 645
B.3.2 定義済みプロパティ 646

B.3.3 一時的な機能 647
B.3.4 Ch と C の非互換性 648
B.4 C を Ch に移植する場合のヒント 650

付 録 C C++との比較 653
C.0.1 C++と Ch の両方にある機能 653
C.0.2 C++のクラスに対する Ch での拡張 654
C.0.3 Ch でサポートされていない C++の機能 654
C.0.4 C++と Ch の相違点 656

付 録 D C シェルとの比較 658
D.1 構文 658
D.2 制御フロー 661

付 録 E MATLAB との比較 662
E.1 演算子 663
E.2 関数および定数 665
E.3 制御フロー 675

付 録 F Fortran との比較 676
F.1 Ch での参照と FORTRAN での equivalence 676
F.2 Ch および FORTRAN での参照呼び出し 677

付 録 G Ch で一般的に使用される移植性のあるシェルコマンドの概要 680
G.1 ファイルシステム 680
G.2 バイナリファイル 681
G.3 テキストファイル 682
G.4 ファイルの比較 683
G.5 シェルのユーティリティ 683
G.6 ファイルのアーカイブ 683

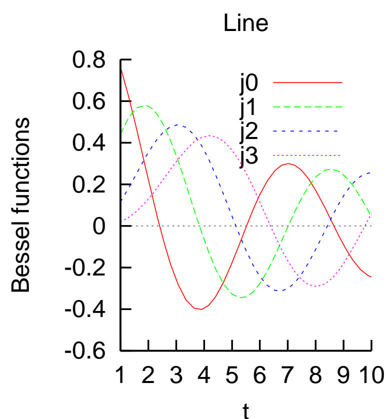
付 録 H vi テキストエディタの概要 684

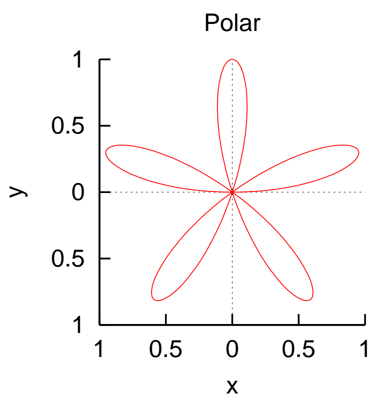
付 録 I 最新バージョンへのコードの移植 687
I.1 Ch バージョン 6.1.0.13631 へのコードの移植 687
I.2 Ch バージョン 6.0.0.13581 へのコードの移植 687

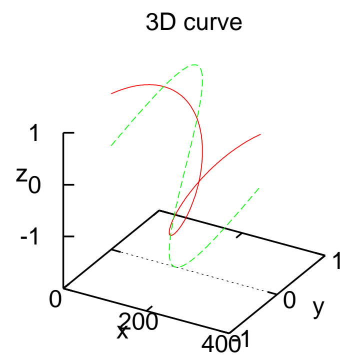
索引 690

Ch グラフィック ギャラリー

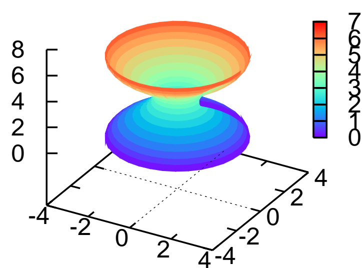
プロット



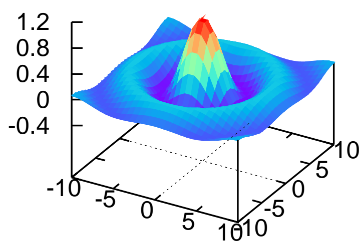




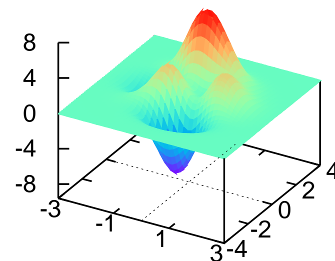
Cylindrical



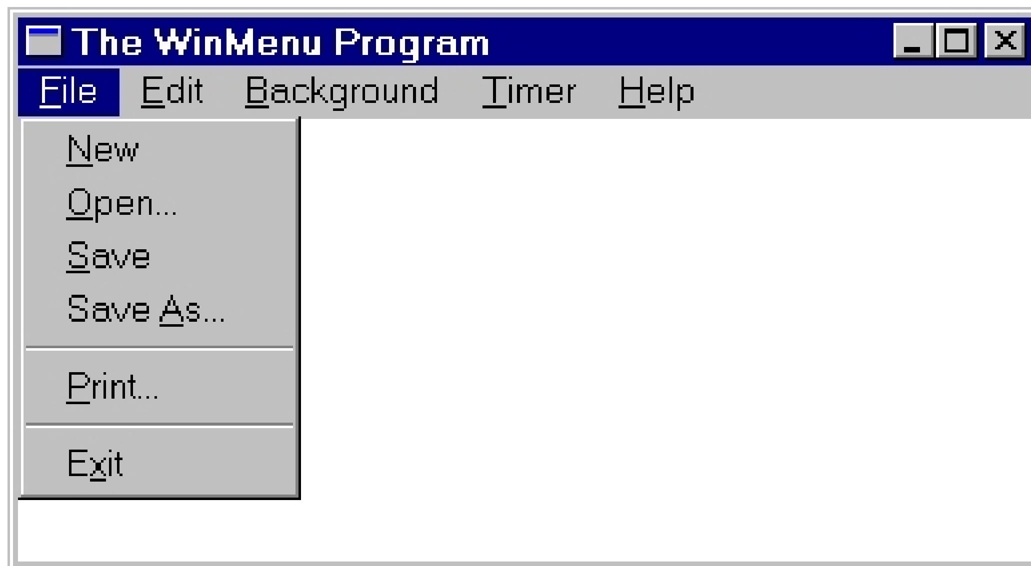
3D Mesh



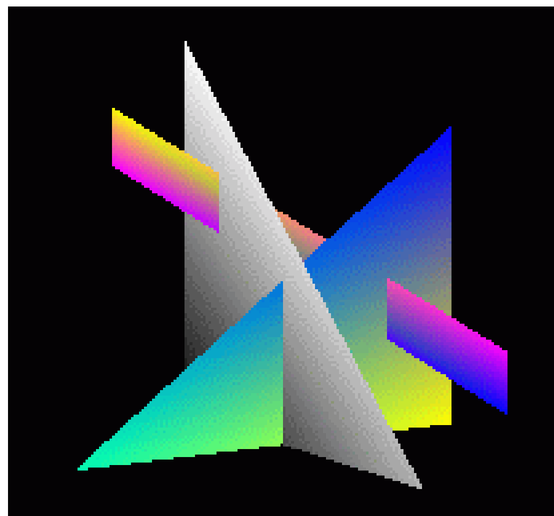
3D Mesh



グラフィカル ユーザー インターフェース



グラフィックとアニメーション



序論

Ch とは?

***Ch* は *C+*。*Ch* は、組み込み可能な *C/C++*インタープリタです。**
Ch は C にインタープリタ機能を実装したものであり、スクリプト、迅速なアプリケーション開発、配布およびレガシーシステムでの統合のために、C++、他の言語、ソフトウェアパッケージからの突出した機能を備えています。Ch は、経験のある C/C++プログラマと初心者の両方を対象に設計されています。C 言語のスキルを利用することによって、プログラマは C を学習し、C 言語をあらゆるプログラミングの目的に使用することができます。

***Ch* は組み込みが可能です。**
C/C++コンパイラとは異なり、Ch は C/C++アプリケーションとハードウェア内にスクリプトエンジンとして組み込み可能です。Ch を用いると、ユーザーは、多くのアプリケーションに対するマクロ言語またはインタープリタを開発および管理する必要がなくなります。

***Ch* は *2D/3D* グラフィカルプロットと数値計算を実行することができます。**
Ch は特に科学技術分野アプリケーション用に設計されています。Ch は内蔵グラフィカル機能をサポートし、線形代数と行列計算、2D/3D グラフィカルプロット、および線形系、微分方程式の解法、積分、非線形方程式、曲線の近似、Fourier 解析などの拡張された高度な数値関数も備えています。たとえば、線形方程式 $\mathbf{b} = A * \mathbf{x}$ は、Ch を用いて正確にそのまま記述可能です。ユーザーは、高速で正確な数値アルゴリズムを用いた、内部的なプログラムの最適化について心配する必要はありません。Ch は、Riemman 平面に対する拡張複素平面を用いた全実数域と複素数域で、IEEE 浮動小数点演算に基づく数値計算が実行可能な、現存する唯一の環境です。C に対する拡張により、Ch は C/C++領域での数値計算に対する理想的な選択肢となっています。

***Ch* は *C* 互換のシェル・プログラミングです**
一般的に *csh* といった C シェルは C のようなシェルですが、Ch は C 互換のシェルです。Ch は非常に高水準な言語 (VHLL) 環境です。Windows 版の Ch は、クロス・プラットフォーム・シェル・プログラミングのために、*vi*、*ls*、*awk*、*sed*、*mv*、*rm*、*cp*、*find*、*grep* などのような頻繁に用いられている Unix ユーティリィティとコマンドをサポートしています。これらは、繰り返し実行されるタスクの自動化に使用することができます。

いくつかの複雑な問題 (1000 行以上にのぼる C のコード) は Ch のコードわずか数行で解決することができます。インタラクティブな Ch コマンドシェルは、迅速なプロトタイピングや、授業および学習に特に適しています。

Ch は他の多くの言語やソフトウェアパッケージから機能やアイディアを取り入れています。Ch はその多くの部分を C/C++に負っています。Ch の開発に何らかの影響を及ぼした他の言語およびソフトウェアパッケージの一覧を以下に示します。

- C シェルのように、Ch はログインシェルやシェルプログラミングに使用することができますが、Ch は C のスーパーセットとしての、真の C シェルであると言えます。

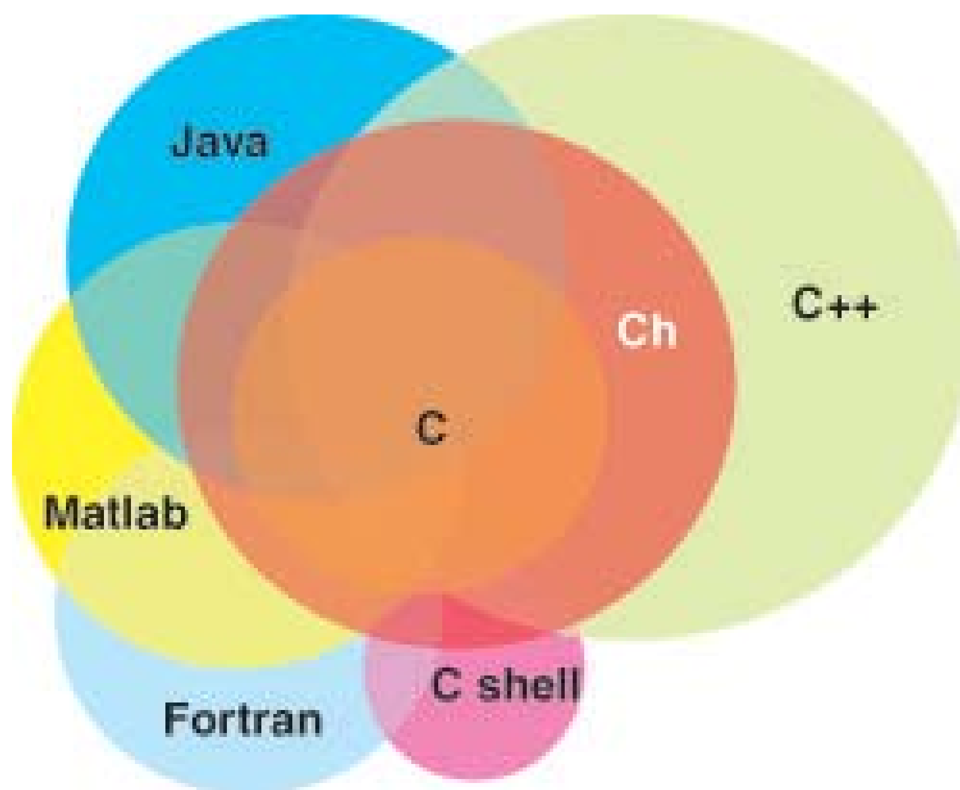


図 1: Ch と他の言語およびソフトウェアパッケージとの関係

- Basic のように、Ch はコンピュータ経験の浅い初心者のために設計されており、使用されてきました。
- Perl のように、Web サーバーの中で Common Gateway Interface(CGI) に使用することができます。
- Java のように、Ch はインターネットコンピューティングに使用することができます。Ch アプレットは、異なるコンピュータプラットフォーム上のネットワークを越えて実行可能です。
- JavaScript のように、Ch スクリプトは Active Server Pages(ASP) のように HTML ファイルへ組み込むことが可能です。
- Fortran 77/90 のように、Ch は科学計算に使用することができます。
- MATLAB/Mathematica のように、迅速なプロトタイピングに使用することができます。

Ch とこれらの言語およびソフトウェアパッケージとの Ch の関係は図 1に示されます。

主な機能

Ch は、ISO 1990 C Standard (C90) のすべての機能と、C90 Addendum 1 におけるワイドキャラクター、複素数、可変長配列 (VLAs)、IEEE754 の浮動小数点の演算、一般的な関数を含む最新の ISO 1999 C Standard (C99) における主要な新機能をサポートしています。Ch は、POSIX および socket/Winsock、Windows、X11/Motif、OpenGL、ODBC、GTK+などの多くの業界標準はもちろんのこと、オブジェ

クトベースのプログラミングのための、C++のクラス、オブジェクト、そしてカプセル化など、Ch は C に対して、多くの拡張機能をもっています。Ch の主な機能をまとめると、以下の通りになります。

- **新規の学習を必要としない**　すべての C プログラマが、新しい言語を学ぶことなしに、Ch 仮想マシンで C のコードを実行することにより、Ch の使用を開始することができます。C 言語の機能を使って、すべてのタスクを完了することが可能です。これにより、多くの異なった言語構文を学習し、覚えるといった苦労を最小限にすることができます。

- **インタプリタ**　Ch では C のプログラムを、コンパイル、リンク、実行、デバッグといった煩わしい手順を必要とせずに実行することができます。

- **対話的**　コードを 1 行ずつ入力して、C コードを対話式に実行することができます。したがって、初心者が C を学習するに当たっては、非常に直感的です。Ch は、C におけるプログラミング教育と学習を最新の C99 の機能を用いて行うための、非常に効果的なツールです。さらに、新しい機能を容易にテストすることができます。Ch は、リアルタイムの対話的なコンピューティングのための理想的な環境です。

- **数値計算**　計算処理に加えて、C における、char、int、flow、double、および ISO C99 で導入された、新たなタイプの複素数と可変長配列 (VLA) のようなあらゆる C のデータ型のサポートに加えて、Ch は計算配列をファーストクラスオブジェクトとして扱います。微分方程式の解法、積分、Fourier 解析のような多くの高水準数値関数、さらには、2D/3D プロット機能をもつことにより、Ch は工学および科学の問題を解決するに当たって、非常に強力な言語環境になっています。2D/3D プロット機能を使用したプログラムは、C++の SoftIntegration Graphicl Library を用いて、C++コンパイラ上でもコンパイルすることが可能です。

- **超高水準言語**　Ch は低水準言語と超高水準言語（very-high level language(VHLL)）とのギャップの橋渡しを行います。C のスーパーセットとして、ハードウェアインターフェイスのためのメモリへのアクセスのような低水準言語の機能を保持しています。コマンドシェルとしての Ch はまさに超高水準言語です。数千行の C コードを必要とするような問題であっても、Ch コードでは、ほんの数行で容易に解決することができます。

- **オブジェクトベース**　Ch は、単純化された I/O ハンドリングはもちろん、データの抽象化および情報隠蔽を伴ったオブジェクトベースのプログラミングのため、C++におけるクラスとカプセル化をサポートしています。たとえば、単一の制御クラスのみが、高水準システム設計および解析のための Ch Control System Toolkit の実装に使用されます。Ch の単純さを保つために、Ch では C++の複雑な機能は省かれています。

- **テキスト処理**　Ch はビルトインされた型と **foreach**-ループのような、拡張テキスト処理機能をもっています。これらの機能は、システム管理、シェルプログラミングおよび Web アプリケーションで特に有用です。

- **クロスプラットフォームシェル**　Ch はユーザーに便利なユニバーサルシェルを提供します。Ch は、Windows における MS-DOS シェル以外に、Unix における C-シェル、Bourne シェル、Bash、tcsh、または Korn シェルに似たログインコマンドシェルとして、使用可能です。Ch は、繰り返しのタスクの自動化、迅速なプロトタイピング、回帰テスト、および異なるプラットフォームを越えたシステム管理のための多くの内蔵機能をもっています。

- **安全なネットワークコンピューティング**　Safe Ch は最初から、sandbox、プログラマ/管理コントロール、抑制されたポインタ、限定された関数、文字型のための自動メモリ制御、ネットワークコンピューティングのために効果的なアドレスセキュリティ問題をチェックする自動配列境界、などのような異なるセキュリティ層を用いて設計されています。

- **ポーティング可能**　C 標準に準拠したプログラムはポーティング可能です。しかし、コンパイルプロセスはプラットフォームに依存します。Ch のプログラムは、Windows や Unix を含む異なるプラットフォーム間で実行することが出来ます。プログラマは 1 台のマシン上で、プログラムを開発、管理し、これを Ch がサポートしているすべてのプラットフォーム上に配布することができます。

- **ライブラリ**　Ch ライブラリは、すべての既存の C ライブラリとモジュールを含んでいます。すなわち、Ch ライブラリは無限大の可能性をもっています。たとえば Ch は、POSIX、TCP/IP socket、Winsock、Win32、X11/Mofit、GTK+、OpenGL、ODBC、LAPACK、XML、NAG 統計ライブラリ、コンピュータビジョンとイメージプロセッシングのインテル OpenCV、National Instruments'NI-DAQ と NI-Motion、正規表現のための PCRE などをサポートしています。

- **バイナリのモジュールとのインタフェース**　Ch SDK を使用することで、Ch は新しいプロセスを再起動せずに、バイナリオブジェクトとのインターフェースを実現することができます。Ch はシームレスに異なるコンポーネントを統合することができます。Ch プログラムはレガシーシステムと既存の C/C++コードとの統合のために、静的もしくはダイナミックライブラリ内の関数を呼ぶことが出来ます。またその逆に、バイナリオブジェクト中の関数は Ch 関数を呼ぶこともできます。

- **有効化された *Web***　開発、Web サーバ用 CGI(Common Gateway Interface) のためのクラスのような、開発モジュールを用いると、Ch で Web ベースのアプリケーションとサービスの迅速に開発することが可能になります。

- **組み込み可能**　Ch は組み込み可能です。Embedded Ch は、他のアプリケーションプログラム、ハードウェアおよび携帯型の装置に組込むことができます。これにより、ユーザは、異なるプラットフォームにわたる、開発および独占的なスクリプト言語の管理から解放されます。

このドキュメントの構成

Ch は C のすべての機能を含んでいます。1 章では、Ch 言語環境で Ch の簡潔な概要と C/C++プログラムの動作方法に関して説明しています。2 章、5-10 章、14 章、17-18 章、20 章では C 言語の機能について説明します。12 章、13 章、15 章では、IEEE の浮動小数点の演算とタイプジェネリックな数学関数、複素数、および可変長配列 (VLA)、などの C99 で追加されたそれぞれの新機能について説明します。11 章と 19 章は、C++において利用可能な参照型およびクラスの特徴をそれぞれ示します。任意の C コンパイラのように、Ch には、異なる設定と構築をもったいくつかの独自の機能があります。3 章と 24 章に記述された機能は、インタープリタとしての Ch の設定と構築に関連しています。23 章で説明されている二次元および三次元プロット機能は、Ch と C++の両方で利用可能です。16 章で説明されている計算配列と 21 章で説明されている Safe Ch は Ch だけで利用可能です。24 章の中の、計算配列に基づいた、高度な数値関数は、工学と科学の多くのアプリケーションに便利です。

付録 A は既知の問題とプラットフォーム特有の機能のリストを提供します。

付録 B 処理系定義の動作および、C に対する Ch の拡張機能についての情報を提供します。

付録C は Ch と C++の違いを比較します。Ch はポータブルコマンドシェルです。

付録 D 、E、および Fはそれぞれ C シェル、MATLAB、および Fortran と Ch の比較情報を提供します。

付録G は異なるプラットホームにわたる、Ch におけるポータブルシェルプログラミングで共通に使用されるコマンドのリストを提供します。

第I部

言語の機能

第1章　はじめに

本章では Ch 言語環境[1]の概要について説明します。

1.1　起動

1.1.1　Unix での起動

任意のコマンドシェルでコマンド `ch` を入力して、Ch コマンドシェルを起動できます。

Mac OS X x86 での起動

ソフトウェアをいったんダウンロードし、インストールすると、Application フォルダからダッシュボード上の **Ch** アイコンをクリックして、Ch コマンドシェルを起動できます。

Ch Professional または Student Edition では、さらに ChIDE を起動するために、Application フォルダからダッシュボード上の ChIDE アイコンをクリックすることもできます。そして次に、ChIDE 上の **Ch** アイコンをクリックして、Ch コマンドシェルを起動します。

Linux での起動

Linux では、スタートアップメニューにあるエントリ `System Tools` の下の **Ch** アイコンをクリックして、Ch を起動できます。さらにスタートアップメニューにある `Run Program` をクリックすることもできます。そして次に、**ch** を入力して、`Run in terminal` をチェックし、Ch を起動します。

Ch Professional または Student Edition では、スタートアップメニューにあるエントリ `Programming Tools` の下の ChIDE アイコンをクリックすることもできます。そして次に、ChIDE 上の **Ch** アイコンをクリックして、Ch コマンドシェルを起動します。

コマンド

```
ch -d
```

は、Ch 用のアイコンをデスクトップ上に作成します。Ch が ChIDE とともにインストールされている場合、ChIDE 用のアイコンもデスクトップ上に作成されます。

[1]訳注: 現行バージョンでは Windows プラットフォームのみが、日本語版として提供されています。

Unix 上で Login Shell として起動

直接システムに接続されている端末を通して、または、インターネットかローカルエリアネットワークに接続されているモデムを通して Unix コンピュータシステムにログインできます。システムにログインするには、システムログインプロンプトでユーザー名を入力します。すると、*login* プログラムの実行が開始されます。端末には ”*password:*” という文字列が表示され、ユーザーからのパスワード入力を待機する状態になります。パスワードを入力すると、ログイン名が */etc/passwd* ファイル内の対応するエントリと照合され、検証されます。同様に、パスワードがチェックされます。*/etc/passwd* ファイルには、システムのユーザーごとに 1 つの行が保存されています。この行で指定される情報として、ログイン名、ホームディレクトリ、ログイン時に実行を開始するプログラムなどがあります。ログイン処理後に起動するプログラムは、最後のコロンの後に指定されています。最後のコロンの後に何も指定されていない場合は、既定で、システムは Bourne シェル/bin/sh を使用します。たとえば、*/etc/passwd* に、`harry`, `john`、および `marry` というシステムの 3 人のユーザーに関する 3 つの行が保存されているとします。

```
harry:x:121:15::/home/harry:/bin/ch
john:x:125:20::/home/john:/bin/csh
marry:x:130:25::/usr/data:/usr/data/bin/word_processor
```

ユーザー `harry` へのホームディレクトリは */home/harry* です。`harry` がシステムにログインすると、Ch シェルが実行を開始します。ユーザー `john` のホームディレクトリは */home/john* であり、`john` がログインすると C シェルが起動します。`marry` がシステムにログインすると、*word_processor* プログラムが起動します。このプログラムは、たとえば専用の文書処理ソフトウェアパッケージです。クライアントとして機能する別のワークステーションにリモートでログインできます。ただし、ローカルワークステーションによってクライアントとの接続が拒否される場合があります。その場合はリモートクライアントでエラーが発生し、エラーメッセージが表示されます。どのサーバーがクライアントの出力を受け取るかをクライアントが判別できるように、適切な通信が確立されなければなりません。同時に、ワークステーションの X サーバーは、リモートシステムによる出力の送信を許可します。これは、クライアント上で環境変数 *DISPLAY* を設定し、*xhost* コマンドを使用して、ワークステーションの X サーバーにあるリモートシステムの名前リストにクライアントを追加することによって、実現します。たとえば、ローカルマシン *cat* からリモートマシン *mouse* にログインし、*mouse* の出力を *cat* に送信するには、次のコマンド

cat> `xhost mouse`

をローカルマシン *cat* で実行し、次のコマンド

mouse> `putenv("DISPLAY=cat:0.0")`

をリモートマシン *mouse* で実行します。マシン *mouse* を頻繁にリモートで使用する場合は、マシン *mouse* のホームディレクトリにあるスタートアップファイル *.chrc* にコマンド `putenv("DISPLAY=cat:0.0")` を保存し、ローカルマシン *cat* のスタートアップファイルに次のエイリアスを設定します。

```
alias("mouse", "xhost mouse; rsh mouse");
```

次に、コマンド *mouse* は、ローカルな X サーバーのリモートシステムのリストにリモートマシン *mouse* を追加し、リモートログインプロセスを開始します。コマンド *hostname* でホストマシンの名前を取得できます。

Ch がログインシェルである場合は、すぐに Ch 言語環境を使用できます。そうでない場合は、端末プロンプトでコマンド `ch` を入力して Ch 言語環境を起動できます。

X ウィンドウシステムで xterm コマンドシェルを起動しているユーザーが、マウスを使用してウィンドウの境界をドラッグすることでウィンドウのサイズを変更した場合は、コマンド **resize** によって端末を現在の xterm ウィンドウのサイズに設定できます。コマンド **resize** は Ch をコマンドシェルとして認識しないので、ユーザーは Ch シェルで関数 `_resize()`、環境変数 **COLUMNS** および **LINES** を現在の xterm ウィンドウのサイズに設定できます。

1.1.2 Windows での起動

ソフトウェアをダウンロードしてインストールした後、Ch 言語環境を起動するには、4 つの方法があります。たとえば Ch Professional Edition 6.3 を起動するには、次のようにします。

1. デスクトップ画面で **[Ch Professional]** アイコンをクリックして、MS-DOS に似た標準 Ch シェルを起動します。
2. **[スタート]** –> **[プログラム]** –> **[SoftIntegration Ch 6.3 Professional]** –> **[Ch 6.3]**。
3. **[スタート]** をクリックし、次に **[ファイル名を指定して実行]** をクリックして、`ch.exe` と入力します。
4. MS-DOS プロンプトに移動し、`ch` と入力します。

Ch Professional または Student Edition では、デスクトップ画面上の ChIDE アイコンをクリックして、ChIDE を起動できます。

1.2 コマンドモード

Ch が起動されるか、Ch プログラムが実行されるとき、既定では、Unix では.chrc、Windows では _chrc のスタートアップファイルがユーザーのホームディレクトリに存在する場合は、そのファイルが実行されます。通常、このスタートアップファイルは、コマンド、関数、ヘッダーファイルなどの検索パスを設定します。

Windows では、既定のセットアップでのスタートアップファイル _chrc が Ch のインストール時にユーザーのホームディレクトリ上に作成されます。しかしながら Unix においては、既定ではユーザーのホームディレクトリにスタートアップファイルはありません。

システム管理者はユーザーのホームディレクトリにスタートアップファイルを追加できます。ただし、ホームディレクトリにまだスタートアップファイルがない場合、ユーザーは次のようにオプション -d を使用して Ch シェルを実行し、

```
> ch -d
```

ディレクトリ CHHOME/config からユーザーのホームディレクトリにサンプルのスタートアップファイルをコピーできます。CHHOME は文字列”CHHOME”ではなく、Ch がインストールされているファイルシステムのパスを意味することに注意してください。

Linux では、このコマンドにより、Ch 用のアイコンもデスクトップ上に作成されます。Ch が ChIDE とともにインストールされている場合、ChIDE 用のアイコンもまた、デスクトップ上に作成されます。

よく知られている次のプログラミング出力ステートメントを使用して、Ch 言語環境を紹介します。

```
hello, world
```

このステートメントは、1978 年にカーニハンとリッチーが一般に広めました。このステートメントを出力する際の難易度のレベルが、その他の基準と共に、言語の簡単さと親しみやすさを判断するのにしばしば使用されます。以前の C または FORTRAN の経験があるユーザーは、このステートメントを出力するためには、最初にコンパイルとリンクのプロセスを通して実行可能なオブジェクトコードを作成し、次にプログラムを実行して出力を得るという手順が必要なことを覚えているでしょう。大規模なプログラムの場合は、プログラムの整合性を維持するために `make` ユーティリティを使用する必要があります。

Ch プログラムの実行には、コンパイルとリンクのプロセスは不要です。Ch は対話的に使用でき、迅速なシステム応答を提供します。

具体的な例として、C シェルの画面のプロンプトを次に示します。

```
%
```

システムからの出力は、このシステムプロンプトに示されているように、斜体で表示されます。Ch 言語環境を起動するには、端末キーボードで `ch` と入力します。画面は次のようになります:

```
>
```

このプロンプトは、システムが Ch 言語環境になっており、ユーザーの端末のキーボード入力を受け入れる準備ができていることを示します。ファイル */etc/passwd* で Ch を既定のシェルとして設定することもできます。このように設定した場合は、前のセクションで説明したように、ユーザーがログインすると常に Ch プログラミング環境が自動的に起動されます。

Ch コマンドプロンプトでの構文的に正しい端末入力は、実行されます。コマンドが正しく完了すると、Ch プロンプト > が再び表示されます。実行が失敗した場合、エラーメッセージが表示されます。Ch プロンプト > では、`cd`, `ls`、`pwd` などの任意の Unix コマンドを実行できます。

このシナリオでは、Ch は Unix シェルとして、Bourne シェル、C シェル、または Korn シェルと同じ方法で使用されます。たとえば、現在の作業ディレクトリを出力するには、`pwd` と入力します。画面は次のようになります。

```
> pwd
/usr/local/ch
>
```

端末から入力したコマンドはタイプライタフォントで表されます。Ch では、コマンド実行の結果としてシステムから出力があれば表示されます。この場合、`/usr/local/ch` が現在の作業ディレクトリであると仮定して、`pwd` コマンドの実行による出力は上のようになります。

Ch は ISO C のスーパーセットであるため、従来の Unix シェルよりも強力です。式を入力すると、Ch による評価結果がすぐに出力されます。たとえば、`1+3*2` という式を入力すると、出力は 7 になります。入力が `8` ならば、出力は 8 になります。このコマンドモードで任意の有効な Ch 式を評価できます。したがって、Ch を初めて使用するユーザーも、これを電卓として使用できます。**help** コマンドでは、Ch の新規ユーザーが作業を開始する手助けになる、わかりやすい例を参照できます。

```
> help
(表示メッセージ ...)
>
```

C プログラマが最初に覚えるのは、標準 I/O 関数 **printf()** を使用して `hello, world` という出力を得ることかもしれません。Ch は C のスーパーセットであるため、I/O 関数 **printf()** によって、次のように出力を得ることができます:

```
> printf("hello, world")
hello, world
>
```

_path のようなシステム変数を含むすべての変数は、C の構文では **printf()** 関数、C++の構文では **cout** を使用して出力できます。対話的なコマンドモードで、変数名を入力するだけで変数の値を表示できます。次に例を示します。

```
> int i
> i = 10
> i*i
100
> printf("\%d", i)
10
> cout << 2*i
20
>
```

Ch では、以下の 4 つの関数が環境変数の処理に使用できます。関数 **putenv()** ではシステムに環境変数を追加できます。関数 **getenv()** は、与えられた環境変数の値を返します。関数 **remenv()** では環境変数を削除できます。関数 **isenv()** では、記号が環境変数であるかどうかをテストできます。

これらの関数を適用した対話的なコマンド実行を次に示します。

```
> putenv("ENVVAR=value")
> getenv("ENVVAR")
value
> isenv("ENVVAR")
1
> remenv("ENVVAR")
> isenv("ENVVAR")
0
>
```

システムには、オンラインドキュメント付きの数百個のコマンドがあります。誰もそのすべてを把握してはいません。熟練したコンピュータユーザーの場合、いつも決まって使用する少数のツールがあり、その他にどのようなツールがあるかについては漠然と理解しています。付録 G では、一般的なコマンドを機能別に分けて一覧に示します。コマンドおよびコマンドラインオプションの詳細は、`man` コマンドに続けて、調べたいコマンドの名前を入力することで参照できます。

1.3 プログラムモード

1.3.1 コマンドファイル

Ch 言語環境では、C プログラムをコンパイルすることなく実行できます。Ch のコマンドライン引数インタフェースは、C と互換性があります。Ch では、C プログラムは、**コマンドファイル** または単に **コマンド** と呼ばれます。コマンドファイルには読み取り許可と実行許可の両方が与えられますていなければばなりません。Ch では、コマンドファイルをコンパイルすることなく実行できます。たとえば、テキストエディタで `hello.c` というコマンドファイルを作成します。プログラム `hello.c` は次のとおりであるとします。

```
/* A simple program */
#include <stdio.h>
int main() {
    printf("hello, world\n");
    return 0;
}
```

コマンド `hello.c` を入力すると、次のように `hello, world` と出力されます。

```
> hello.c
hello, world
>
```

ファイルを Unix のコマンドとして使用するためには、ファイルは実行可能でなければばなりません。プログラム `hello.c` を実行可能にするには、次のコマンドを実行することが必要な場合があります。

```
chmod +x hello.c
```

ファイル拡張子が.ch であるプログラムは、Ch で実行できます。たとえば、上記のプログラムをファイル `hello.ch` に保存した場合、次のようにして実行できます。

```
> hello.ch
hello, world
>
```

コマンドモードでコマンドファイルを実行するには、ファイル名が Ch の有効な識別子でなければばなりません。つまり、'`./`'、'`../`'、'`~/`'、'/' などの相対または絶対ディレクトリパスから始まっている必要があります。たとえば、上記のファイル名を `hello.ch` から `20` に変更すると、そのファイル名は数値になり、識別子ではなくなります。

```
> mv hello.ch 20
> 20
20
> ./20.ch
hello, world
>
```

多くの統合開発環境 (IDE) が Ch をサポートしています。たとえば、図 1.1に示すように、ChIDE を使用してプログラムの編集と実行を行うことができます。

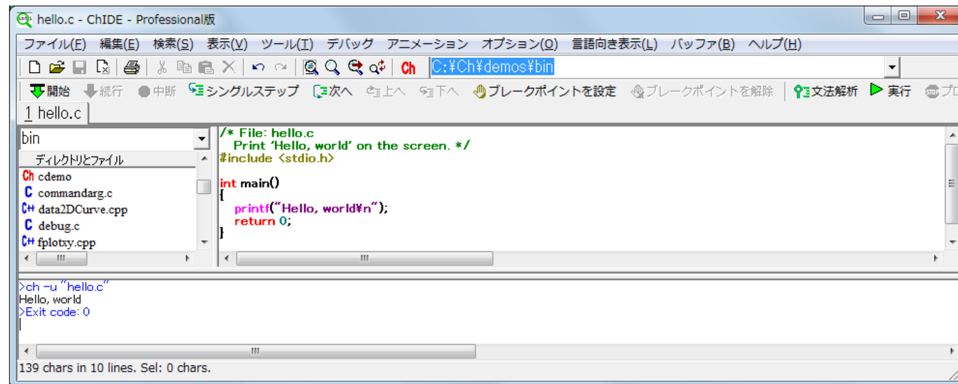


図 1.1: ChIDE を使用したプログラムの編集と実行

1.3.2 スクリプトファイル

main() 関数を使用していない、または `#!/bin/ch` から始まっているプログラムは、**スクリプト** と呼ばれます。Ch では、ステートメント、関数、およびコマンドを**スクリプトファイル**または**スクリプト** として分類できます。コマンドファイルと同様に、スクリプトファイルにも読み取り許可と実行許可が与えられなければなりません。たとえば、スクリプトファイル `prog` に次のステートメントが記述されているとします。

```
#!/bin/ch
int i = 90;
/* copy hello.c to hello.ch */
cp hello.c hello.ch
printf("i is equal to %d from the script file\n", i);
```

このスクリプトファイルは、次のようにして対話的に実行できます。

```
> prog
i is equal to 90 from the script file
>
```

または、次のように別々の 2 つのステップで実行できます。

```
> chparse prog
> chrun
i is equal to 90 from the script file
>
```

最初にコマンド `chparse prog` でスクリプトファイル `prog` を解析し、次に組み込みコマンド `chrun` で、解析されたプログラムを実行します。スクリプトファイル `prog` を実行すると、

ステートメント `cp hello.c hello.ch` により、ファイル `hello.c` が `hello.ch` という Ch プログラムにコピーされます。`.ch` というファイル拡張子が付いているため、プログラム `hello.ch` を Ch 言語環境でコマンド `hello` として実行できます。

C シェルや Korn シェルなどの他のスクリプト言語でプログラム `prog` を呼び出すことができます。

1.3.3　関数ファイル

Ch プログラムは多数の個別ファイルに分割できます。各ファイルは、プログラムのあらゆる部分にとってアクセス可能な、最上位レベルにある多数の関連した関数を含むことができます。複数の関数が記述されているファイルには、通常、Ch プログラムの一部として認識されるためのサフィックスである `.ch` が付加されます。コマンドファイルとスクリプトファイルの他に、Ch には関数ファイルも存在します。Ch の **関数ファイル** は、一つの関数定義だけを含むプログラムです。関数ファイルには読み取り許可が与えられなければなりません。既定では、関数ファイルの拡張子は.chf です。関数ファイルの名前と、関数ファイル内の関数定義の名前は、同じ名前になります。関数ファイルを使用して定義した関数は、Ch プログラミング環境のシステム組み込み関数と同様に扱われます。たとえば、プログラム `addition.chf` に次のステートメントが含まれていると仮定します。

```
int addition(int a, int b) {
  int c;
  c = a + b;
  return c;
}
```

このプログラムを自動的に呼び出して、次の対話的な実行セッションに示すように、2 つの整数を追加加算できます。

```
> int i = 9
> i = addition(3, i)
12
>
```

このプログラムでは最初に、整数値 3 と、9 の値を持つ整数型の変数 i を関数 `addition()` によって加算し、次に、その結果を変数 i に代入します。この場合、関数 `addition()` は、**sin()** や **cos()** のような組み込み関数と同様に扱われます。

1.4　複素数

次のような二次方程式があるとします。

$$ax^2 + bx + c = 0$$

これは、次のような公式によって解くことができます。

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \tag{1.1}$$

公式 (1.1) に従い、次の方程式

$$x^2 - 5x + 6 = 0$$

に対する 2 つの実数解 ($x_1 = 2$) と ($x_2 = 3$) をプログラム 1.1 から得ることができます。

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

int main() {
    double a = 1, b = -5, c = 6, x1, x2;
    x1 = (-b +sqrt(b*b-4*a*c))/(2*a);
    x2 = (-b -sqrt(b*b-4*a*c))/(2*a);
    printf("x1 = %f\n", x1);
    printf("x2 = %f\n", x2);
}
```

プログラム 1.1: $x^2 - 5x + 6 = 0$ の解

プログラム 1.1 の出力は次のとおりです。

```
x1 = 3.000000
x2 = 2.000000
```

次の方程式

$$x^2 - 4x + 13 = 0$$

に対して、($x_1 = 2 + i3$) および ($x_2 = 2 - i3$) という 2 つの複素数の解が存在します。これらの複素数は double 型で表すことができません。この方程式を実数域で解こうとすると、結果は無効になります。Ch はこの無効な結果を、Not-a-Number（非数値）を表す NaN として報告します。プログラム 1.2 とその出力を参照してください。

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

int main() {
    double a = 1, b = -4, c = 13, x1, x2;
    x1 = (-b +sqrt(b*b-4*a*c))/(2*a);
    x2 = (-b -sqrt(b*b-4*a*c))/(2*a);
    printf("x1 = %f\n", x1);
    printf("x2 = %f\n", x2);
}
```

プログラム 1.2: 実数域での $x^2 - 4x + 13 = 0$ の解

プログラム 1.2 の結果は以下のようになります。

```
x1 = NaN
x2 = NaN
```

複素数を使用すると、$x_1 = 2 + i3$ および $x_2 = 2 - i3$ という 2 つの複素数の解を持つ方程式

$$x^2 - 4x + 13 = 0$$

をプログラム 1.3 で解くことができます。

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

int main() {
    double complex a = 1, b = -4, c = 13, x1, x2;
    x1 = (-b +sqrt(b*b-4*a*c))/(2*a);
    x2 = (-b -sqrt(b*b-4*a*c))/(2*a);
    printf("x1 = %f\n", x1);
    printf("x2 = %f\n", x2);
}
```

プログラム 1.3: 複素数域での $x^2 - 4x + 13 = 0$ の解

プログラム 1.3 の出力は次のようになります。

```
x1 = complex(2.000000,3.000000)
x2 = complex(2.000000,-3.000000)
```

1.5 計算配列

Ch の配列は、ISO C と互換性があります。Ch の配列はポインタで密接に結合されます。数値計算とデータ分析の目的で Ch に導入された計算配列は、Ch の Professional Edition および Student Edition で利用可能です。Ch では計算配列をファーストクラスオブジェクト[2]として扱うことができます。たとえば、次のような配列式があるとします。

$$\mathbf{x} = \mathbf{Ab} + \mathbf{3b} \tag{1.2}$$

ここで、

$$\mathbf{A} = \begin{bmatrix} 1 & 2 & 2 \\ 4 & 4 & 6 \\ 7 & 8 & 9 \end{bmatrix} \quad \mathbf{b} = \begin{bmatrix} 5 \\ 6 \\ 8 \end{bmatrix}$$

この配列式をプログラム 1.4 で計算できます。配列式 **Ab+3b** は、3 つの異なる方法を使用して計算されます。

[2]訳注：ファーストクラスオブジェクトとは、プログラミング言語の中で他のオブジェクトに比べ追加の制限なく使用可能な実体をいいます。たとえば Ch では、プログラム 1.4 にもありますように配列型データを整数型等のデータと全く同様に扱うことができます (式 (1.2) をそのままプログラム内にコーディングすることができます) が、他のプログラミング言語において必ずしもそれが可能であるとは限りません。

```
#include <stdio.h>
#include <array.h>
void arrayexp1(array double A[3][3], array double b[3], array double x[3]) {
  x = A*b+3*b;
}

array double arrayexp2(array double A[3][3], array double b[3])[3] {
  array double x[3];
  x = A*b+3*b;
  return x;
}

int main() {
  array double A[3][3] = {{1,2,2},
                          {4,4,6},
                          {7,8,9}};
  array double b[3] = {5,6,8}, x[3];
  x = A*b+3*b;
  printf("x = %.3f", x);

  arrayexp1(A, b, x);
  printf("x = %.3f", x);

  x = arrayexp2(A, b);
  printf("x = %.3f", x);
}
```

プログラム 1.4: 配列式 **Ab+3b** の計算

プログラム 1.4を実行すると、出力は次のようになります。

```
x = 48.000 110.000 179.000
x = 48.000 110.000 179.000
x = 48.000 110.000 179.000
```

プログラム 1.4 では、配列 `A`, `b` および `x` を double 型の計算配列として宣言しています。計算配列用の型修飾子のマクロである `array` は、ヘッダーファイル **array.h** に定義されています。プログラムでは、計算配列を使用するためにこのヘッダーファイルをインクルードします。宣言時に配列 `A` と `b` の値を初期化します。最初に、関数 `main()` で配列 `x` の値を計算します。次に、関数 `arrayexp1()` で配列 `x` の値を計算し、関数の引数を通して結果をメインプログラムに戻します。最後に、関数 `arrayexp()` で配列 `x` の値を計算し、3 つの要素を持つ double 型の計算配列を返します。Ch では、C の配列である可変長配列 (VLA)、および計算配列の処理が非常に便利になっています。VLA の詳細については、後の章で説明します。

たとえば、次のような連立一次方程式があるとします。

$$\mathbf{A}\mathbf{x} = \mathbf{b} \tag{1.3}$$

ここで、

$$\mathbf{A} = \begin{bmatrix} 1 & 2 & 2 \\ 4 & 4 & 6 \\ 7 & 8 & 9 \end{bmatrix} \quad \mathbf{b} = \begin{bmatrix} 5 \\ 6 \\ 8 \end{bmatrix}$$

これをプログラム 1.5で解くことができます。

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

int main() {
  array double A[3][3] = {{1,2,2},
                          {4,4,6},
                          {7,8,9}};
  array double b[3] = {5,6,8}, x[3];
  linsolve(x, A, b);
  // or x = inverse(A)*b;
  printf("x = %.3f\n", x);
}
```

プログラム 1.5: Ax=b の解

プログラム 1.5では、関数 `inverse()` は計算配列を返します。プログラムの出力は次のようになります。

```
x = -5.000 2.000 3.000
```

1.6 プロット

便利なプロットライブラリが Ch Professional Edition および Ch Student Edition で利用可能です。プログラム 1.6は、x の範囲が $-\pi < x < \pi$ である関数 $sin(x)$ をプロットします。プログラム 1.6 の出力を図 1.2 に示します。

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

int main() {
    array double x[100], y[100];       // Use 100 data points
    char *title="sine wave",           // Define labels
         *xlabel="radian",
         *ylabel="amplitude";

    lindata(-M_PI, M_PI, x);          // X-axis data
    y = sin(x);                        // Y-axis data
    plotxy(x,y,title,xlabel,ylabel);   // Call plotting function
}
```

プログラム 1.6: $-\pi < x < \pi$ の関数 $\sin(x)$ のプロット

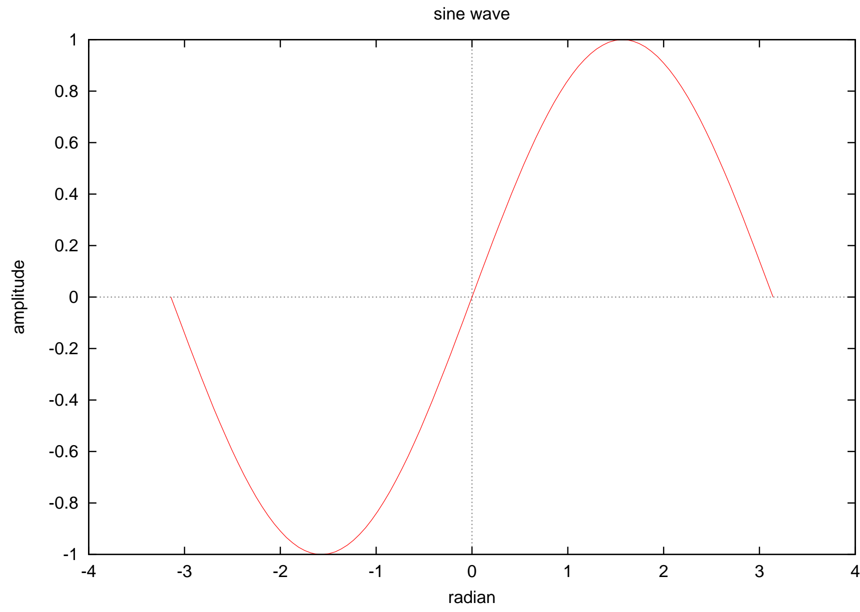


図 1.2: $-\pi < x < \pi$ の関数 $\sin(x)$ のプロット

第2章　字句要素

Ch ソースファイルは、ある特定の文字セットから選択された文字のシーケンスです。言語の最小の字句要素はトークンです。トークンの種類には、キーワード、識別子、定数、文字列リテラル、および区切り子があります。定数と文字列リテラルについては、第 6章で説明します。

2.1　文字セット

Ch で使用される文字セットは、以下のメンバで構成されます。
26 個のローマ字の大文字

```
A B C D E F G H I J K L M
N O P Q R S T U V W X Y Z
```

26 個のローマ字の小文字

```
a b c d e f g h i j k l m
n o p q r s t u v w x y z
```

10 個の 10 進数字

```
0 1 2 3 4 5 6 7 8 9
```

以下の 31 個の図形文字

```
! " # % & ' ( ) * + , - . / :
; < = > ? [ \ ] ^ _ { | } ~ $ `
```

空白文字。水平タブ、垂直タブ、および改ページを表す制御文字。
警告、バックスペース、キャリッジリターン、および改行を表す制御文字。
図形文字$ (ドル記号) と ` (アクサングラーブまたは逆引用符) は C 標準の一部ではありません。ドル記号$は、コマンドモードではイベント指示子として使用され、コマンドモードとプログラムの両方で変数置換に使用されます。

アクサングラーブ (`) は、コマンド置換に使用されます。

2.1.1　三連文字

Ch では、 Ch のユーザーが ISO646-1083 不変コードセットを使用して Ch プログラムを記述できるように、ソースファイルに現れる 2 つの連続した疑問符で始まる次の 3 つの文字の並び (三連文字シーケンスと呼びます) をすべて、以下に示す対応する単一文字に置き換えます。

??=	#	??)	]	??!	|
??(	[	??'	^	??>	}
??/	\	??<	{	??-	~

三連文字シーケンスはこれら以外にはありません。上記に示すいずれかの三連文字の先頭にない個々の**?**は、変換されません。

上記の3文字の並びを、置き換え文字として解釈させたくない場合は、文字のエスケープコード`'\?'`を使用します。次に例を示します。

```
> printf("??!")
|
> printf("?\?!")
??!
>
```

たとえば、文字列`"?\?!"`は文字のエスケープコード`'\?'`を含んでいるため、`"??!"`という文字列を表す際に使用できるのに対して、三連文字形式の`"??!"`は、`|`という文字を表します。エスケープコードについては、セクション6.3.1で説明します。

2.2　キーワード

2.2.1　キーワード

以下のシンボルは、Ch の既定のキーワードです。プログラムにおけるこれらの意味は、C 標準および Unix/C 仕様の解釈に従います。

言語構文キーワード

ComplexInf ComplexNaN Inf NaN NULL auto break const complex char case continue class double default delete do else enum extern float for foreach fprintf goto if inline int long new operator printf private public register restrict return scanf static struct short signed sizeof string_t switch this union unsigned volatile void while

C++からのキーワード　以下のキーワードの意味は C++における意味と同じです。
class delete new private public this

C/C++にないキーワード　Ch には、以下の追加キーワードが加えられています。
ComplexInf ComplexNaN Inf NaN NULL foreach fprintf printf scanf string_t

NaN というシンボルは、Not-a-Number (非数値) を意味し、**Inf** は無限大 ∞ の数値を表します。

リーマン球面における複素数の Not-a-Number と複素数の無限大は、それぞれ、**ComplexNaN** および **ComplexInf** と表されます。Ch のキーワード **NULL** は、ポインタ値 (void *)0 と整数値 0 に対するマクロ NULL の使用に整合性がないという C の問題を解決します。Ch の NULL は、ポインタ型として使用される場合は `(void*)0` という値を持ち、整数型として使われる場合は 0 の値を持ちます。

C のヘッダーファイル **stdio.h** に定義されている標準関数 **fprintf**、**printf**、および **scanf** の意味は、Ch でも保持されます。ただし、**fprintf**、**printf**、および **scanf** は Ch では拡張されています。これについては第20章で説明します。

ファーストクラスオブジェクトの新しい組み込みデータ型である **string_t** は、C の文字列に関するメモリの問題を解決するために追加されています。キーワード **foreach** が foreach-loop 構文に追加されています。主に、文字列のインデックスでループを扱うために使用されます。これら 2 つのキーワードの詳細については、第17章で説明します。

汎用関数

Ch で使用可能な汎用関数を以下に示します。

abs access acos acosh alias asin asinh atan atan2 atanh atexit ceil clock conj cos cosh dlerror dlopen dlrunfun dlsym exp elementtype fgets floor fmod fprintf fread free frexp fscanf getenv gets imag ioctl ldexp log log10 max memcpy memmove memset min modf open polar pow printf read real scanf setrlimit shape sin sinh sprintf sqrt sscanf stradd strcat strchr strcmp strcoll strcpy strerror streval strlen strncat strncpy strparse strtod strtok strtol strtoul strxfrm tan tanh transpose umask vprintf vfprintf vsprintf

汎用関数は、組み込みのシステム関数です。標準の C 関数を拡張しています。ほとんどの汎用関数はポリモーフィック（多態性：関数名等一つの表現を使って複数個の動作を使い分けられること）です。たとえば、関数呼び出し sin(x) では、組み込みのシステム関数を使用するため、引数 x を関数 sin() で有効な任意のデータ型とすることができます。たとえば、次のコードは有効です。

```
#include <array.h> // for the macro array

int i;
float f;
double df;
complex z;
double complex dz;
array double a[2][3];
array double complex az[2][3];
...
f = sin(i);
f = sin(f);
df = sin(df);
z = sin(z);
dz = sin(dz);
a = sin(a);
az = sin(az);
```

`sin(i)` の関数呼び出しの戻り値の型が引数の型とは違う点に注意してください。汎用関数の詳細については、セクション10.12で説明します。

ヘッダーファイル **chshell.h** で定義されている関数 **iskey()** は、名前が Ch のキーワードかどうかの判別に使用することができます。引数がキーワードでない場合は、0 が返されます。引数が汎用関数の名前である場合は、1 が返されます。引数がキーワードまたは予約済みのシンボルである場合は、2 が返されます。次に例を示します。

```
> iskey("abcde")
0
> iskey("abs")
1
> iskey("while")
2
>
```

2.2.2 予約済みのシンボル

以下のシンボルは、C++における継承と例外処理についての Ch の今後の拡張に備えて、予約されています。

virtual protected try catch

以下のシンボルは、マルチタスキングについての Ch の今後の拡張に備えて、予約されています。

event_t recvevent sendevent beginparalleltask endparalleltask

2.3 識別子

識別子は、変数、クラス、型、関数などの名前を形成する、文字のシーケンスです。識別子は、アンダースコア文字、ローマ字の小文字と大文字、その他の文字から構成されます。ドル記号文字 `'$'` は識別子内には使用できません。小文字と大文字は区別されます。先頭文字は、数字であってはなりません。識別子の最大長は 5119 です。

プリプロセッサトークンがトークンに変換されるとき、プリプロセッサトークンをキーワードか識別子のいずれかに変換できる場合は、キーワードに変換されます。

2.3.1 事前定義済みの識別子

表2.1に示す識別子は、 Ch では事前に定義されています。これらの事前定義済みの識別子の既定値は、表2.2に示しています。主な制限を以下に示します。

- **_ipath**、**_fpath**、および **_lpath**、**_path**、 の項目の区切り文字はそれぞれ、Unix に対しては `";:"`、Windows に対しては `";"` となります。空白は、ディレクトリパスの一部として、Unix および Windows の両方で使用できます。これらのシステム変数についての詳細は、次の章で説明します。

表 2.1: 事前定義済みの識別子

識別子	データ型	説明
_argc	int	main(int argc, char**argv) の argc に相当。
_argv	char**	main(int argc, char*argv [ ]) の argv に相当。
__class__	char []	メンバ関数内部のクラス名
__class_func__	char []	メンバ関数内部のクラス名および関数名。
_cwd	string_t	現在の作業ディレクトリ。
_cwdn	string_t	現在の作業ディレクトリ名。
_environ	char**	C 文字列に対するポインタの配列。各配列項目は環境文字列をポイントします。
_errno	int	システム呼び出しエラー番号。
_formatf	string_t	float 型の既定出力。
_formatd	string_t	double 型の既定出力。
_fpath	string_t	関数ファイルのパス。
_fpathext	string_t	関数ファイル名の拡張子。
__func__	char []	関数内部の関数名。
_histnum	string_t	保存されたコマンドの履歴数。
_histsize	int	保存されたコマンドの履歴のサイズ。
_home	string_t	ホームディレクトリ。
_host	string_t	コンピュータのホスト名。
_ignoreeof	int	true の場合、シェルは端末からの EOF を無視します。このため、Ctrl+d と入力して Ch シェルを誤って強制終了することを防ぎます。
_ignoretrigraph	int	true の場合、シェルは三連文字を無視します。
_ipath	string_t	前処理の命令#included があるヘッダーファイルのパス。
_lang	string_t	**_lc_all** と該当のシステム変数 (“_lc_”で始まる) のいずれにもロケールが指定されていない場合に、ロケールカテゴリに使用されるロケールの名前。
_lc_all	string_t	**_lang** の設定で指定されているロケールカテゴリ用の任意の値や “_lc_” で始まる任意のシステム変数のオーバーライドに使用されるロケールの名前。
_lc_collate	string_t	照合情報のためのロケールの名前。
_lc_ctype	string_t	文字情報のためのロケールの名前。
_lc_monetary	string_t	通貨関連数値の編集情報を含むロケールの名前。
_lc_numeric	string_t	数値の編集 (基数文字) 情報を含むロケールの名前。
_lc_time	string_t	日付/時間形式情報のためのロケールの名前。
_logname	string_t	ユーザーデータベースを基にした、ユーザーの初期作業ディレクトリの名前。

表 2.1: 事前定義済みの識別子 (続き)

識別子	データ型	説明
_lpath	string_t	関数 **dlopen**(const char **pathname*, int *mode*)) で使用される、動的に読み込まれるライブラリを検索するためのパス。パス名に埋め込みの/が含まれない場合は、**_lpath** のパスが最初に検索されます。その後は、ネイティブ関数の呼び出しの検索順序に従います。たとえば、環境変数 **LD_LIBRARY_PATH** は SunOS 内の検索です。
_new_handler	void (*)()	ユーザーが定義した演算子 **new** 用のハンドラ関数へのポインタ。
_path	string_t	コマンドのパス。
_pathext	string_t	コマンド名の拡張子。
_ppath	string_t	**#pragma package** <packagename>で使用される **_fpath**、**_ipath**、**_ipath** の追加パスへのパス。
_prompt	string_t	対話型の Ch シェルのプロンプト。
_setlocale	int	true の場合、ヘッダーファイル wchar.h および wctype.h にあるマルチバイト関数を処理するために、関数 `setlocale(CL_ALL, "")` が呼び出されます。
_shell	string_t	使用中のシェルの名前。
_status	int	実行されたコマンドの状態を示す exit 値。0 は正常実行、ゼロ以外は失敗。
_term	string_t	端末の種類。
_tz	string_t	端末の種類。
_user	string_t	ユーザーアカウント名。
_warning	int	3 - すべての警告メッセージ。 2 - ほとんどの警告メッセージ。 1 - 重大な警告メッセージのみ。 0 - 警告メッセージなし。

表 2.2: 事前定義済みの識別子の既定値

識別子	データ型	既定値
_argc	int	コマンドに依存
_argv	char* [ ]	コマンドに依存
__class__	static const char []	“ ”
__class_func__	static const char []	“ ”
_cwd	string_t	現在の作業ディレクトリ。cwd が使用できない場合、**_cwd** は **_home** の値を使用します。
_cwdn	string_t	現在の作業ディレクトリ名
_environ	char**	C 文字列へのポインタの配列。各配列項目は環境文字列をポイントします。
_errno	int	0
_formatf	string_t	`".2f"`
_formatd	string_t	`".4lf"`
_fpathext	string_t	“chf”

_fpath	string_t	`"CHHOME/lib/libc;CHHOME/lib/libch;"`
		`"CHHOME/lib/libopt;CHHOME/lib/libch/numeric;"`
		通常の Ch の場合
		`"CHHOME/lib/libc;CHHOME/lib/libch;"`
		`"CHHOME/lib/libch/numeric;"` セーフ Ch の場合

__func__	static const char []	“ ”
_histnum	string_t	“0” (コマンドが処理されるときに内部的に変換されます)
_histsize	int	128
_home	string_t	環境変数 HOME の値 (設定されている場合)。それ以外の場合は、ホームディレクトリ (Unix の場合) および現在の_drive:/または C:/ (Windows の場合)。
_host	string_t	コンピュータのホスト名
_ignoreeof	int	0
_ignoretrigraph	int	0
_ipath	string_t	`"CHHOME/include;CHHOME/toolkit/include;"`
_lang	string_t	“C”
_lc_all	string_t	NULL
_lc_collate	string_t	NULL
_lc_ctype	string_t	NULL
_lc_monetary	string_t	NULL
_lc_numeric	string_t	NULL
_lc_time	string_t	NULL
_logname	string_t	初期作業ディレクトリの名前
_lpath	string_t	`"CHHOME/lib/dl;CHHOME/toolkit/dl;"`
_new_handler	void (*)()	NULL

表 2.2: 事前定義済みの識別子の既定値 (続き)

識別子	データ型	既定値

_path	string_t	通常の Ch の場合
		`"CHHOME/bin/;CHHOME/sbin;`
		`CHHOME/toolkit/bin;CHHOME/toolkit/sbin;`
		`/bin;/usr/bin;/sbin;"` (Unix);
		`"CHHOME/bin/;CHHOME/sbin;`
		`CHHOME/toolkit/bin;CHHOME/toolkit/sbin;`
		`/bin;/usr/bin;/sbin;/usr/openwin/bin;"` (SunOS/Solaris);
		`"CHHOME/bin;CHHOME/sbin;`
		`CHHOME/toolkit/bin;CHHOME/toolkit/sbin;`
		`WINDIR;WINDIR/COMMAND;`
		`WINDIR/SYSTEMDIR;"` (Windows 95/98/ME);
		`"CHHOME/bin;CHHOME/sbin;`
		`CHHOME/toolkit/bin;CHHOME/toolkit/sbin;`
		`WINDIR; WINDIR/SYSTEMDIR;"` (Windows NT/2000/XP);
		セーフ Ch の場合
		`"CHHOME/sbin;CHHOME/toolkit/sbin;`
		(各種 OS すべて)

_pathext	string_t	“” (Unix の場合) および “.com;.exe;.bat;.cmd” (Windows の場合)
_ppath	string_t	`"CHHOME/package;"`
_prompt	string_t	一般ユーザーの場合は **stradd**(**_cwdn**,`"> "`)、スーパーユーザーの場合は **stradd**(**_cwdn**,`"# "`)
_setlocale	int	0
_shell	string_t	使用中のシェルの名前
_status	int	0
_term	string_t	環境変数 **TERM** の値
_tz	string_t	ローカルタイムゾーン
_user	string_t	ユーザーアカウント名
_warning	int	1

- システム変数、**_cwd**、**_cwdn**、**_home**、**_lang**、**_lc_all**、**_lc_collate**、**_lc_ctype**、**_lc_monetary**、**_lc_numeric**、**_lc_time**、**_logname**、**_path**、**_shell**、**_term**、**_tz**、**_user** が更新されるときは、対応する環境変数 **HOME**、**LANG**、**LC_ALL**、**LC_COLLATE**、**LC_CTYPE**、**LC_MONETARY**、**LC_NUMERIC**、**LC_TIME**、**LOGNAME**、**PATH**、**PWD**、**SHELL**、**TERM**、**TZ**、**USER** も更新されます。

- パス名にある **CHHOME** は文字列 "CHHOME"ではなく、Ch がインストールされているファイルシステムのパスを表します。たとえば、Windows では **CHHOME** に `C:\Ch` を、Unix では CHHOME に `/usr/local/ch` を使用します。同様に、パス名の **WINDIR** と **SYSTEMDIR** はそれぞれ、システム変数 **WINDIR** と **SYSTEMDIR** の値です。

2.4 区切り子

区切り子は、独立した構文と意味に重要性を持つシンボルです。コンテキストに依存して、区切り子は実行される処理を指定することがあります。その場合、区切り子は演算子と呼ばれます。オペランドは、演算子の動作の対象となるエンティティです。Ch では、以下の区切り子が有効です。

```
! != # ## $ % %= & && &= * *= + ++ += - -- -=
-> . .* ./ / /= < << <<= <= = == > >= >> >>= ?
^ ^= ^^ " ' ` { | |= || } ~
```

2.5 コメント

Ch には 2 つの形式のコメントがあります。Ch プログラムのコメントは、/*と*/ という区切り記号の組で囲むことができます。これら 2 つのコメント区切り記号を入れ子にすることはできません。文字定数、文字列リテラル、またはコメント内を除いて、文字/*はコメントを開始します。コメントの内容は、マルチバイト文字を識別して、終端の文字*/を特定する際にのみ検査されます。

Ch のシンボル//は、それに続くテキストの行末までをコメントにします。//を使用して/*または*/をコメントにすることができ、/* */を使用して//をコメントにすることができます。文字定数、文字列リテラル、またはコメント内を除いて、文字//は、次の改行文字まで (改行文字は含まない) の、含まれる文字がすべてマルチバイト文字であるコメントを開始します。このようなコメントの内容は、マルチバイト文字を識別して終端の改行文字を検出するためにのみ検査されます。次に例を示します。

```
"a//b"                  // four-character string literal
// */                   // comment, not syntax error
f = g/**//h;            // equivalent to f = g / h;
//\
i();                    // part of a two-line comment
/\
/ j();                  // part of a two-line comment
/*//*/ l();             // equivalent to l();
m = n//**/o
  + p;                  // equivalent to m = n + p;
```

これら 2 つの組み合わせ方式によって、コメントを含むコードのセクションをコメントにする便利な構造を実現できます。コメントが行頭で始まらないときは、//の使用を推奨します。

また、プリプロセッサディレクティブ#if、#elif、#else、および#endif を組み合わせて使用すると、コードの大きなセクションをコメントにすることができます。コマンドステートメントの引数の位置では、コメントは使用できません。次に例を示します。

```
> int i=2    // comment ok
> i*4        /* comment ok */
8
```

```
> ls         // comment bad
> cmd        /* comment bad */
```

上記の例では、コマンドステートメント **ls** と **cmd** にはコメントを適用できません。

第3章　プログラムの構造

3.1　Ch ホームディレクトリ内のディレクトリとファイル

Ch ホームディレクトリのディレクトリとファイルを表3.1に示します。

この **CHHOME** は Ch のホームディレクトリです。

CHHOME は文字列 “CHHOME”ではなく、Ch のインストール先を示すファイルシステムパスであることに注意してください。

既定のインストールでは、**CHHOME** は Windows の場合 `C:\Ch`、Unix の場合 `/usr/local/ch` となります。

表 3.1: Ch ホームディレクトリ内のディレクトリとファイル

ディレクトリ名	内容
README	重要な情報
bin	実行可能なバイナリファイル
config	構成ファイル
demos	デモプログラム
dl	動的に読み込まれるライブラリ
docs	ドキュメント
extern	他の言語およびバイナリオブジェクトとのインタフェース
include	Ch で使用されるヘッダーファイル
lib	ライブラリ
license	ライセンス情報
package	Ch パッケージ
sbin	セーフ Ch のコマンド
toolkit	ツールキット
www	Web Web 関連のプログラム

3.2　起動

Windows のデスクトップで Ch のアイコンをクリックして Ch コマンドウィンドウを起動するか、Windows のコマンドウィンドウまたは Unix のコマンドシェルに次のコマンドを入力することによって、Ch 言語環境を起動できます。

```
ch          --------  通常のシェル
```

```
ch -S     -------- セーフシェル (chs と同じ)
chs       -------- セーフシェル
```

環境変数 **CHHOME** [1]は、Ch のインストール先の最上位ディレクトリです。Unix では、たとえば /usr/ch、Windows では、たとえば C:\Ch です。表3.2のスタートアップファイルは、Ch 言語環境が呼び出されたときに実行されます。

初回起動時、Ch シェルは通常、**CHHOME/config/chrc** にあるコマンドを実行します。ユーザーのホームディレクトリにある **.chrc** ファイルも実行対象に含まれます。ただし、そのファイルが読み取り可能であることが条件です。

Unix でログインプログラムによって起動するときのように、'-' で始まる名前を指定してシェルを呼び出した場合、シェルはログインシェルとして実行されます。

この場合、シェルは、**CHHOME/config/chrc** にあるコマンド (ホームディレクトリの **.chrc** を含む) を実行した後、ホームディレクトリの **.chlogin** ファイルにあるコマンドを実行します。このファイルには、**.chrc** と同じアクセス許可の確認が適用されます。

通常、**.chlogin** ファイルには、端末の種類と環境を指定するコマンドが含まれます。

ログインシェルの終了時、ホームディレクトリの **.chlogout** ファイルにあるコマンドが実行されます。このファイルには、**.chrc** と同じアクセス許可の確認が適用されます。

-d オプションを付けて Ch を起動すると、ホームディレクトリに **.chrc** ファイルが存在するかどうかが最初にチェックされます。存在しない場合は、**CHHOME/config/.chrc** がホームディレクトリにコピーされます。

-f オプションを付けて Ch を起動すると、高速起動になります。この場合、**CHHOME/config/chrc** ファイルと **~/.chrc** ファイルは実行されません。

セーフ Ch シェルの起動手順は、通常のシェルの起動手順と同じです。ただしスタートアップファイルは、**chrc**、**.chrc**、**.chlogin**、および **chlogout** の代わりに、**chsrc**、 **.chsrc**、 **.chslogin**、および **chslogout** が使用されます。

Windows では、通常の Ch とセーフ Ch で **.chrc** と **.chsrc** の代わりに、ホームディレクトリ内のスタートアップファイル **_chrc** と **_chsrc** が使用されます。

[1]訳注：CHHOME は Ch 言語環境の中では環境変数の扱いを受け、getenv() 関数等を通して操作することができますが、他のアプリケーションからアクセスすることはできません。CHHOME の他、HOME、CHVERMAJOR、CHVERMINOR、CHVERMICRO、CHBUILD、CHEDITION、CHRELEASEDATE、MANPATH 及び PAGER も同様です。

表 3.2: Ch スタートアップファイル

Windows のスタートアップファイル	説明
~/_chrc	**CHHOME/config/chrc** で組み込みます。
~/_chsrc	**CHHOME/config/chsrc** で組み込みます。
~/_chlogin	**CHHOME/config/chlogin** で組み込みます。
~/_chslogin	**CHHOME/config/chslogin** で組み込みます。
~/_chlogout	ログアウト時にログインシェルによって読み込まれます。
Unix のスタートアップファイル	説明
~/.chrc	**CHHOME/config/chrc** で組み込みます。
	通常のシェルによる実行開始時に読み込まれます。
~/.chsrc	**CHHOME/config/chsrc** で組み込みます。
	セーフシェルによる実行開始時に読み込まれます。
CHHOME/config/chlogin	通常のシェルでのログイン時、chrc の実行後にログインシェル
~/.chlogin	によって読み込まれます。**CHHOME/config/chlogin** で組み込みます。
CHHOME/config/chslogin	セーフシェルでの chsrc ログインの実行後に、セーフ ch の
	ログインシェルによって読み込まれます。
~ /.chslogin	**CHHOME/config/chslogin** で組み込みます。
~ /.chlogout	ログアウト時にログインシェルによって読み込まれます。

既定では、関数ファイルのパスであるシステム変数**_fpath**の値は、通常のChでは“`CHHOME/lib/libc`; `CHHOME/lib/libch`; `CHHOME/lib/libopt`; `CHHOME/lib/libch/numeric`” セーフ Ch では “`CHHOME/lib/libc`; `CHHOME/lib/libch`; `CHHOME/lib/libch/numeric`”です。

上記の既定のディレクトリに置かれていない関数ファイルに定義された関数は、スタートアップファイル**.chrc**、**.chsrc**、**chrc**、および **chsrc.** では使用できません。しかし、汎用関数はスタートアップファイルで使用できます。

3.2.1　スタートアップファイルのサンプル

`CHHOME/config` ディレクトリにスタートアップファイルのサンプルがあります。Ch のインストール後、システム管理者は、さまざまなシステム構成に従ってスタートアップファイルを変更できます。

ユーザーは、ホームディレクトリにある各自のスタートアップファイルをカスタマイズできます。便利な方法として、`ch -d` コマンドを使用すると、`CHHOME/config` ディレクトリからホームディレクトリにスタートアップファイルのサンプルをコピーできます。

プログラム3.1は、Unix でのユーザーのホームディレクトリにあるスタートアップファイル**.chrc** の例です。

```
umask(0022);
_warning = 3;     // print all warning. default is 1 with serious warning message only
_format = 8;      // output format for double "%.'format'lf" and float "%.'format-2'f"
_ignoreeof = 1;   // ignore EOF. defalut is 0
_path = stradd(_path, ".;");
//_ppath = stradd(_ppath, "/my/package/path;");
//_fpath = stradd(_fpath, "/my/function/path;");
//_ipath = stradd(_ipath, "/my/headerfile/path;");
//_lpath = stradd(_ipath, "/my/dynloadlib/path;");
//_pathext = stradd(_pathext, ";.ch");

#define RLIMIT_CORE  4
struct rlimit {int rlim_cur, rlim_max;} rl={0,0};
setrlimit(RLIMIT_CORE, &rl); /* no core dump */

if(_prompt != NULL) { // change the default prompt "cwdn> "
   _prompt = stradd(_user, "@", _host, ":", _cwd, _histnum, "> ");
}
putenv("TERM=xterm");
alias("rm", "rm -i");
alias("mv", "mv -i");
alias("cp", "cp -i");
alias("ls", "ls -F");
alias("go", "cd /very/long/dir");
alias("opentgz", "gzip -cd _argv[1] | tar -xvf -");
```

プログラム 3.1: スタートアップファイル**.chrc** の例

この例では、ユーザーは、関数 **umask**(0*lmn*) を使用して、新しいファイルまたはディレクトリのアクセス許可の設定を指定できます。パラメータの 1 桁目の '0' は、8 進数を示します。

その後ろの 3 桁の *lmn* は、各アクセスグループの合計アクセスコードとして使用される 3 桁の 8 進コードを表します。左端の数値 *l* は所有者、2 番目の数値 *m* はグループ、*n* はその他すべてのユーザーを表します。

読み取りアクセス権は 4、書き込みアクセス権は 2、実行または検索アクセス権は 1 です。不要なアクセス権を無効にするには、関数 **umask()** を使用します。次のような関数呼び出しがあるとします。

```
umask(0022);
```

この関数呼び出しは、グループおよびその他に対する書き込みアクセス権を削除します。システム変数**_warning** は、シェルが警告メッセージをどのように表示するかを示します。**_warning** の値の意味は、表 2.1 で定義しています。次のようなステートメントがあるとします。

このステートメント

```
_warning = 3;
```

は、**_warning** の値を既定値である 1 から、すべての警告メッセージを表示する 3 に変更します。

float 型及び double 型の値を出力するときの既定のフォーマットはそれぞれ `".2f"` 及び `".4lf"` ですが、これらはシステム変数 **_formatf** 及び **_formatd** を再設定することで変更できます。たとえばステートメント

```
_formatf = ".6f";
_formatd = ".6lf";
```

は float 型及び double 型の値を出力するときの既定のフォーマットをそれぞれ ".6f" 及び ".6lf " に変更します。

次のステートメント

```
_ignoreeof = 1;
```

は、システム変数 **_ignoreeof** を true に設定します。したがって、シェルは端末からの **EOF** を無視します。このため、Ctrl-d と入力して Ch シェルを誤って強制終了することを防ぎます。次のようなステートメントがあるとします。

```
_path = stradd(_path, ".;");
```

このステートメントは、Ch によるコマンド検索の対象となるように、システム変数 **_path** に現在の作業ディレクトリを追加します。Unix のスタートアップファイルの既定では、上記のステートメントはコメント化されています。現在のディレクトリ内のファイルをコマンドシェルから実行可能にするには、上記のステートメントをコメント解除する必要があります。同様に、この例の **_fpath**、**_lpath**、**_ipath**、および **_ppath** の各システム変数に関する以降のコマンドは、変数にディレクトリを追加します。string 型のシステム変数 **_pathext** は、コマンドのファイル拡張子を格納します。prog.ch などの Ch コマンドをファイル拡張子 .ch を明示的に入力せずに呼び出すには、システム変数 **_pathext** にファイル拡張子 .ch を追加します。システム変数の意味と既定値については、表2.1を参照してください。

C 関数の **setrlimit()** を使用して、リソースの最大消費量を制御できます。

この関数の最初の引数は、制御されるリソースを表します。たとえば、リソース RLIMIT_CORE は、バイト単位でのコアファイルの最大サイズを示します。

2 番目の引数は、リソースの限界を表す **rlimit** 構造体です。

rlimit の **rlim_cur** メンバは現在の制限 (またはソフト制限) を指定し、**rlim_max** メンバは最大の制限 (またはハード制限) を指定します。

ソフト制限は、プロセスによってハード制限以下の任意の値に変更できます。

プロセスは、ソフト制限以上の任意の値になるように、ハード制限を小さくすることができます。

次のようなコードがあるとします。

```
#define RLIMIT_CORE  4
struct rlimit {int rlim_cur, rlim_max;} rl={0,0};
setrlimit(RLIMIT_CORE, &rl)
```

このコードは、コアファイルの作成を禁止するために、このファイルの最大サイズに対するソフト制限とハード制限の両方を 0 に変更します。

システム変数 **_prompt** は、Ch シェルの対話型プロンプトを示すシンボルを格納します。通常のユーザーに対しては、コマンド stradd(_cwdn, "> ") の実行結果が既定値です。つまり、現在の作業ディレクトリ名とシンボル ‘>’ です。

次のようなステートメントがあるとします。

```
_prompt = stradd(_user, "@", _host, ":", _cwd, _histnum, "> ");
```

プログラム3.1に示したこのステートメントは、既定のプロンプトを変更して、ユーザー名、シンボル '@'、マシン名、現在の作業ディレクトリ、コマンド履歴番号、およびシンボル '>' で構成される次のような文字列にします。"user@machine:/path/dir#> ".

環境変数には、ユーザーの環境についての特別な情報が保持されます。**putenv()** 関数と **getenv()** 関数 で環境情報を設定および取得できます。次のようなステートメントがあるとします。

```
putenv("TERM=xterm");
```

このステートメントは、環境変数 TERM の値を xterm に変更します。**TERM** は、端末の種類を示す環境変数です。**vi** などの一部のアプリケーションは、この変数に基づいて、ユーザーが使用している端末の種類を決定します。

この例の最後の部分は、**allias** コマンドに関するものです。**allias** コマンドは、頻繁に使用するコマンドまたは一連のコマンドの略語を作成します。たとえば、次のようなコマンドがあるとします。

```
alias("rm", "rm -i");
```

このコマンドは、コマンド **rm** を **rm -i** と同等にします。最も一般的に使用される Unix コマンド (**rm, mv, cp, ls**) を、Windows の Ch で使用できます。また、Ch には、すべての MS-DOS コマンドが含まれています。さまざまなオペレーティングシステムでさまざまなコマンドが使用されるため、Windows のスタートアップファイルは内容が多少異なる場合があります。たとえば、Windows の Ch では、MS-DOS コマンド **del** のコマンド別名として (alias("del", "del /P")) を設定できます。

次のような別名があるとします。

```
alias("go", "cd /very/long/dir");
```

この別名を使用すると、コマンド go を入力するだけで、現在の作業ディレクトリを/very/long/dir に変更できます。次のような別名があるとします。

```
alias("opentgz", "gzip -cd _argv[1] | tar -xvf -");
```

この別名を使用して、拡張子.tgz または.tar.gz を持つアーカイブ・ファイルの解凍と untar を実行できます。仮引数_argv[1] は、入力されたコマンド内の実際の引数に置き換えられます。たとえば、この別名を使用した次のようなコマンドがあるとします。

```
opentgz file.tar.gz
```

このコマンドは、次のコマンドと同等です。

```
gzip -cd file.tar.gz |tar -xvf -
```

別名の詳細については、セクション4.6を参照してください。

Ch をログインシェルとして使用する場合、ユーザーのホームディレクトリにあるスタートアップファイル**.chlogin** 内のコマンド stty によって、端末の特性を設定します。たとえば erase 文字をバックスペース、kill 文字を現在のコマンドラインの中止、intr 文字を現在のコマンドの中断、susp 文字を現在のコマンドの一時停止として設定します。この例では、次のようなコマンドがあります。

```
stty intr '^C' erase '^?' kill '^U' susp '^Z'
```

このコマンドは、中断文字を `Ctrl-C` に、消去文字を `Ctrl-H`、強制終了文字を `Ctrl-H`、一時停止文字を `Ctrl-Z` に設定します。コマンド `stty -a` を使用すると、現在のすべての設定を表示できます。

3.2.2 コマンドラインオプション

非対話型の Ch シェルでは、コマンドラインに引数として指定したコマンドを実行できます。次の構文を使用します。

```
ch [-Sacdfghinruw] [argument...]
```

次のコマンドラインオプションを除いて、コマンドラインに指定した文字列は、呼び出されたコマンドに引数として渡されます。

- S セーフシェルを指定します。セーフシェルでは、**system()** などの多くの関数は使用できません。ログインシェルの場合、**CHHOME/config/chsrc** と **CHHOME/config/chlogin** の実行後、多くの汎用関数は無効になります。詳細については、第 21章を参照してください。

- a アプレットなどの移植可能なコードを指定します。`CHHOME/lib/libopt` 内のプラットフォーム依存関数は使用できません。

- c 最初のファイル名引数からコマンドを読み取ります (ファイルが存在し、読み取り可能である必要があります)。残りの引数は、**_argv** の引数として渡されます。プログラムが関数 **main**(*int argc, char *argv[]*) を含む Ch コマンドである場合、関数 **main()** の `argv` にも引数が渡されます。

- d ch の起動時に、ユーザーのホームディレクトリに **.chrc** ファイルが存在するかどうかを最初にチェックします。存在しない場合は、**CHHOME/config/.chrc** をユーザーのホームディレクトリにコピーします。chs の起動時に、**.chsrc** ファイルがユーザーのホームディレクトリに存在するかどうかを最初にチェックします。存在しない場合は、**CHHOME/config/.chsrc** をユーザーのホームディレクトリにコピーします。Windows では、通常の Ch およびセーフ Ch のスタートアップファイルとして、それぞれ **.chrc** と **.chsrc** ではなく **_chrc** と **_chsrc** を使用します。

- f 高速起動を指定します。起動時に、**chrc** および **.chrc** のいずれのファイルも読み込まず、ログインシェルの場合は **chlogin** および **.chlogin** のいずれのファイルも読み込みません。

- g CGI スクリプトのデバッグ用に指定します。Web ブラウザがテキストシェルになります。

- h ヘルプとして Ch の使用方法に関するメッセージを表示します。

- i 強制的な対話型シェル用に予約されています (指定しても無視されます)。

- n コマンドを解析 (解釈) するが実行はしないことを指定します。このオプションを使用すると、Ch シェルスクリプトの構文エラーをチェックできます。システム変数 **_warning** の警告フラグは最高レベルに設定されます。すべての警告メッセージが出力されます。スタートアップファイルは解析されるだけで実行されません。

- r stderr ストリームを stdout にリダイレクトします。このオプションは、Windows オペレーティングシステムで実行しているプログラムをデバッグするときに便利です。たとえば、コマンド ch -r chcmd > junkfile は、プログラム chcmd の stderr ストリームから junkfile ファイルにエラーメッセージを送信します。

- u 主に IDE で入出力を処理するために、stdout ストリームをバッファリングしません。

- v Ch のエディションおよびバージョン番号を stdout ストリームに出力します。

- w プログラムの解析と実行の両方に対して、システム変数_**warning** の警告フラグを最高レベルに設定します。すべての警告メッセージが出力されます。

オプション-a を使用して、Ch プログラムが異なるプラットフォームに移植可能かどうかをテストできます。たとえば、次のコマンドは、プログラム cmd.ch が移植可能かどうかをテストします。

```
ch -a cmd.ch
```

オプション-g は、CGI コードをデバッグするときに特に便利です。CGI スクリプトの 1 行目が次のように始まっているとします。

```
#!/bin/ch -g
```

この場合、Web ブラウザがテキストシェルになります。この CGI スクリプトを実行すると、エラーメッセージを含むすべての出力が Web ブラウザ内に表示されます。

3.3 Ch プログラム

3.3.1 コマンドファイル

Ch 言語環境では、C プログラムをコンパイルすることなく実行できます。Ch のコマンドライン引数インタフェースは、C と互換性があります。Ch では、C プログラムは、**コマンドファイル**または単に**コマンド**と呼ばれます。

ファイルが Ch プログラムとして識別されるのは、ファイルに読み取り/実行アクセス許可が与えられていて、かつ次のいずれかのトークンで始まる場合です。

1. コメントシンボル/*または//
2. 型指定子、型修飾子、またはストレージクラス指定子
3. シンボル#と、それに続くプリプロセッサディレクティブ
4. シンボル#と、それに続く!/bin/ch または!/bin/sch
5. 識別子 main
6. 関数名 printf

7. Ch シェルプロンプトに入力した場合、現在のシェルでプログラムを実行するために使用されるドット"."

Ch プログラミング環境では、コマンドファイルをコンパイルすることなく実行できます。string 型のシステム変数 **_pathext** は、コマンドのファイル拡張子を格納します。変数 **_pathext** の既定値は、Unix では `""`、Windows では `".com;.exe;.bat;.cmd"` です。

ファイル拡張子 `.ch` を明示的に入力しないで `prog.ch` などの Ch コマンドを呼び出すには、ユーザーのホームディレクトリにあるスタートアップファイル **.chrc** (Unix の場合) または **_chrc** (Windows の場合) で、システム変数 **_pathext** にファイル拡張子 `.ch` を追加します。

Ch プログラムには、ユーザーがそのプログラムを実行できるように、読み取りと実行の両方のアクセス許可が与えられなければなりません。プログラムのアクセス許可は、コマンド `chmod` で変更できます。次にコマンドの例を示します。

```
chmod 755 program.ch
```

このコマンドは、プログラム `program.ch` に対するプログラム所有者の権限を読み取り/書き込み/実行、グループおよびその他のユーザーの権限を読み取り/実行に変更します。

コマンド名の前に相対パスまたは絶対パスを指定した場合は、指定したパスでその名前が検索されます。それ以外の場合は、システム変数 **_path** に指定したパスが順に検索されます。**_path** の既定値は表2.2に記載されています。汎用関数 **stradd()** を使用して、**_path** にパスを追加できます。たとえば、次に示すコマンドは、**_path** の最後にパス `/home/mydir/bin` を追加します。

```
> _path = stradd(_path, "/home/mydir/bin;")
/usr/ch/bin/;/usr/ch/sbin;/usr/ch/toolkit/bin;
/usr/ch/toolkit/sbin;/bin;/usr/bin;/sbin;/home/mydir/bin;
>
```

Ch を起動するたびにこのパスを自動的に追加するには、次のコマンド

```
_path = stradd(_path, "/home/mydir/bin;")
```

を、ユーザーのホームディレクトリにある **.chrc** (Unix の場合) や **_chrc** (Window の場合) などの スタートアップファイルに追加します。スタートアップファイルをカスタマイズする方法の詳細については、セクション3.2を参照してください。

Unix および Windows の両方で、システム変数 **_path** のパス名には、`C:/Program Files/package` のように空白を使用できます。Windows では、ファイル拡張子が `.dll` である動的に読み込まれるライブラリのパスも、**_path** に追加できます。

関数 **system()** は、ファイル拡張子が **system()** であるプログラムを、**_pathext** に指定されているプログラムと同様に処理できます。次に例を示します。

```
system("help.ch")
```

または

```
system("/usr/ch/bin/help.ch")
```

3.3.2　スクリプトファイル

Ch 言語環境では、他のシェルスクリプトとプログラムを認識できます。WWW CGI (Common Gateway Interface) などの他のシェルやプログラムによって認識されるためには、Ch プログラムは次の行で始まっている必要があります。

```
#!/bin/ch
```

その後に、`-S` (セーフシェル) や`-f` (高速起動) などのコマンドラインオプションを指定します。推奨はされていませんが、`#`記号の前と`!`記号の後ろにスペースを使用することができます。関数 **main()** を使用しておらず、かつ、`#!/bin/ch` から始まっているプログラムは、**スクリプト**と呼ばれます。スクリプトはコマンドと同様に扱われます。スクリプト内では、コマンドラインインタフェースに対してシステム変数 **_argc** と **_argv** を使用できます。これらの 2 つのコマンドラインインタフェース変数は、コマンドファイルに対しても使用できます。

3.3.3　関数ファイル

Ch プログラムは多数の個別ファイルに分割できます。各ファイルは、プログラムのあらゆる部分からアクセス可能な、最上位レベルにある多数の関連した関数で構成されます。複数の関数が記述されているファイルには、通常、Ch プログラムの一部として認識されるためのサフィックスである.ch が付加されます。

コマンドファイルとスクリプトファイルの他に、Ch には**関数ファイル**も存在します。 Ch の**関数ファイル**は、関数定義を使用して始まるプログラムです。関数ファイルは読み取り可能にする必要があります。関数ファイルの拡張子は、string 型のシステム変数 **_fpathext** によって指定します。

システム変数 **_fpathext** の既定値は`".chf"`です。関数ファイルの名前と、関数ファイル内の関数定義の名前は、同じにする必要があります。関数ファイルを使用して定義した関数は、Ch のシステム組み込み関数と同様に扱われます。たとえば、プログラム `addition.chf` に次のステートメントが含まれているとします。

```
/**** function file for adding two integers ****/
int addition(int a, int b) {
  int c;
  c = a + b;
  return c;
}
```

関数 `addition()` を自動的に呼び出して、2 つの整数を加算できます。多くのローカル関数をネストできる関数定義が一つだけ関数ファイルの中にあるようにすることをお勧めします。関数ファイルから関数 `addition()` を呼び出すプログラムのプロトタイプは次のように記述できます。

```
extern  int addition(int a, int b);
```

プログラムでは、関数ファイルから呼び出す関数のためのこのプロトタイプは、省略可能です。

第5章で説明するプリプロセッサディレクティブ`#endif` は、関数ファイル内で関数の引数の右かっこより後ろに置くことはできません。たとえば、次のコードは無効です。

```
int fun(int arg1,
#ifdef NeedWidePrototypes
int arg2,
double arg3) {
#else
char arg2,
float arg3) {
#endif
     /* ... */
}
```

代わりに、次のように記述します。

```
int fun(int arg1,
#ifdef NeedWidePrototypes
int arg2,
double arg3
#else
char arg2,
float arg3
#endif
) {
     /* ... */
}
```

関数ファイル内で関数より前にあるインクルードファイルは、関数定義が解析される前に処理されます。たとえば、次のコードは有効です。

```
#include<stdio.h>
FILE *fopen(const char *filename, const char *type) {
  return _fopen(filename, type);
}
```

関数ファイルをプログラムの関数プロトタイプとして使用した場合、関数定義より前にあるプリプロセッサディレクティブは無視されます。そのようなディレクティブは、プログラムの最後に関数が処理されるときに解析されます。関数ファイル内の関数をコマンドモードのプロンプトで呼び出した場合、`#include` を除くすべてのディレクティブが処理され、インクルードされるヘッダーファイルは、関数ファイル内の関数プロトタイプが使用される前に解析されます。したがって、関数ファイル内の条件付きプリプロセッサディレクティブは、関数がプログラム内で使用される場合に限り、関数定義の前でも有効です。次に示す関数ファイルに定義された関数 `func()` は、プログラムでは使用できますが、コマンドモードのプロンプトでは使用できません。

```
#ifdef HEADER1
#include<header1.h>
```

```
#else
#include<header2.h>
#endif
int func() {
  ...
}
```

コードを次のように変更すると、プログラムおよびコマンドモードの両方でこの関数を使用できます。

```
#include<header.h> // include hearer1.h and hearder2.h conditionally
int func() {
  ...
}
```

Ch では、システム変数 **_fpath** に指定されたパスを順に検索して、関数ファイルを検索します。**_fpath** の既定値は表2.2に記載されています。関数ファイルの追加パスをシステム変数 **_fpath** に追加できます。たとえば、次のコマンドは、**_fpath** の最後にパス `/home/mydir/lib` を追加します。

```
> _fpath = stradd(_fpath, "/home/mydir/lib;")
/usr/ch/lib/libc;/usr/ch/lib/libch;/usr/ch/lib/libopt;
/usr/ch/lib/libch/numeric;/home/mydir/lib;
>
```

システム変数 **_fpath** をコマンドモードで変更した場合、その変更は、現在のシェルで対話的に呼び出す関数に対してのみ有効になります。現在のシェルでの関数検索パスは、サブシェルでは使用されず、継承もされません。このパスの関数ファイルを現在の Ch シェルとすべての Ch プログラムで使用できるようにするには、次のコマンド

```
_fpath = stradd(_fpath, "/home/mydir/lib;")
```

をユーザーのホームディレクトリのスタートアップファイル **_chrc** (Windows の場合) または **.chrc** (Unix の場合) に追加します。関数ファイルの検索パスが正しく設定されていない場合は、関数 `addition()` を呼び出したとき、次のような警告メッセージが表示されます。

```
WARNING: function 'addition()' not defined
```

これは、関数 `addition()` を呼び出したときのメッセージです。

関数をコマンドモードで呼び出すと、関数ファイルが読み込まれます。関数を呼び出した後に関数ファイルを変更すると、コマンドモードでの以降の呼び出しには、読み込み済みの古いバージョンの関数定義が依然として使用されます。変更後のバージョンである新しい関数ファイルを呼び出すには、次のコマンドでシステム内の関数定義 (たとえば `addition`) を削除します。

```
> remvar addition
```

または、新しい Ch シェルを起動します。

.chf ファイルには、複数の関数定義とクラス定義を記述できます。複数の関数定義とクラス定義が記述された.chf ファイルは、関数ファイルとして扱うのではなく、**pragma** ディレクティブを使用して明示的に読み込んでください。たとえば、次のコードがあるとします。

```
#pragma importf <myfunc.chf>
#pragma importf <myclass.chf>
```

このコードは、システム変数 **_fpath** で指定したディレクトリにある、複数の関数定義とクラス定義が記述された `myfunc.chf` ファイルと `myclass.chf` ファイルを読み込みます。この **pragma** ディレクティブは、アプリケーションに通常組み込まれるヘッダーファイル内に記述できます。

3.4 プログラムの実行

プログラムの起動は、指定した Ch プログラムが実行環境によって呼び出されたときに発生します。プログラムは、まず解析され、内部的なデータ構造が作成された後に実行されます。プログラム実行前に、静的な記憶域のすべてのオブジェクトが初期化されます (初期値が設定されます)。プログラムが終了すると、実行環境に制御が戻ります。

プログラムが終了すると、実行環境に制御が戻ります。

3.4.1 コマンドモードでのプログラミングステートメントの実行

Ch シェルプロンプトでは、すべての式、プログラミングステートメント、および関数が直ちに解析されて実行されます。次に例を示します。

```
> int i
> for (i = 0; i < 3; i++) printf("i = %d\n", i)
i = 0
i = 1
i = 2
> int func1(int i){int j; j = i+i; return j;}
> i = func1(10)
20
> int func2(int i){int j; j = i*i;\
return j;}
> i = func2(i)+func1(1)
402
> 2*i
804
>
```

上記の例では、for ループと関数 `func1()` および関数 `func2()` の定義をシェルプロンプトで入力します。末尾のセミコロンは、シェルプロンプトでは必要ありません。すべてのプログラミングス

テートメントは、1 行のコマンドラインで指定する必要があります。1 行のコマンドラインは、関数 `func2()` の定義に示すように、行継続記号 '\' とそれに続く改行文字で区切った複数の行で構成できます。このように指定しないと、Ch はエラーメッセージを発行します。たとえば、上記の例で for ループが 2 行に分割されていると、正しい結果になりません。関数の定義が複数の行に分割されている場合、Ch では構文エラーとして扱われます。

```
> int i
> for (i = 0; i < 3; i++)    // break the for-loop into two lines
> printf("i = %d", i)        // and the result is unexpected
i = 3
> int fun1(int i){int j;     // the definition of fun1() is broken
ERROR: missing '}'
WARNING: missing return statement for function fun1() and
default zero is used
>
```

3.4.2 プログラムの起動

通常、Ch プログラムは以下の順序で実行されます。最初に、スタートアップファイル **CHHOME/config/chrc** が実行されます。スタートアップファイル内のグローバル変数とシステム変数のすべての値は、実行される現在のプログラムで使用するために保持されます。次に、いわゆるプリプロセッサディレクティブのすべてのモジュールを含むプログラムが解析され、内部プログラムツリーが作成されます。次に、内部プログラムツリーの個々の実行可能ステートメントが実行されます。最後に、関数 **main()** または **WinMain()** が宣言されていれば実行されます。

関数 main() は、int 型の戻り値を使用して、以下のいずれかの形式で定義します。パラメータがない場合は、次のとおりです。

```
int  main(void) { /* ... */ }
```

または、2 つのパラメータ (ここでは `argc` と `argv`) を指定する場合は、次のようになります。ただし、パラメータ名には任意の名前を使用できます (パラメータは、それを宣言する関数に対してローカルであるため)。

```
int main(int argc, char *argv[]) { /* ... */ }
```

または

```
int main(int argc, char **argv[]) { /* ... */ }
```

3 つのパラメータを指定する場合は、次のようになります。

```
int main(int argc, char *argv[], char **environ) { /* ... */ }
```

Windows では、関数 **WinMain()** を Windows API に従って次のように定義します。

```
int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
{ /* ... */}
```

関数 **main()** のパラメータを宣言する場合は、次の制約に従います。

- argc の値が負ではありません。
- argv[argc] は null ポインタです。
- argc の値がゼロより大きい場合、配列番号 argv[0]〜argv[argc-1] には (両端の番号を含む)、文字列へのポインタが含まれます。
- argc の値がゼロより大きい場合、argv[0] がポイントする文字列はプログラム名を表します。argc の値が 1 より大きい場合、argv[1]〜argv[argc-1] がポイントする文字列は、プログラムのパラメータを表します。
- パラメータ argc および argv と argv 配列がポイントする文字列は、プログラムによって変更可能であり、プログラムの起動から終了までの間、最後に格納した値を保持します。
- パラメータ environ は、環境変数テーブルへのポインタです。

これらの関数の詳細については、セクション10.10を参照してください。システム変数 **_argc** および **_argv** に適用される制約と値は、それぞれパラメータ argc および argv と同じです。2 つのシステム変数 **_argc** と **_argv** の詳細については、セクション4.16を参照してください。

3.4.3 プログラムの終了

関数 **main()** の戻り値の型は、int 型と互換性のある型にする必要があります。関数 **main()** の初回呼び出しから戻ることは、関数 **main()** から返された値を引数にして **exit** 関数を呼び出すことと同等です。ステータス値は、システム変数 **_status** に格納されます。

3.4.4 検索順序

プログラムでの順序

Ch 言語環境では、与えられた識別子を次の検索順序に従って解釈します。

- 定義済みのマクロかどうかをチェックします。
- キーワードかどうかをチェックします。
- 関数の変数を含めて定義済みの変数かどうかをチェックします。
- 左かっこ'(' が後に続いているかどうかをチェックします。左かっこが後に続いている場合は、システム変数 **_fpathext** 内の拡張子リストにあるファイル拡張子を名前に付加します。ファイル拡張子ごとに、関数ファイルが見つかるまで、システム変数 **_fpath** に指定された各ディレクトリを検索します。

- システム変数**_path** に指定されたディレクトリ内のコマンドかどうかをチェックします。次に、システム変数**_pathext** 内の拡張子リストにあるファイル拡張子を名前に付加します。ファイル拡張子ごとに、実行可能で読み取り可能なコマンドが見つかるまで、システム変数**_path** に指定された各ディレクトリを検索します。

プロンプトでの順序

対話型モードでプロンプトに対して識別子を指定した場合、別名リストとの照合テストが最初に行われます。その後は、前のセクションで説明した検索順序に従います。

第 4章で説明する Ch プログラム `which` では、与えられた識別子がどのように解釈されるかを示します。

3.4.5 複数のファイルを使用するプログラムの実行

このセクションでは、複数のファイルを使用したプログラム実行について説明します。Ch でのパッケージの処理については、セクション22.3で説明します。

Ch プログラムのファイル名はコマンド名です。コマンドの拡張子は、システム変数**_pathext** で指定できます。複数のファイルから成るプログラムは、`import`、`importf`、 およびセクション5.9で説明するプリプロセッサディレクティブ `pragma` を使用して編成できます。

インクルードするヘッダーファイルがシステム変数**_ipath** に指定したディレクトリで検索されるのとは異なり、`import` および `importf` の後に続くプログラムは、システム変数**_path** および**_fpath** に指定したディレクトリでそれぞれ検索されます。

さらに、`import` と `importf` の後に続けて文字列を記述できます。この場合、ファイルは最初に現在のディレクトリで検索されます。文字列式がポイントするファイルが存在しない場合、pragma ステートメントは無視されます。たとえば、`command` というコマンドが別々の 4 つのファイル (`command.c`、`module1.c`、`module2.c`、および `module3.c`) から成る場合、プログラム `command.c` は次のように記述できます。

```
/* Program command.c */
#include <stdio.h>
int main() {
  int i =90;
  printf("main() program \n");
  /* ... main program goes here */
}
#pragma importf "module1.c" /* search for module1.c in current
            directory first then directories specified in _fpath */
#pragma import "module2.c"  /* search for module2.c in current
            directory first then directories specified in _path */
#pragma importf <module3.c> /* search for module3.c in directories
                                        specified in _fpath only */
```

module1.c、module2.c、および module3.c ファイル内の静的変数は、ファイルスコープをもっています。この例の import と importf の違いに注意してください。module1.c ファイルは、最初に現在の作業ディレクトリで検索されてから、_**fpath** に指定したディレクトリで検索されます。module2.c ファイルは、最初に現在の作業ディレクトリで検索されてから、_**fpath** ではなく _**path** に指定したディレクトリで検索されます。module3.c ファイルは、_**fpath** に指定したディレクトリでのみ検索されます。コマンドファイル command.c には、読み取り/実行アクセス許可が必要です。module1.c、module2.c、および module3.c の各ファイルには、読み取りアクセス許可が必要です。

別の方法として、元の C コードのファイル command.c、module1.c、 module2.c、および module3.c はそのままにして、command.ch という名前の Ch コマンドを追加することもできます。

```
#!/bin/ch
/* command.ch  */
#pragma import  "command.c"
#pragma importf "module1.c"
#pragma import  "module2.c"
#pragma importf <module3.c>
```

複数のファイルを使用するコマンドの場合は、ディレクトリを作成し、コマンドで使用する他のファイルをそのディレクトリに保存することをお勧めします。たとえば、コマンド xxx の場合は、コマンド xxx によって呼び出されるファイルを保存するためのディレクトリ xxx_ch を作成します。したがって、command というコマンドの場合は、システム変数 _**path** 内の検索パスに親ディレクトリが含まれる command_ch ディレクトリを作成できます。このコマンドは、次のように記述できます。

```
/* Program command.c */
#include <stdio.h>
int main() {
  int i =90;
  printf("main() program \n");
  /* ... main program goes here */
}
#pragma import <command_ch/module1.c> /* search for module1.c in
                             directories specified in _path only */
#pragma import <command_ch/module2.c>
#pragma import <command_ch/module3.c>
```

これで、command ファイルを実行可能な Ch コマンドとして使用できます。

pragma によってインクルードされるファイル内の静的変数はファイルスコープをもっています。多くの場合、これは問題なく機能します。ただし、C プログラムでは、別のモジュールによってインクルードされるヘッダーファイル内に静的変数が宣言される場合があります。各モジュールは、別々にコンパイルされます。これは、静的変数が、対応する Ch プログラム内のすべてのモジュールによってアクセスされる必要があることを意味します。このような場合は、include ディレクティブを使用できます。前述のプログラム例は、次のように記述できます。

```
/* Program command.c */
#include <stdio.h>
int main() {
  int i =90;
  printf("main() program \n");
  /* ... main program goes here */
}
#ifdef _CH_
#include "module1.c"  /* search for module1.c in current directory
                   first then directories specified in _ipath */
#include "module2.c"  /* search for module2.c in current directory
                   first then directories specified in _ipath */
#include <module3.c>  /* search for module3.c in directories
                         specified in _ipath only */
#endif
```

同様に、元の C コードのファイル command.c、module1.c、module2.c、および module3.c はそのままにして、command.ch という名前の Ch コマンドを追加できます。

```
#!/bin/ch
/* command.ch  */
#include "module1.c"
#include "module2.c"
#include <module3.c>
```

複数のファイルから成るプログラムは、現在のシェルで実行されるドットコマンドを使用して編成することもできます。インクルードされるヘッダーファイルとは異なり、ドットの後に続くプログラムが、システム変数 **_path** に指定したディレクトリで検索されます。プリプロセッサディレクティブ include を使用してファイルをインクルードするのと同様に、ドットコマンドの静的変数は、プログラムのすべてのモジュールから参照できます。ドットコマンドを使用すると、前述のサンプルプログラム command.c を次のように記述できます。

```
/* Program command.c */
#include <stdio.h>
#ifdef _CH_
. "module1.c"  /* search for module1.c in current directory first
                  then directories specified in _ipath */
. "module2.c"  /* search for module2.c in current directory first
                  then directories specified in _ipath */
. <module3.c>  /* search for module3.c in directories
                  specified in _ipath only */
#endif
int main() {
```

```
  int i =90;
  printf("main() program \n");
  /* ... main program goes here */
}
```

これらのファイルがすべて 1 つのディレクトリ (たとえば/my/package/dir) に置かれている場合、コマンド *command* を別のディレクトリで実行できます。そのためには、このプログラムの次の行を変更します。

```
#ifdef _CH_
```

これを次のように変更します。

```
#ifdef _CH_ && strcat(_ipath,"/my/package/dir;") \
            && strcat(_path,"/my/package/dir;")
```

同様に、元の C コードのファイル command.c、command.c、module1.c、および module2.c はそのままにして、command.ch という名前の Ch コマンドを追加できます。

```
#!/bin/ch
/* command.ch  */
. "module1.c"
. "module2.c"
. <module3.c>
```

第 22章で、Ch で実行するライブラリとソフトウェアパッケージを作成する方法の詳細について説明します。

3.4.6 プログラムのデバッグ

プログラムの構文エラーをチェックするために、Ch プログラムを実行しないで解析する場合は、ファイル名を指定したシェルコマンド **chparse** を使用できます。解析後は、シェルコマンド **chrun** を入力してプログラムを実行できます。
次に例を示します。

```
> chparse program.c
> chrun
>
```

次のようなプログラム hello.c があるとします。

```
int main() {
  printf("hello, world\n";
}
```

次のように、コマンド chparse でプログラムのエラーを診断できます。

```
> chparse hello.c
ERROR: missing )
ERROR: Syntax error at line 2
>
```

2 行目の関数 printf() にかっこが抜けていることが検出されます。

最初にプログラム全体を解析します。次に、シェルコマンド **chdebug** を使用して、プログラムをステップ単位で対話的に実行できます。次の例では、最初に `more` コマンドで `program.c` を表示します。その後、**chdebug** コマンドで実行します。

```
> more program.c
int main() {
   int i, *p;
   i = 10;
   p = &i;
}
> chdebug program.c

You are debugging file 'program.c'

Type     (1) expression for evaluation
         (2) 'run' to continue
         (3) hit return key to step to next line
         1:      int main() {

         2:          int i, *p;

         3:          i = 10;

         4:          p = &i;
i
=> 10
i*i
=> 100
&i
=> 1c21a0

         5:      }
p
=> 1c21a0
*p
=> 10
```

```
        1:      int main() {

>
```

デバッグモードでは、3 つのオプションがあります。式の評価、プログラムの連続実行、またはプログラムのステップ実行です。ステップ実行では、実行する前に行番号を含むソースコードを表示します。評価する式を入力すると、シンボル=>の後に式の結果が表示されます。この例では、`i*i` が 100 であり、変数 `i` のアドレスがポインタ `p` の値と同じであることが示されています。ポインタの詳細については第9章で説明します。

コマンド **chparse-chrun** および **chdebug** を使用すると、プログラムは現在のシェルで実行されます。次に示すように、`-n` オプションを使用すると、構文エラーをチェックする目的のためだけに、プログラムを実行しないで解析することができます。

```
> ch -n program.c
>
```

この場合、プログラムはサブシェルで実行されます。

ヘッダーファイル **assert.h** に定義されているマクロ `assert()` も、プログラムをデバッグするために使用できます。プログラムの内部にデバッグ関数`_stop("デバッグ用メッセージ\n")` を追加することによって、プログラムにブレークポイントを設定できます。実行段階では、プログラムはこのステートメントで停止し、ユーザーの入力を待ちます。

プログラムが停止したポイントでの変数または式の名前を入力すると、その変数または式の値を表示できます。また、デバッグモードでプログラムを実行する場合、実行時に暗黙のポインタである'this'を使用して、クラスのメンバ関数内でクラスのメンバにアクセスできます。

Windows では `stderr` ストリームの扱いが異なるため、Ch プログラムをデバッグするにはコマンドラインオプション`-r` を使用する必要があります。次にコマンドの例を示します。

```
ch -r program.c > junkfile
```

このコマンドは、`stderr` ストリームから `junkfile.` にエラーメッセージを送ります。file `junkfile.`

3.5 スコープの規則

3.5.1 識別子のスコープ

識別子は、1) オブジェクト、2) 関数、3) タグあるいはクラス、構造体、共用体、または列挙体のメンバ、4) typedef 名、5) ラベル名、6) マクロ名、7) マクロのパラメータのいずれかを表します。プログラムの別々の箇所にある異なる複数のエンティティを同一の識別子で表すことができます。列挙体のメンバは列挙定数と呼ばれます。

ソースファイル内のマクロ名は、プリプロセッサトークンのシーケンスに置換されます。そのシーケンスが受け渡しフェーズでのマクロ定義になります。

識別子が表す異なるそれぞれのエンティティから、識別子はスコープと呼ばれるプログラムテキストの領域内でのみ参照可能 (つまり使用可能) です。同じ識別子で表される別々のエンティティは、異なるスコープを持っているか、異なる名前空間に属しています。スコープの種類として、関数、ファ

イル、ブロック、関数プロトタイプ、プログラム、システムがあります。関数プロトタイプは、パラメータの種類を含む関数の宣言です。

ラベル名は、関数スコープをもつ唯一の識別子です。ラベル名はそれが記述されている関数のどこででも **goto** 文の中で使用できます。構文で:とステートメントを続けて記述することによって、暗黙的に宣言します。

他のどの識別子にも、(宣言子または型指定子での) 宣言の位置によって決定されるスコープがあります。識別子を宣言する宣言子または型指定子が、ブロックの外に出現する場合は、識別子はプログラムスコープをもちます。識別子が、記憶クラスの修飾子 **static** を使用してブロックの外側で静的変数として宣言されている場合、識別子はファイルスコープをもちます。識別子が`__declspec(global)`で宣言されている場合、識別子は現在の Ch シェルのシステムスコープをもちます。システムスコープの識別子は、複数のプログラムからアクセスできます。識別子を宣言する宣言子または型指定子が、ブロックの内側または関数定義のパラメータ宣言リスト内に出現する場合、識別子はブロックスコープをもちます。このスコープは、関連するブロックの終わりで終了します。識別子を宣言する宣言子または型指定子が、関数プロトタイプ (関数定義の一部ではない) のパラメータ宣言リスト内に出現する場合、識別子は関数プロトタイプスコープをもちます。このスコープは、関数宣言子の終わりで終了します。識別子が、同じ名前空間にある 2 つの異なるエンティティを表す場合、スコープが重なる可能性があります。その場合、一方のエンティティのスコープ (内側のスコープ) は、他方のエンティティのスコープ (外側のスコープ) の厳密なサブセットになります。内側のスコープ内では、識別子は内側のスコープで宣言されたエンティティを表します。外側のスコープで宣言されたエンティティは、内側のスコープ内では隠されます (参照不可です)。2 つの識別子のスコープが同じになるのは、スコープが同じ場所で終了する場合、かつその場合だけです。

別途明記しない限り、このドキュメントでエンティティ(構文的な構成ではなく) を表すために識別子という用語を使用する場合は、関連する名前空間のエンティティを意味します。そのエンティティの宣言は、識別子が記述されている位置において参照可能です。

クラス、構造体、共用体、および列挙体のタグは、タグを宣言する型指定子に記述したタグの直後から始まるスコープを持ちます。各列挙定数は、列挙子リストに記述した、定数を定義する列挙子の直後から始まるスコープを持ちます。その他すべての識別子は、宣言子が完了した直後から始まるスコープを持ちます。

3.5.2 識別子のリンケージ

異なる複数のスコープで宣言した識別子、または同じスコープで複数回宣言した識別子は、リンケージと呼ばれるプロセスによって、同じオブジェクトまたは関数を参照できます。リンケージには 4 種類あります (グローバル、外部、内部、リンケージなし)。

プログラム全体を構成するソースファイルのセットにおいて、外部リンケージを使用して特定の識別子を宣言すると、各宣言は同じオブジェクトまたは関数を表します。ソースファイル内で内部リンケージを使用して識別子を宣言すると、各宣言は同じオブジェクトまたは関数を表します。リンケージのない識別子の各宣言は、一意なエンティティを表します。

オブジェクトまたは関数のファイルスコープ識別子の宣言に`__declspec(global)`が含まれている場合、識別子はグローバルなリンケージを持ちます。

オブジェクトまたは関数のファイルスコープ識別子の宣言に記憶クラス指定子 **static** が含まれている場合、識別子は内部リンケージを持ちます。

識別子の以前の宣言が参照できるスコープで、記憶クラス指定子 **extern** を使用して宣言した識別子の場合、先行する宣言で内部または外部リンケージを指定していると、後の宣言での識別子のリンケージは、先行する宣言で指定したリンケージと同じになります。先行する宣言が参照不可である場合、または先行する宣言でリンケージを指定していない場合は、識別子は外部リンケージを持ちます。

関数の識別子の宣言に記憶クラス指定子がない場合、リンケージは、記憶クラス指定子 **extern** を使用して宣言した場合とまったく同様に決定されます。オブジェクトの識別子の宣言がファイルスコープを持ち、記憶クラス指定子がない場合、リンケージは外部です。

次の識別子にはリンケージがありません。つまり、関数のパラメータとして宣言された識別子と、記憶クラス指定子 **extern** なしで宣言されたオブジェクトのブロックスコープ識別子です。

内部および外部の両方のリンケージを持つ同じ識別子が記述されている場合は、構文エラーです。

3.5.3 識別子の名前空間

プログラムの任意の箇所において特定の識別子の複数の宣言が参照できる場合は、構文のコンテキストによって、異なるエンティティを参照する使用法のあいまいさを解決します。さまざまな識別子に応じて分類された個別の名前空間を次に示します。

- *macro names* マクロ名 (プリプロセッサディレクティブ`#define` によって定義されたマクロ)
- *label names* ラベル名 (ラベル宣言と使用法の構文によってあいまいさを解決)
- クラス、構造体、共用体、および列挙体のタグ (**class**、**struct**、**union**、または **enum** キーワードのいずれかの後に続けてあいまいさを解決)
- クラス、構造体または共用体のメンバ。各クラス、構造体または共用体にはメンバのために個別の名前空間があります (**.** または–>演算子でのメンバへのアクセスに使用する式の種類によってあいまいさを解決)。
- 通常の識別子と呼ばれる、その他すべての識別子 (通常の宣言子で宣言、または列挙定数として宣言)

3.5.4 オブジェクトの記憶期間

有効な記憶クラス指定子を表3.3に示します。

表 3.3: 記憶クラス指定子

指定子	機能
auto	ローカル自動変数
extern	外部変数
__declspec(global)	システム全体のグローバル変数
__declspec(local)	入れ子のローカル関数
register	(無視される)
static	静的変数

オブジェクトには記憶期間があり、それによって存続期間が決定します。静的、自動、および割り当てという 3 種類の記憶期間があります。

`__declspec(global)` で修飾した変数は、現在のシェルでドットコマンドを使用してさまざまなプログラムを実行する場合に、複数のプログラムで通用します。

`__declspec(global)` で修飾された変数は、一度だけ宣言し、現在のシェルで複数のプログラムによって使用します。このようなグローバル変数は、システムスタートアップファイル `chrc` またはユーザーのスタートアップファイル `_chrc`(Windows) または `.chrc`(Unix) で通常宣言されます。関数とクラス/構造体/共用体の変数はグローバル変数として宣言できません。

識別子が外部または内部リンケージを使用して宣言されているか、記憶クラス指定子 **static** によって宣言されているオブジェクトは、静的な記憶期間を持っています。そのようなオブジェクトに関しては、記憶域はプログラムの実行中つねに確保されており、格納された値はプログラム開始処理の前に一度だけ初期化されます。プログラム全体が実行される間、オブジェクトは存在し、一定のアドレスを持ち、最後に格納された値を保持します。

識別子がリンケージなし、記憶クラス指定子 **static** なしで宣言されたオブジェクトは、自動記憶期間を持ちます。可変長の配列の型を持たないこのようなオブジェクトでは、関連付けられたブロックに入るたびに、オブジェクトの新しいインスタンス用に記憶域が確保されます。オブジェクトの初期値はゼロです。オブジェクトに対して初期化が指定されている場合、ブロックの実行で宣言に達するたびに初期化が実行されます。そうでない場合、宣言に達するたびに値は不定になります。ブロックの実行がなんらかの方法で終了した時点で、オブジェクト用の記憶域は解放されます。(囲まれたブロックに入るか、関数が呼び出されると、現在のブロックの実行は中断しますが終了はしません。)

可変長の配列の型を持つオブジェクトについては、プログラムの実行で宣言に達するたびに、オブジェクトの新しいインスタンス用に記憶域が確保されます。オブジェクトの初期値はゼロです。プログラムの実行が宣言のスコープを出ると、オブジェクト用の記憶域の確保は保証されません。

記憶域が確保されていないオブジェクトが参照された場合、動作は未定義です。記憶域が確保されないオブジェクトを参照するポインタの値は、不定です。記憶域が確保されている間、オブジェクトは一定のアドレスを持ちます。

記憶域は、関数 **calloc()**、**malloc()**、および **realloc()** によって実行時に動的に**割り当て**、後で関数 **free()** によって解放できます。また、演算子 **new** と **delete** によって、それぞれ記憶域を動的に割り当ておよび割り当て解除することができます。メモリ割り当てとポインタの詳細については、第9章で説明します。

第4章　移植可能な対話型コマンドシェルとシェルプログラミング

本章では、コマンドモードでコマンドシェルとして対話的に Ch を使用する方法を説明します。他のシェルと同様に、Ch シェルはユーザーがプロンプトに対して入力したコマンドラインを読み取り、実行する内容を解釈するコマンドインタープリタです。Ch では対話型シェルおよびシェルプログラムの両方で、すべての演算子と関数のみならず、ほとんどのコマンドを使用可能です。演算子と関数の詳細については、それぞれ第7章と第10章を参照してください。セマンティクスの観点からは、Ch シェルは C シェルに類似しています。Ch は C のスーパーセットですが、いわゆる C シェルは C とはまったく異なります。付録 Dに、C シェルと Ch の構文の一部を比較した一覧を記載しています。

4.1　シェルプロンプト

シェルごとに独自のシェルプロンプトがあります。既定では、通常の Ch シェルのプロンプトは ‘`cwd>` ’です。ここで、`cwd` は現在の作業ディレクトリを表します。これは、通常の Ch シェルがコマンドラインからの入力を処理する準備ができていることをユーザーに示します。既定では、セーフ Ch のシェルプロンプトは ‘`safech>` ’です。セーフ Ch の詳細については、第 21章を参照してください。Unix のスーパーユーザーまたは Windows の Administrator ユーザーの場合、通常の Ch シェルおよびセーフ Ch シェルのシェルプロンプトは、それぞれ、‘`#`’ と ‘`safech#`’ です。

表4.1で Ch と他の一般的なシェルについて既定のシェルプロンプトを比較しています。

表 4.1: シェルプロンプトの比較

シェル	一般ユーザーのプロンプト	スーパーユーザーのプロンプト
Ch シェル (Windows)	>	#
Ch シェル (Unix)	>	#
セーフ Ch シェル (Windows)	safech>	safech#
セーフ Ch シェル (Unix)	safech>	safech#
C シェル	%	#
Bourne、Korn、BASH の各シェルプロンプト	$	#

Ch のシェルプロンプトの既定のシンボルを変えたり、ホスト名や現在の作業ディレクトリなどの情報をシェルプロンプトに追加したりすることができます。そのためには、システム変数 **_prompt** を設定します。たとえば、対話型のコマンドシェルでは、次のように設定します。

```
> _prompt = "$ "
$
$ _prompt = "% "
%
% _prompt = stradd(_cwd,"> ")
/usr/ch>
```

最初に '$' および '%' のシンボルを Ch シェルプロンプトに設定し、次に関数 **stradd()** を呼び出して、現在の作業ディレクトリの末尾に '>' というシンボルを付けるように設定します。この例では、現在の作業ディレクトリは/usr/ch です。**_prompt** の値を設定することによって、任意の文字をシェルプロンプトとして選択できます。通常、システム変数 **_prompt** は、ユーザーのホームディレクトリにあるスタートアップファイル **.chrc** (Unix) または **_chrc** (Windows) で設定します。

4.2 コマンドの対話的な実行

Ch シェルのコマンドラインモードコマンドとは、コンパイル済みの実行可能なバイナリファイル、シェルスクリプト、C および Ch のプログラムなどのことです。次に例を示します。

```
> pwd
/home/myname
> mkdir subdir1
> cd subdir1
> pwd
/home/myname/subdir1
> which ls
ls is aliased to ls -F
>
```

この例では、プログラム **pwd** で現在の作業ディレクトリ`/home/myname` を表示しました。次にコマンド **mkdir** で新しいディレクトリ `subdir1` を作成します。さらに組み込みコマンド cd で現在のディレクトリを変更します。**cd** シェルプログラム **which** は **ls** が別名であることを示します (別名についてはセクション4.6で説明します)。

コマンドモードでコマンドファイルを実行するには、ファイル名が Ch の有効な識別子であるか、または '`./`'、'`../`'、'`~/`'、'`../`' などの相対または絶対ディレクトリパスから始まっている必要があります。たとえば、`20` や 20.e1 などの数値は有効な識別子ではありません。コマンドを二重引用符で囲むことができます。コマンドのオプションを引用符内に指定することはできません。引用符を使用すると、プログラム内でコマンドと変数名の競合を避けることができます。また、コマンドが空白を含む名前のディレクトリ内にある場合にも使用できます。次に例を示します。

```
> int ls = 10
> ls*2
20
>  "ls" -1
```

```
(display files in the current directory in a single column)
> "C:/Program Files/Windows NT/Accessories/wordpad.exe"
(launch wordpad program)
```

コマンド間をセミコロンで区切ると、同一のコマンドラインに複数のコマンドを入力できます。複合コマンドの例を次に示します。

```
> cp filename1 filename2; vi filename2
>
```

このコマンドは、filename1 というファイルを filename2 というファイルにコピーし、次いでコマンド vi を呼び出して filename2 ファイルを編集します。

4.2.1 現在のシェル

Ch の現在のシェルでプログラムを実行するための原則と構文は、sh、bash、および ksh の各シェルの原則および構文と同じです。既定では、Ch シェルではプログラムをサブシェルで実行します。組み込みドットコマンド

```
. filename
```

は、サブシェルではなく現在のシェルでプログラム *filename* を実行します。プロンプトにコマンドを入力すると、'.' を付けたかどうかにかかわらず、システム変数 **_path** に指定した検索パスが、コマンドを格納しているディレクトリの検索に使用されます。**cmd** というプログラムに次の 2 つのステートメントが指定されているとします。

```
int x = 3;
double y = 4;
```

以下の例では、このプログラムは最初にサブシェルで実行され、次に現在のシェルで実行されます。

```
> cmd    // run cmd in a subshell
> x      // print the value of variable x in current shell
ERROR: variable 'x' not defined
ERROR: command 'x' not found
> . cmd  // run cmd in the current shell
> x
3
> x*y
12.0000
> showvar
    x          3
    y          4.0000
```

プログラム cmd の最初の実行 (シンボル '.' の指定なし) は、サブシェル内で行われます。そのため、このプログラムが終了した後では、現在のシェルでまだ定義されていない変数 x は使用できません。プログラム cmd の 2 回目の実行 (シンボル '.' を指定) は、現在のシェルで行われます。プログラムが終了した後では、変数 x には値 3 が格納されています。この値は、現在のシェルで、プログラム cmd 内で x に代入されます。現在のシェルでは、変数をプロンプトで対話的に使用できるので、プロンプトで対話的に使用する複数の変数を 1 つのコマンドに指定して、ドットコマンドとして現在のシェルでそれを実行することも可能です。すべての変数と各変数の値は、シェルコマンド **showvar** を使用して表示できます。

シェルコマンド **stackvar** はすでに使われてなく、コマンド **showvar** に置き換えられました。

Ch では、システム変数 **_path** で指定したディレクトリ内で cmd などのコマンドが検索されます。プログラム cmd がシステム変数 **_path** で指定したいずれかのディレクトリにない場合、エラーメッセージが表示されます。現在のシェルで cmd が変数として使用されている場合は、次のように絶対パスまたは相対パスを先頭に付けることで、そのコマンドを使用できます。

```
> /dir1/dir2/cmd    // run cmd in the directory /dir1/dir2
> ./cmd             // run cmd in the current working directory
> ../cmd            // run cmd in the parent directory
> ~/cmd             // run cmd in the home directory
```

これらのコマンドではそれぞれ、ディレクトリ /dir1/dir2、作業ディレクトリ、親ディレクトリ、およびホームディレクトリにあるコマンドファイル cmd を実行します。

コマンドのパス名は、Unix および Windows の両方で'/' を使用して区切ることができます。ただし、Windows では区切り記号'\' も使用できます。たとえば、Windows では次のいずれかの形式で、プログラム notepad (メモ帳) を起動できます。

```
> notepad
> C:/Windows/notepad
> /Windows/notepad
> "/Windows/notepad"
> C:\Windows\notepad
> \Windows\notepad
```

4.2.2 バックグラウンドジョブ

Windows の MS-DOS コマンドシェルでは、Win32 プログラムはすべてバックグラウンドジョブとして実行されます。Ch では、Unix および Windows の両方においてコマンド処理の方法が一貫しています。コマンドは & メタ文字を使用してバックグラウンドで起動できます。そのため、シェルによる新しいコマンドの受け付けを妨げません。たとえば、次のコマンド

```
> notepad &
```

は、バックグラウンドでプログラム notepad を起動します。

4.3 プログラムステートメントの対話的な実行

前述したように、Ch シェルは、実行可能なバイナリファイルおよびシェルスクリプトだけでなく、C または Ch のプログラムをコンパイルなしで直接実行できます。C プログラムの対話的な実行は、コンパイル、リンク、実行、デバッグという長期サイクルを伴わず、アプリケーションの迅速な開発と配布のために特に有効です。たとえば、ファイル `hello.c` に次のステートメントが含まれているとします。

```
#include <stdio.h>
int main(void) {
  printf("Hello, world!\n");
  return 0;
}
```

このファイルは、コンパイルすることなく Ch シェルで次のように実行できます。

```
> hello.c    // execute hello.c program without compilation
Hello, world!
>
```

ソースファイルだけでなくプログラムステートメントも Ch シェルで直接かつ対話的に実行できます。対話型のコマンドラインモードでは、プログラムステートメントの末尾のセミコロンは必要ありません。次に例を示します。

```
> int i
> i = 10
10
> i * 2
20
> printf("i = %d", i)
i = 10
> printf("i in hexadecimal number = %x", i)
i in hexadecimal number = a
>
```

Ch は C の拡張機能もコマンドラインモードでサポートしています。例として、Ch Professional Edition および Student Edition の計算配列を次に示します。

```
> array int a[2][3] = {1, 2, 3, 4, 5, 6}
> a
1 2 3
4 5 6
> 2 * transpose(a)
2 8
```

```
4 10
6 12
>
```

ここで、a は2×3の計算配列であり、単一のオブジェクトとして処理されます。汎用関数**transpose()**は、一次元ベクトルまたは二次元行列の引数を転置して返します。計算配列は、工学および科学の分野の数値計算に便利です。計算配列の詳細については、第16章を参照してください。 ヘッダーファイル内のマクロや **typedef** による定義済みの型をシェルプロンプトで使用するために、セクション3.4.6で説明する **chparse** コマンドと **chrun** コマンドを使用して、ヘッダーファイルを読み込むことができます。たとえば、次のコマンドでは

```
> chparse /usr/local/ch/include/stdlib.h
> chrun
> size_t i
> i = 90
90
>
```

stdlib.h を変数 i の型宣言子として使用する前に、型 **size_t** を typedef で指定したヘッダーファイル **stdlib.h** を読み込みます。この場合、ヘッダーファイル **stdlib.h** は現在のシェルで実行されます。

シェルプロンプトに入力したステートメントが無効な場合は、デバッグのためのエラーメッセージが表示されます。

```
> blah
ERROR: variable 'blah' not defined
ERROR: command 'blah' not found
>
```

コマンドモードで関数を呼び出す場合、システム変数 **_fpath** に指定した検索パスが、関数定義を格納しているディレクトリの検索に使用されます。Ch シェルのコマンドプロンプトからプログラム内の関数を呼び出すには、まずそのプログラムを読み込む必要があります。コマンド **chparse** によってプログラムを読み込んだ後、セクション3.4.6で説明したように、コマンド **chrun** でプログラムを実行できます。同時に、プログラム内の関数もプロンプトで呼び出すことができます。一方では、最初にプログラムが現在のシェルで実行されてから、プログラム内の関数を対話的に使用できます。たとえば、以下はプログラム4.1のプログラム currentshell.cpp の対話的な実行です。

```
> . currentshell.cpp
func(5) = 10
15
> func(10)
20
> class tag c
> c.memfunc(10)
20
>
```

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

int func(int i) {
  return 2*i;
}

class tag {
  private:
    int m_i;
  public:
    tag();
    int memfunc(int);
};
tag::tag() {
  m_i=10;
}
int tag::memfunc(int i) {
  cout << m_i+i << endl;
  return m_i+i;
}

int main() {
   class tag c1;

   printf("func(5) = %d\n", func(5));
   c1.memfunc(5);
}
```

プログラム 4.1: 現在のシェルで実行した C++プログラム `currentshell.cpp`

ドットコマンド `. currentshell.cpp` は、現在のシェルで C++プログラム currentshell.cpp を読み込み、実行します。次の出力が生成されます。

```
func(5) = 10
15
```

コマンドプロンプトでコマンド `func(10)` を入力すると、現在のシェルで読み込まれたプログラム内の関数 `func()` が呼び出されます。宣言ステートメント `class tag c` は `class tag` のオブジェクト `c` をインスタンス化します。メンバ関数 tt tag::memfunc() tt tag::memfunc() が関数呼び出し `c.memfunc(10)` によって対話的に呼び出されると、値 20 が結果として表示されます。オブジェクトベースのプログラミングでクラスを使用する方法の詳細については、第19章で説明します。現在のシェルで実行しているプログラムがクラッシュした場合は、現在のシェルが終了することに注意してください。デバッグ目的の場合は、このようなプログラムを新たな Ch シェルで実行することをお勧めします。そうすることにより、現在のシェルが終了しても、バックグラウンドで実行中の Ch シェルを次のように引き続き使用できます。

```
> ch
> . currentshell.cpp
```

4.4 組み込みコマンド

Ch の組み込みコマンドを表4.2に示します。

表 4.2: Ch の組み込みシェルコマンド

コマンド	説明
X:	Windows のドライブ **X** のディレクトリに移動します。
cd	ホームディレクトリに移動します。
cd –	1 つ前のディレクトリに移動します。
cd ––	2 つ前のディレクトリに移動します。
cd –––	3 つ前のディレクトリに移動します。
cd *dir*	ディレクトリ *dir* に移動します。
cd *dir name*	空白を含むディレクトリ *dir name* に移動します。
chdir	ホームディレクトリに移動します。
chdir –	1 つ前のディレクトリに移動します。
chdir ––	2 つ前のディレクトリに移動します。
chdir –––	3 つ前のディレクトリに移動します。
chdir *dir*	ディレクトリ *dir* に移動します。
chdir *dir name*	空白を含むディレクトリ *dir name* に移動します。
. *filename*	ドットコマンド。サブシェルではなく現在の シェルでコマンド *filename* を読み取り、実行します。
exec *command*	現在のシェルでコマンドを実行します。

ユーザーが作業しているディレクトリは、**現在の作業ディレクトリ**または *cwd* (current working directory) と呼ばれます。現在の作業ディレクトリを確認するには、シェルプロンプトでコマンド **pwd** を入力します。

Ch には、単純、絶対、および相対という 3 種類のディレクトリ名またはパス名があります。単純パス名は、ファイルシステム階層内の位置に関する情報を含まないファイル名またはディレクトリ名です。単純パス名は、現在の作業ディレクトリのサブディレクトリに移動するときに使用します。絶対パス名は、ファイルシステム階層内のディレクトリの絶対位置を示します。絶対パス名は、ルートディレクトリを表す文字 '/' から始まります。Windows では、絶対パス名が `X:/`のようなドライブを表す文字で始まる場合もあります。たとえば、パス名**/usr/ch** は、ルートディレクトリを起点にしたディレクトリ **ch** の絶対位置を示します。相対パス名は、ルートではなく作業ディレクトリを起点にして目的のファイルまたはディレクトリへのパスを表します。たとえば、パス名**../ch** は現在の作業ディレクトリを起点にしてディレクトリの相対位置を示します。相対パス名では、**.** シンボルは現在の作業ディレクトリを、**..** シンボルは親ディレクトリをそれぞれ表します。

Ch シェルでは、組み込みコマンド **cd** および **chdir** を使用すると、現在の作業ディレクトリから目的のディレクトリに移動できます。ディレクトリ名を指定しないコマンド **cd** または **chdir** では、現在の作業ディレクトリからシステム変数 **_home** が示すホームディレクトリに移動します。コマンド **cd** *dir* または **chdir** *dir* は、現在の作業ディレクトリからディレクトリ *dir* に移動します。コマンド **cd** – または **chdir** –は、現在の作業ディレクトリから 1 つ前のディレクトリに移動します。同様に、**cd** ––

または **chdir** -- は 2 つ前のディレクトリに移動し、**cd** ---または **chdir** ---は 3 つ前のディレクトリに移動します。次に例を示します。

```
> pwd
/home
> cd /usr/ch
> pwd
/usr/ch
> chdir -
> pwd
/home
> cd myname
> pwd
/home/myname
> cd ../../usr/ch
> pwd
/usr/ch
>
```

ここで、`/usr/ch` は絶対パス名、`myname` は単純パス名、`../../usr/ch` は相対パス名です。
セクション4.2と4.3に、ドットコマンドを使用して現在のシェルでプログラムを実行する例を示しています。
組み込みコマンド **exec** は、現在のシェルに代わって他のコマンドを実行します。現在のシェルは終了します。

4.4.1 対話型シェル専用のコマンド

Ch の対話型のシェルとシェルプログラムの両方で、すべての演算子と関数および組み込みコマンドが使用可能です。しかし、コマンドラインモードで有効なすべてのコマンドがシェルプログラムで使用できるわけではありません。Ch を対話型シェルとして呼び出したときだけ有効なコマンドは、**対話型コマンド**と呼ばれます。対話型コマンドは Ch プログラム内では無効です。表4.3にすべての対話型コマンドを示します。

表 4.3: 対話型シェルだけで有効な対話型コマンド

コマンド	説明
!	前に実行したコマンドを繰り返します。
chdebug *filename*	プログラム *filename* をデバッグします。
chparse [-S] *filename*	構文を確認するためだけに *filename* を解析します。 -S オプションはセーフシェルを指定します。
chrun	解析済みのプログラムを実行します。
exit	Ch シェルを終了します。
history	コマンド履歴を表示します。
remvar	変数を削除します。
remkey	キーワードを削除します。
showvar	すべてのスタック内の変数とその値を表示します。

Ch シェルの対話型コマンドラインモードでは、変数 (関数型の変数を含む) を **remvar** コマンドで削除できます。次に例を示します。

```
> int i         // define variable i
> i = 90
90
> remvar i      // remove variable i
> i
ERROR: variable 'i' not defined
ERROR: command 'i' not found
>
```

コマンド `int i` は Ch シェルで変数を宣言し、コマンド `remvar i` は変数 `i` を削除します。コマンド **remvar** は、対話型コマンドであり、Ch プログラム内では無効です。シェルプログラム内にある `var` などの変数を削除する場合は、プリプロセッサディレクティブ
`#pragma remvar(var)` を使用する必要があります。
同様に、**remkey** コマンドで次のようにキーワードを削除することができます。

```
> remkey(sin)    // generic function sin is removed as a keyword
> float sin
> sin =10.0
```

プログラム内では、プリプロセッサディレクティブ `#pragma remkey(key)` を使用して汎用関数 `sin()` を削除する必要があります。
コマンド **showvar** を使用すると、現在のシェルのすべてのグローバル変数とその値を表示することができます。変数の値を表示するには、既定の形式が使用されます。sruct/class/enum 型、関数定義のない関数プロトタイプ、および型定義された変数のタグ名は表示されません。構造体の型および配列のメンバは、インデントなしで表示されます。次に例を示します。

```
> int x = 3;
> double d = 10.1234
> double a[2][3] = {1,2,3,4,5,6};
> array double b[2][3] = {1,2,3,4,5,6};
> struct tag {int i, int j;} s = {10, 20};
> showvar
    x              3
    d              10.1234
    a [C array]
1 2 3
4 5 6
    b [Ch array]
1 2 3
4 5 6
    s
.i = 10
.j = 20
```

コマンド **showvar** を使用すると、セクション4.2.1に示したように現在のシェル内で実行されるコマンド内のすべての変数とその値を表示することができます。

イベント指定子 ! とコマンド **history** の詳細については、次のセクション4.5で説明します。コマンド **chdebug**、**chparse**、および **chrun** の詳細については、セクション4.5を参照してください。汎用関数 **alias()** は、通常、システムスタートアップファイル **chrc** およびユーザースタートアップファイル **.chrc** (Unix) または **_chrc** (Windows) で使用されます。コマンドモードではコマンド **alias** および **unalias** (セクション 4.6 を参照) を使用できます。

4.5 プロンプトでのコマンドの繰り返し

このセクションで説明する機能は、 Ch シェルのコマンドラインモードでのみ有効です。このセクションでは、プロンプトでコマンドを繰り返すための履歴と簡易置換について説明します。

プロンプトでのコマンドの繰り返しに最も便利な方法は、方向キーを使用することです。現在のコマンドの前に入力したコマンドの場合はキーボードの上矢印キー'↑' を、現在のコマンドの後に入力したコマンドの場合は下矢印キー'↓' をそれぞれ使用することで、以前に入力したコマンドを簡単に取得できます。取得したコマンドを変更するには、まず、キーワード上で左矢印キー'←' または右矢印キー'→' を使用してその場所までカーソルを移動します。次に、Del キーまたは Backspace キーを使用して文字を削除するか、任意の文字を入力して、Emacs テキストエディタを使用する場合と同様にコマンドライン編集用の文字を挿入します。

4.5.1 履歴置換

履歴置換を使用すると、以前にシェルプロンプトで入力したコマンドの単語を使用することができます。これにより、複雑なコマンドや引数のスペルの訂正や繰り返しが容易になります。コマンドラ

インは履歴の一覧に保存されます。この一覧のサイズはシステム変数_histsize によって制御されます。履歴はシェルコマンド history で表示できます。最新のコマンドは保持されます。先頭が ! 記号である履歴置換は、コマンドラインの先頭でのみ実行できます。履歴置換は入れ子にできません。

次にコマンドの例を示します。

```
> _histsize        // print the current value of _histsize
20
> _histsize = 4    // change the current value of _histsize to 4
4
> pwd
/usr/ch
> history          // print the history list of commands
123   _histsize        // print the current value of _histsize
124   _histsize = 4    // change the current value of _histsize to 4
125   pwd
126   history          // print the history list of commands
>
```

_histsize の現在の値を出力し、次にこの値を 4 に変更します。もう 1 つのコマンド **pwd** の後にコマンド **history** が指定されている場合は、履歴の一覧内のコメントを含む直近の 4 つのコマンドを出力します。各コマンドの前に表示される数字はコマンドライン番号です。履歴置換では、イベント指定子を使用することで履歴の一覧内にある以前のコマンドラインを繰り返すことができます。

イベント指示子は履歴の一覧内のコマンドラインエントリへの参照です。表 4.4 に示すさまざまなイベント指示子を使用すると、履歴の一覧にある長いコマンドラインの実行を繰り返す場合にさらに便利です。最もよく使用されるイベント指示子は!です。!はユーザーが入力した最後のコマンドラインを繰り返します。たとえば、more コマンドを使用してファイルを表示し、目的のファイルの部分が存在しない場合は、!を入力するだけで more を繰り返すことができます。!によって繰り返したコマンドラインが最初に表示されてから、実行されます。そのため、正しいコマンドラインを入力したかどうかを確認できます。!は、さらに高度で時間を節約できる多くのイベント指定子の基礎となります。イベント指定子を表 4.4 に示しています。コマンド!n は、コマンドの履歴の一覧にある番号が n のコマンドを繰り返します。コマンド!-n は、番号が m-n であるコマンドを繰り返します。ここで、m は現在のコマンドの番号であるとします。つまり、コマンド!-1 はコマンド! と同じです。また、コマンド!str もコマンドを繰り返すためによく使用されるコマンドです。最新のファイルを再度編集する場合は、前の vi コマンドの番号を思い出す必要はなく、!vi と入力するだけで済みます。

表 4.4: イベント指定子

コマンド	説明
!	1 つ前のコマンドを参照します。この置換は単独で 1 つ前のコマンドを繰り返します。
!!	!と同じです。
!n	コマンドライン *n* を参照します。
!-n	現在のコマンドライン-*n* を参照します。
!str	先頭が *str* である最新のコマンドを参照します。

次の例は、イベント指示子を使用して、履歴の一覧にあるコマンドを繰り返す方法を示しています。

```
> _histsize = 5
5
> pwd
/usr/local//ch
> !
pwd
/usr/local/ch
> strlen("abc")
3
> history
136   _histsize = 5
137   pwd
138   pwd
139   strlen("abc")
140   history
> !137
pwd
/usr/local/ch
> !h
history
138   pwd
139   strlen("abc")
140   history
141   pwd
142   history
> !-4
strlen("abc")
3
>
```

4.5.2 簡易置換

簡易置換では、前のコマンドに対して変更を行い、同時に、変更後のコマンドを実行できます。これは、コマンド内の入力ミスを修正したり、同様のコマンドを繰り返したりするときに便利です。

表 4.5: 簡易置換

コマンド	説明
^*old*^*new*	前のコマンド内の文字列 *old* を文字列 *new* に置き換えます。
^*old*^*new*^	^*old*^*new* と同じです。
^*old*	前のコマンド内の文字列 *old* を削除します。
^*old*^	^*old* と同じです。

コマンド^*old*^*new* および^*old*^*new*^では、前のコマンドの文字列 *old* を文字列 *new* に置き換えることができます。次に例を示します。

```
> mkkdir mydir
ERROR: variable 'mkkdir' not defined
ERROR: command 'mkkdir' not found
> ^kk^k
> history
11    mkkdir mydir
12    mkdir mydir
13    history
>
```

入力ミス `mkkdir` を訂正するには、コマンド `^kk^k` を使用します。次の例では、簡易置換コマンドを使用して、異なる 5 つの月に対して 5 つのディレクトリを作成する繰り返しタスクを実行します。

```
> mkdir Jan
> ^Jan^Feb
> ^Feb^March
> ^March^April
> ^April^May
> history
31    mkdir Jan
32    mkdir Feb
33    mkdir March
34    mkdir April
35    mkdir May
36    history
>
```

簡易置換コマンド^*old* および^*old*^を使用すると、前のコマンドの文字列 *old* を削除できます。次に例を示します。

```
> cp file file1.c
> ^1
> history
56   cp file file1.c
57   cp file file.c
58   history
>
```

4.5.3 ファイルの補完

Ch シェルは、一意の語の一部が入力された場合にその語を補完することができます。語の一部(たとえば、”ls /usr/local/ch/de”) を入力し、Tab キーを押すと、シェルはファイル名”/usr/local/ch/de” を”/usr/local/ch/demos/”と補完し、不完全な語を入力バッファ内の完全な語に置き換えます。末尾の”/”; の補完では、補完されたディレクトリの末尾に”/”を追加し、他の補完された語の末尾にスペースを追加します。これにより、入力を迅速化し、正確な入力のための視覚的なインジケータを提供できます。

完全一致がない (‘/usr/local/ch/demos’ が存在しない) 場合は、端末がベル音を発します。単語が既に完全である (システムに ‘/usr/local/ch/de’ が存在するか、先行して全体を入力した) 場合は、末尾に ‘/’ またはスペースが抜けていればそれが追加されます。

ファイル補完は入力バッファの末尾でのみ機能します。

複数の選択肢がある場合、シェルはコマンド ls `ls -F` を使用して実行可能な補完を一覧に表示し、プロンプトと未完成のコマンドラインを再度出力します。次に例を示します。

```
> ls /usr/local/ch/d[^D]
dl/      demos/  docs/
> ls /usr/local/ch/d
```

選択肢が 100 を超える場合は、すべての選択肢を表示するかどうかをユーザーに確認します。

```
> ls fil[tab]
Display all 102 choices? (y or n)
```

Ch シェルでは、さらに入力を続けるとより長い一致が得られる可能性がある場合でも、次のように最短の使用可能な一意の一致で補完を実行します。

```
> ls
fodder   foo       food      foonly
> rm fo[tab]
```

この場合、単にビープ音を発します。これは、”fo”が”fod”または”foo”に拡張する可能性があるためです。しかし、もう 1 つの”o”を入力すると

```
> rm foo[tab]
foo food foonly
> rm foo
```

補完では、‘food’ や ‘foonly’ とも一致する場合でも、‘foo’ と補完します。
最初のコマンドが”cd”である場合は、補完機能ではディレクトリのみを含む選択肢を表示するだけです。

```
> ls dir[tab]
dir1/ dir2/ dir3/ dir4@ dirf1 dirf2 dirf3@
> cd dir[tab]
dir1/ dir2/ dir3/ dir4@
```

ファイルまたはディレクトリのシンボリックリンクには、シンボル’@’ が付いています。
シェルはファイル補完では'\ ' をスペースとして、'\$' を'$' として処理します。

```
> ls test\ t[tab]
ls "juck tmp"
```

シェルは、上記のように、ファイル補完にスペースを含むディレクトリを囲む二重引用符を追加します。
組み込みコマンド **cd** の場合、ファイル名補完にスペースを含むディレクトリの場合でもバックスラッシュを省略できます。

```
> cd aa b[tab]
> cd "aa bb"/
```

Windows のディレクトリ名またはファイル名には、多くの場合、スペースが含まれます。その場合でも、シェルでファイル名またはディレクトリ名を補完できます。

```
> cd Prog[tab]
> cd "Program Files"/
```

4.5.4 コマンドの補完

最初のトークンの末尾より前で Tab キーを押すと、Ch シェルはコマンド補完を使用してトークンを処理します。シェルは、環境変数 `PATH` に指定されているディレクトリでファイルを検索します。Windows および Unix の両方で、この環境変数の値はシステム変数 **_path** と同じです。Unix でのコマンドの補完には、実行可能ファイルのみが選択されます。Windows では、拡張子 `.com`、`.exe`、`.bat`、`.cmd`、`.ch` を持つファイルのみが選択されます。
一致したコマンドが 1 つだけの場合、シェルでは不完全なコマンドを入力バッファ内にある完全なコマンドに置き換えます。シェルでは、その他の補完されたコマンドの末尾にスペースを追加して、入力を迅速化し、正しく補完するための視覚的なインジケータを提供します。次に例を示します。

```
> lps[tab]
> lpstat
```

ファイル補完と同様に、複数の選択肢がある場合、シェルはコマンド `ls -F` を使用して実行可能な補完を一覧に表示し、プロンプトと未完成のコマンドラインを再度出力します。選択肢が 100 を超える場合は、すべての選択肢を表示するかどうかを確認します。

```
> lp[tab]
lp lpstat
> lp
```

環境変数 `PATH` で指定されたディレクトリ内に一致するコマンドがない場合、シェルは現在のディレクトリで一致する可能性のあるディレクトリを検索します。一致するディレクトリが 1 つだけの場合、シェルは不完全なコマンドを入力バッファ内にある完全なコマンドに置き換えます。また、入力を迅速化するため、補完されたディレクトリの末尾に'/' を追加します。ディレクトリに複数の選択肢がある場合、シェルはコマンド `ls -F` を使用して実行可能な補完を一覧に表示し、プロンプトと未完成のコマンドラインを再度出力します。また、選択肢が 100 を超える場合は、すべての選択肢を表示するかどうかを確認します。

一致がまったくない場合は、端末がベル音を発します。

コマンド補完のために現在のディレクトリ内でのみコマンドを検索するには、現在のディレクトリを表す`./`で始まるコマンドを入力する必要があります。次に例を示します。

```
> cd /bin
> ./log[tab]
logger   login    logname
> ./log
```

コマンドラインでタブを直接入力すると、シェルはすべてのコマンドを表示します。

```
> [tab]
Display all 1296 choices? (y or n)
```

4.6 別名

対話型コマンドモードでは、Ch シェルは **alias** コマンドおよび **unalias** コマンドを使用して、作成、表示、変更できる別名の一覧を管理しています。シェルは各コマンドの最初の語を確認して、それが既存の別名の名前に一致しているかどうかを確認します。一致している場合は、コマンドはその名前に置き換える別名定義を使用して再処理されます。

別名は、通常、システムスタートアップファイル **chrc** およびユーザーのホームディレクトリにあるスタートアップファイル**.chrc** (Unix) または**_chrc** (Windows) で、汎用関数 **alias**() を使用して作成されます。汎用関数 **alias**() は次のプロトタイプを使用してオーバーロードされます。

```
int alias(string_t name, string_t alius);
string_t alias(string_t name);
int alias(void);
```

さまざまな引数とそれに対応する戻り値を表4.6に示します。関数呼び出し **alias** (`name`, `alius`) は、シンボル `name` をコマンド `alius` の別名に設定します。

表 4.6: 関数呼び出し alias()

関す呼び出し	戻り値
alias("*name1*", "*alius*")	0
alias("*name1*", "*alius*")	1
alias("*name2*", "")	0
alias("*name2*", NULL)	0
alias("*name3*", NULL)	1
alias(NULL, "*alius*")	-1
alias(NULL, NULL)	-1
alias("*name1*")	alius
alias("*name3*")	NULL
alias(NULL)	NULL

`name` が有効な識別子である場合、関数は 0 を返します。`name` が既に別名である場合、関数は 1 を返します。`name` の値が NULL である場合、-1 を返します。2 番目の引数 `alius` が NULL である場合、関数はコマンド `alius` からシンボル `name` の別名設定を解除します。関数呼び出し **alias**(`name`) は、文字列としてシンボル `name` の別名を返します。シンボル `name` が別名でない場合、関数は NULL を返します。関数呼び出し **alias**() では、すべての名前だけでなくその別名も標準出力に出力し、別名の数を返します。この汎用関数の戻り値は対話型の実行セッション内では、次のように示されます。関数 alias() はコマンドモードとシェルプログラムの両方で呼び出すことができます。C の規則に従って、別名内の文字`'\'`や`'"'`は、エスケープ文字`'\'`を使用してそれぞれ`'\\'`および`'\"'`として渡すことができます。次のコマンドは、関数 **alias**() のさまざまな機能を示しています。

```
> alias("ls", "ls -a")
0
> alias("ls", "ls -agl")
1
> alias("cp", "cp -i")
0
> alias()
cp      cp -i
ls      ls -alg
2
> alias("ls", NULL)
0
> alias()
cp      cp -i
1
```

```
> alias("cp")
cp -i
>
```

表 4.7: **alias**() 内の仮引数

仮引数	説明
_argv[0]	最初の入力語 (コマンド)。
_argv[n]	n 番目の引数。
_argv[#]	コマンドライン全体。
_argv[$]	最後の引数
_argv[*]	すべての引数、またはコマンドに 1 つの単語しかない場合は NULL 値。

別名では**引数置換**を使用できます。別名の定義内にある表4.7に示されている仮引数は、別名が使用されるときにコマンドラインの実際の引数と置き換えられます。引数の置換を要求しない場合、引数はそのまま維持されます。次に例を示します。

```
> echo abc xyz
abc xyz
> alias("myecho1", "echo _argv[1]")
> myecho1 abc xyz
abc
> alias("myecho2", "echo _argv[$]")
> myecho1 abc xyz
xyz
```

この例では、別名にコマンド `myecho1 abc xyz` の最初の引数のみが使用されます。最後の引数は、別名 `myecho2` で使用されます。別の例としては、現在のディレクトリとそのサブディレクトリ内でファイルを検索して出力するには、次のように別名 `find` を使用してシステムコマンド **find** を置き換えることができます。

```
> alias("find", "find . -name _argv[1] -print")
> find filename
(display files with name 'filename')
```

現在のプロセスでは、表4.8に示したコマンド **alias** および **unalias** を対話型のコマンドシェルでさらに便利に使用できます。

表 4.8: コマンド **alias** および **unalias**

コマンド	説明
alias *name alius*	別名を作成します。
alias *name* "*string with space*"	別名を作成します。
alias *name*	*name* の別名を表示します。
alias	すべての別名を表示します。
unalias *name*	*name* の別名を解除します。

これらの 2 つのコマンドはコマンドモードでのみ有効です。次に例を示します。

```
> alias ls "ls -agl"
> alias cp "cp -i"
> alias
cp      cp -i
ls      ls -alg
> unalias ls
> alias
cp      cp -i
> alias cp
cp -i
>
```

別名は入れ子にすることができます。つまり、別名定義にもう 1 つの別名の名前を含むことができます。これは、次のようなパイプラインで便利です。

```
> alias("ls", "ls -agl")
> alias("lm", "ls * | more")
```

コマンド lm が呼び出されたとき、実際は次のように展開されたコマンド

```
> ls -agl * | more
```

が代わりに呼び出されます。`ls` の出力はプログラム `more` を使用してパイプ処理されます。もう 1 つの例としては、次のコマンドの別名 `opentgz` を手軽に使用して、`file.tar.gz`、`file.tgz` など圧縮済みの保存ファイルを解凍することができます。

```
> alias("opentgz", "gzip -cd _argv[1] | tar -xvf -")
> opentgz file.tar.gz
(display extracted files from file.tar.gz)
```

このコマンドでは、セクション4.11で説明するパイプラインを使用します。
入れ子にした別名は、引数置換が適用される前に展開されます。次に例を示します。

```
> alias("t1", "t2 _argv[1] A")
> alias("t2", "echo a b c")
> t1 x y z
a b c x A
> alias("p1", "p2 a b c")
> alias("p2", "echo _argv[1] A")
> p1 x y z
a A
```

4.7 変数置換

変数置換を使用して、変数名を変数の値で置き換えることができます。変数置換の 3 つの構文$*var*、$(*var*)、および${*var*} を表4.9に示します。

表 4.9: 変数置換

コマンド	説明
$*var*	変数 *var* の値で置換されます。
${*var*}	変数 *var* の値で置換されます。
$(*var*)	変数 *var* の値で置換されます。

展開される変数名またはシンボルはかっこまたは中かっこで囲むことができます。これは、オプションですが、直後の文字列 (その名前の一部として解釈される可能性のある) から展開される変数を保護するのに役立ちます。変数置換によりコードはさらに移植しやすく、柔軟になります。これは、変数がさまざまな状況ごとに異なる値を持つことができるためです。たとえば、変数置換を使用したインストールプログラムでは、既定ディレクトリの代わりに別のインストール先ディレクトリを指定できます。ユーザーは選択したディレクトリにソフトウェアをインストールできます。

変数置換は、入力コマンドラインが分析され、別名が解決された後に実行されます。変数置換は、対話型コマンドモード、プログラム内のコマンドステートメント、およびセクション4.9で説明するコマンド置換処理でのみ有効です。

変数置換での変数は、セクション2.3.1で説明している定義済みの識別子、つまり、文字列のユーザー定義変数、char へのポインタ、または整数のデータ型、セクション4.14で説明する環境変数または未定義のシンボルを使用できます。変数置換の場合、Ch シェルは、最初に Ch の名前空間でスコープ規則に従って変数名を検索します。変数が定義されていない場合は、現在のプロセスの環境変数を検索します。指定された名前を持つ変数が Ch の空間にも環境の空間にもない場合は、置換は行われず、変数は無視されます。

'\' の前に '$' を付けることで変数置換を省略することができます。ただし、コマンド置換が必ず行われる '`' 内とまったく行われない ''' 内を除きます。空白、タブ、または行の末尾の前にある場合、'$' はそのまま渡されます。たとえば、`myname` がユーザーのアカウント名であり、次のようなコマンドがあるとします。

```
> _home                          // _home is a predefined identifier
```

```
/home/myname
> cd $_home
> pwd
/home/myname
> _fpathext                    // _fpathext is a predefined identifier
.chf
> // copy file1 to file1.ch
> cp file1 file1$_pathext
> cd $CHHOME                 // CHHOME is an environment variable
> pwd
/usr/ch
> echo $ CHHOME \$10.5 ${_home}/tmp
$ CHHOME $10.5  /home/myname/tmp
```

これらのコマンドは、$*var* および${*var*} を使用する変数置換の例です。変数 **_home** と **_fpathext** は Ch の定義済み識別子です。変数 **_home** には現在のユーザーのホームディレクトリが含まれています。ユーザーごとに **_home** の値は異なります。シェルプログラムでは、コマンド `cd $_home` はコマンド `cd /home/myname` より柔軟です。同様に、環境変数 CHHOME には Ch のホームディレクトリが含まれています。これはマシンごとに異なる場合があります。コマンド `cd $CHHOME` を使用するシェルプログラムは、コマンド `cd /usr/ch` を使用するプログラムより移植しやすいことは明らかです。すぐ後ろに数字が続く'$' を表示するには、`'\'`. を前に付ける必要があります。

また、`/home/myname/project1/subproject2/plan1` という非常に長い名前のディレクトリにあるファイルを使用してユーザーが作業しているとします。このパス名は文字列`$mydir` と省略できます。次に、このディレクトリをコマンドで使用する場合は、コマンドラインで `/home/myname/project1/subproject2/plan1` の代わりに`$mydir` を使用できます。次に例を示します。

```
> string_t mydir = "/home/myname/project1/subproject2/plan1"
> cd $mydir
> pwd
/home/myname/project1/subproject2/plan1
>
```

4.7.1 式置換

式置換では、Ch の式を評価して結果を置換できます。式置換の書式は次のとおりです。

```
$(expression)
```

または

```
${expression}
```

式は、文字列の式、char へのポインタ、または整数のデータ型である必要があり、定数、変数、関数呼び出し、算術式などの有効な式とすることができます。

単一の変数を式として処理できます。変数置換は単一の変数の値を取得するために使用できますが、式置換は通常、より複雑な式に使用します。環境変数または未定義のシンボルでは、変数置換を使用する必要があります。次の例では、関数 **getenv()** を呼び出す式によってコマンドで使用している環境関数を取得します。次に例を示します。

```
> getenv("CHHOME")          // CHHOME is an environment variable
/usr/local/ch
> cd $(getenv("CHHOME"))
> pwd
/usr/local/ch
> ls $CHHOME/include
... (list of /usr/local/ch/include)
> ls ${CHHOME}/include
... (list of /usr/local/ch/include)
> ls $(getenv("CHHOME"))/include
... (list of /usr/local/ch/include)
> ls $(stradd(getenv("CHHOME"),"/include"))
... (list of /usr/local/ch/include)
>
```

ここで、`ls $CHHOME/include`、`ls ${CHHOME}/include`、`ls $(getenv("CHHOME"))/include`、および `ls $(stradd(getenv("CHHOME"),"/include"))` はコマンド `ls $(stradd(getenv("CHHOME"),"/include"))` と同等です。ただし、`CHHOME` を使用したコマンドの方が移植性は高くなります。文字'/' は識別子として有効な文字ではないので、変数 `CHHOME` の変数置換の場合、中かっこは省略可能です。

4.7.2 コマンド名置換

コマンド名置換はコマンドの実行に役立ちます。実行するコマンドは、実行時に動的に取得されます。コマンド名置換の構文は、変数名置換および式置換の構文と同じです。コマンド名置換の場合、構文ステートメントの先頭に $を指定する必要があります。$記号の後に続く変数または式のデータ型は、文字列、char へのポインタまたは符号なし char 型である必要があります。次に例を示します。

```
string_t cmd = "/Ch/bin/echo.exe option";
$cmd more options
string_t cmd2 = "\"C:/Program Files/ch/bin/echo.exe\"";
$cmd2 option2
char *cmd3 = "ls";
$cmd2 -agl
```

文字列 `cmd` には、コマンド `/Ch/bin/echo.exe` とその `option` の両方が含まれています。空白を含むコマンドを使用するには、コマンド `C:/Program Files/ch/bin/echo.exe` の上記の文

字列 cmd2 に示すようにそのコマンドを二重引用符で囲む必要があります。文字列 cmd3 は、char へのポインタの形式でコマンド ls を含んでいます。

4.8 ファイル名置換

ワイルドカード文字と呼ばれる特定の特殊文字を使用すると、ファイル名置換によりファイル名およびディレクトリ名を短縮することができます。表4.10に Ch で有効なワイルドカード文字を示します。

表 4.10: ファイル名置換

ワイルドカード文字	説明
*	任意の (0 個以上の) 文字に一致します。
?	任意の一文字に一致します。
~	変数_home の値によって指定されるホームディレクトリ。または、ユーザーのパスワード入力によって指定されるユーザーのホームディレクトリ。
./	現在の作業ディレクトリ。
../	現在の作業ディレクトリの親ディレクトリ。

シンボル '?' は、1 文字の値を表すワイルドカード文字であり、*は任意の文字数の文字を表します。たとえば、現在の作業ディレクトリ内のすべてのファイルを表示するには、次のように入力します。

```
> // list all files in current directory with *
> ls *
abc1.ch   abc2.ch   abc3.ch   abc12.c   efc1.c
>
```

拡張子が.ch であるファイルをすべて表示するには、次のように入力します。

```
> ls *.ch
abc1.ch   abc2.ch   abc3.ch
>
```

名前に文字列 c1 が含まれるファイルを表示するには、次のように入力します。

```
> ls *c1*
abc1.ch   abc12.c   efc1.c
```

名前の先頭の文字列が abc で末尾の文字列が.ch であり、この 2 つの文字列の間に 1 つだけ文字が含まれるファイルを表示するには、次のように入力します。

```
> ls abc?.ch
abc1.ch   abc2.ch   abc3.ch
>
```

ホームディレクトリはチルダ文字~として指定できます。たとえば、現在の作業ディレクトリにかかわらず、~の省略形を使用するとホームディレクトリ内のファイルを一覧表示できます。ホームディレクトリのパス名の代わりに~を置き換えることにより、入力は削減されますが、現在の作業ディレクトリは変わりません。たとえば、ユーザーのアカウント名が myname であるとします。

```
> // print the current user name
> echo $_user
myname
> // list files in  home directory of current user
> ls ~
... (list files in home directory of myname)> pwd
>
```

現在の作業ディレクトリは./と指定できます。この置換はさまざまな状況で役立ちます。たとえば、離れているディレクトリのファイルを作業ディレクトリにコピーする必要がある場合、コマンド **cp** とその後続のファイルのパス名と現在の作業ディレクトリの省略形./を使用できます。たとえば、次のコマンド、

```
> pwd
/home/myname/project2
> cp /home/myname/project1/subproject2/plan1/* .
>
```

は、ディレクトリ/home/myname/project1/subproject2/plan1 のすべてのファイルを現在の作業ディレクトリにコピーします。

現在の作業ディレクトリの省略形./のもう 1 つの使用法は、現在のディレクトリ内のプログラムが実行されているかを確認することです。test などよく使用されるファイル名は、異なるディレクトリで複数のプログラムに使用される場合があります。現在のディレクトリが変数_**path** によって指定された検索パスに含まれていないか、または現在のディレクトリが test というファイルの含む他のディレクトリより優先度が低い場合は、ファイル名 test を入力すると、現在のディレクトリのプログラムではなく、他のディレクトリのプログラム test が実行されます。コマンド./test.c では、確実に現在のディレクトリ内でプログラムを実行します。コマンド which -a test を使用すると、検索パスの順序で test という名前のプログラムをすべて一覧表示することができます。次の例では、現在のディレクトリが検索パスに含まれていません。

```
> pwd
/home/myname/project1/subproject2/plan1
> which -a test
/bin/test
/usr/bin/test
```

```
/usr/local/gnu/bin/test
/pkg/gnu/bin/test
> test    // execute /bin/test
> ./test  // execute /home/myname/project1/subproject2/plan1/test
```

現在の作業ディレクトリの親ディレクトリは`..`と指定することができます。親ディレクトリの省略形の最も一般的な使用は、現在の作業ディレクトリから親ディレクトリまたはそのサブディレクトリに切り替えることです。

```
> // print current working directory
> pwd
/home/myname/project1/subproject2/plan2
> // go to directory plan1 of the parent directory
> cd ../plan1
> pwd
/home/myname/project1/subproject2/plan1
>
```

この例のコマンド `cd ../plan1` では、現在の作業ディレクトリが親ディレクトリ `/home/myname/project1/subproject2` のサブディレクトリ `plan1` に変更されます。

4.9 コマンド置換

コマンド置換では、1つのコマンドの出力をプログラムまたはコマンドシェル内の変数にパイプで渡します。このためには、逆引用符と呼ばれることもあるアクセント記号`で埋め込みコマンドを囲む必要があります。次に例を示します。

```
> string_t s = `date`
Wed Jul 25 10:11:18 PDT 2001
> s
Wed Jul 25 10:11:18 PDT 2001
> char *s1 = `date`
> s1
Wed Jul 25 10:12:15 PDT 2001
> s1[0]
W
> free(s1)  // the memory should be freed
>
```

コマンド `string_t s = `date`` および `char *s1 = `date`` では、コマンド **date** の出力を変数 `s` および `s1` にそれぞれパイプで渡します。コマンドの別名とは異なり、変数 `s` と `s1` はコマンド **date** と同等ではありません。これらの変数は、コマンド **date** の出力を格納するだけです。つまり、`s` と `s1` の内容は時間が経過しても変わりませんが、実行コマンド `date` の出力は、時間の経過とと

もに変わります。変数 s1 に割り当てられたメモリは後で解放する必要があることに注意してください。コマンド置換を実装するには、char *型ではなく **string_t** 型を使用することをお勧めします。**string_t** の詳細については、セクション17.2で説明します。コマンド置換のもう 1 つの例として、次の Unix コマンドがあります。

```
vi `grep -l "str1 str2" * `
```

このコマンドにより、現在のディレクトリ内にある文字列 *str1 str2* を含むすべてのファイルを、*vi* テキストエディタ4.7を使用して編集することができます。

セクション4.7で説明した変数置換をコマンド置換の中で使用できます。変数は Ch の名前空間または環境の名前空間での名前とすることができます。また、有効な Ch 式を変数置換で使用することもできます。次に例を示します。

```
> string_t s1 = "/bin", s2;
> s2 = `ls $s1`;
... (list of /bin)
> echo `ls $s1`;
... (list of /bin)
```

'\' の前に '$' を付けると変数置換を省略できることに注意してください。ただし、コマンド置換が必ず行われる ' ` ' 内とまったく行われない" '"内を除きます。空白、タブ、または行の末尾の前にある場合、'$' はそのまま渡されます。たとえば、変数 f1 と f2 がそれぞれ file1 と file2 の値を持つ場合、次の式

```
`echo $f1 \$f2 |sed 's/endofline$/converted/'`
```

は、以下と同等です。

```
`echo file1 \file2 |sed 's/endofline$/converted/'`
```

コマンド置換の単語一覧の各項目を個別に参照しやすくするには、アクセント記号`で囲まれているコマンドからの出力を後処理します。連続した空白文字、改ページ、改行、復帰、水平タブ、および垂直タブを示す文字は、単一の空白文字に置き換えられます。これは C シェルと同じです。

二重のアクセント記号``で囲まれているコマンドの出力は変更されません。たとえば、コマンド **date** の出力に含まれる余分な空白は、コマンド置換
``date`` によって保持されます。

```
> string_t s = ``date``
> s
Mon Aug  6 11:44:16 PDT 2001
> s = `date`
Mon Aug 6 11:44:24 PDT 2001
>
```

コマンド``date``の出力で、単語 Aug の後にある 2 つの空白文字は保持されます。これらはコマンド`date`の場合は単一の空白に置き換えられます。

4.10 入出力のリダイレクト

Ch では、コマンドの入出力は、Bourne シェルの規則に従うシェルによって解釈される特殊な表記を使用して、リダイレクトすることができます。表4.11に示すリダイレクト表記は、対話型シェルで入力するコマンドまたは Ch プログラムのコマンドラインで使用できます。

表 4.11: 入出力のリダイレクト

表記	説明
cmd < *word*	ファイル *word* を標準入力 (ファイル記述子 0) として使用します。
cmd > *word*	ファイル *word* を標準出力 (ファイル記述子 1) として使用します。 ファイルが存在しない場合は作成され、存在する場合は、長さ 0 に切り捨てられます。
cmd > *word* 2>&1	標準出力および標準エラー (診断出力) をファイル *word* にリダイレクトします (UNIX)。
cmd 1 > *word* 2>&1	標準出力および標準エラー (診断出力) をファイル *word* にリダイレクトします (Windows)。
ch -r cmd > *word*	標準出力および標準エラー (診断出力) をファイル *word* にリダイレクトします (Unix と Windows の両方)。 ファイルが存在しない場合は作成され、存在する場合は、長さ 0 に切り捨てられます。
cmd >> *word*	ファイル *word* を標準出力として使用します。 ファイルが存在する場合、出力はそのファイルに追加 (まず EOF を検索する) され、その他の場合、ファイルは作成されます。
cmd >> *word* 2>&1	標準出力および標準エラー (診断出力) をファイル *word* にリダイレクトします。 ファイルが存在する場合、出力はそのファイルに追加 (まず EOF を検索する) され、その他の場合、ファイルは作成されます (Unix)。
cmd 1 >> *word* 2>&1	標準出力および標準エラー (診断出力) をファイル *word* にリダイレクトします。 ファイルが存在する場合、出力はそのファイルに追加 (まず EOF を検索する) され、その他の場合、ファイルは作成されます (Windows)。
cmd << *word*	*word* に対するパラメータとコマンド置換が実行されたら、シェル入力は、結果の *word* に完全に一致する最初の行までまたは EOF まで読み取られます。

出力リダイレクトを使用すると、後ろに引用符が続くコマンド `cmd > output` を使用して、コマ

ンドモードでファイル output を作成できます。'>' シンボルを入力して、ファイル output にコマンド cmd からの出力をリダイレクトします。システムはコマンド cmd が画面に出力した内容を取得し、その内容をファイル output に書き込みます。

シンボル '>' を使用すると、既存のファイルに出力をリダイレクトするときに、出力のリダイレクトによりそのファイルの現在の内容を削除し、コマンドの出力にその内容を置き換えます。追加リダイレクトシンボルと呼ばれるもう 1 つのリダイレクトシンボル '>>' を使用して、ファイルの内容が上書きされないようにすることができます。このシンボルでは、ファイルを置き換えるのではなく、ファイルの末尾にデータを**追加**します。存在しないファイルに出力を追加する場合は、'>>' というシンボルは '>' のように動作し、ファイルを作成し、コマンドの出力をそのファイルにリダイレクトします。

プロセスが、開いている各ファイルに番号を関連付けます。この番号を**ファイル記述子**と言います。Unix で Ch が起動されるときは、 Ch は 3 つのファイルに接続されます。ファイル記述子が 0 である標準入力、ファイル記述子が 1 である標準出力、ファイル記述子が 2 である標準エラーです。標準エラーは Windows では使用できません。記号 '>'、'<'、および '>>' の前にファイル記述子の数字を指定すると、0〜9 までの任意のファイル記述子をリダイレクトできます。たとえば、標準エラーをリダイレクトするには、2>を使用します。コマンド cmd > output 2> output を使用して、標準出力と標準エラーからの出力を同一のファイル output にリダイレクトできます。また、ファイル記述子 n は、シンボル 'n>&m' を使用すると別のファイル記述子 m と同じファイルにリダイレクトされるよう指定できます。たとえば、コマンド cmd > output 2>&1 は、コマンド **cmd** の出力をファイル output にリダイレクトし、

次に標準エラーをリダイレクトします。次のコマンドは、 Ch シェルでの入出力リダイレクトの動作を示しています。

```
> cat datefile
old content
> date > datefile
> cat datefile
Wed Jul 25 17:10:40 PDT 2001
>
> date >> datefile
> cat datefile
Wed Jul 25 17:10:40 PDT 2001
Wed Jul 25 17:11:45 PDT 2001
>
> cat > p1.c
int i, j
(To complete the file and quit from the command cat,
 use Ctrl-D in Unix or Ctrl-Z in Windows)
> p1.c > result_p1 2>&1
> cat result_p1
ERROR: missing ';'
ERROR: syntax error before or at line 2 in file p1.c
==>:
```

```
BUG: <== ???
WARNING: cannot execute command 'p1.c'
>
> cat > input_p2
10
(To complete the file and quit from the command cat,
 use Ctrl-D in Unix or Ctrl-Z in Windows)
> cat > p2.c
int i;
scanf("%d", i);
printf("%d\n", i);
(To complete the file and quit from the command cat,
 use Ctrl-D in Unix or Ctrl-Z in Windows)
> p2.c < input_p2
10
>
```

この例では、コマンド **date** の出力は、まず記号 '>' を使用してファイル `datefile` にリダイレクトされます。`datefile` の内容は上書きされます。次に、コマンド **date** の 2 番目の出力は、記号 '>>' を使用してファイル `datefile` にリダイレクトさます。これは、ファイル `datefile` を上書きするのではなくこのファイルの末尾に追加されます。ファイル `p2.c` の実行のエラーメッセージは、コマンド `p1.c > result_p1 2>&1` によってファイル `result_p1` にリダイレクトされます。記号 '<' を使用して、ファイル `p2.c` の対話型の実行で、既定の標準入力装置であるキーボードではなく、ファイル `input_p2` からの入力を取得します。

次の構文を考えます。

```
cmd >& word
```

これは標準出力と標準エラーの両方をファイル `word` にリダイレクトする構文であり、C シェルでは動作しますが、 Ch および Bourne シェルでは動作しません。次の構文

```
ch cmd > word 2>&1
```

を Unix の Ch シェルおよび Bourne シェルで使用します。Ch にはコマンドオプション `-r` があります (このオプションを使用すると、標準出力 stdout に標準エラー stderr を都合よくリダイレクトできます)。

以下のコマンドでは、

```
ch -r cmd > word
```

標準出力および標準エラーの**両方**をファイル `word` にリダイレクトします。

この構文は Unix と Windows の両方で機能します。

Unix では、関数 **system()** を使用すると、次に示すように、`cmd` などのコマンドの stdout をファイル word1 に、stderr をファイル `word2` にリダイレクトすることができます。

```
system("(cmd > word1) 2> word2");
```

4.11 パイプライン

パイプラインは、記号 | によって区切った 1 つ以上の一連のコマンドです。最後のコマンドを除く各コマンドの標準出力がパイプによって次のコマンドの標準入力に接続されます。各コマンドは個別のプロセスとして実行され、シェルでは最後のコマンドが終了するのを待機します。パイプラインの終了状態は、パイプラインの最後のコマンドの終了状態です。ユーザーは最初のコマンド実行されてその出力が一時ファイルにリダイレクトされ、次に 2 番目のコマンドが一時ファイルからリダイレクトされた入力を使用して実行されている... などと見なすことができます。

パイプラインの一般的な使用では、1 個以上のフィルタを一緒に接続することによって、コマンドの出力を前処理または後処理しています。標準入力から読み取り、標準出力に書き込むコマンドはフィルタと呼ばれます。たとえば、コマンド **grep** では、指定された文字列を含む 1 つ以上のファイル内の行を指定します。ヘッダーファイルディレクトリ `/usr/ch/include`, で型指定された型 **time_t** の定義を参照する場合は、パイプラインを含む次のようなコマンドを使用できます。

```
> pwd
/usr/ch/include
> grep time_t *.h | grep typedef
time.h:typedef int    time_t;
>
```

コマンド `grep time_t *.h` の出力は `grep typedef` にパイプされます。つまり、パイプライン `grep time_t *.h | grep typedef` では、ディレクトリ `/usr/ch/include` のヘッダーファイルに文字列 `time_t` と `typedef` の両方を含むすべての行を一覧表示します。無駄な出力を多く生成する場合がある `grep time_t *.h` または `grep typedef *.h` のみを使用するよりも効果的です。次のコマンドは、所有者が `myname` でコマンドが **vi** であるプロセスの状態のみを表示します。

```
> ps -elf | grep myname | grep vi
1 S     myname 20851 20850  0 156 20  19d0500  115  1086564 23:36:47
ttyp5 0:01 /bin/vi example.txt
>
```

コマンド `ps -elf` は、すべてのプロセスの状態を詳細に一覧表示します。その出力は、コマンド `grep myname` に入力としてパイプで渡されます。次に、コマンド `grep myname` で文字列 `myname` を含む行を表示します。これらの行は、コマンド `grep vi` に入力としてパイプで渡されます。パイプライン処理されたコマンドは、文字列 `myname` および **vi** の両方を含むすべてのプロセスの状態の行を一覧表示します。

もう 1 つの例として、圧縮済みの保存ファイル `file.tar.gz` を以下のコマンドで抽出できます。

```
gzip -cd file.tar.gz |tar -xvf -
```

コマンド **gzip** は、まずファイル `file.tar.gz` を解凍します。展開されたファイルは、次に、コマンド **tar** でファイルを抽出するために標準出力としてパイプで渡されます。許可モードおよびアクセス時間を変更することなく、ディレクトリ `/home/from` 内のファイルをディレクトリ `/home/new/from` にコピーするには、次のコマンドを使用できます。

```
tar cf - /home/from | (cd /home/new; tar xf - )
```

また、関数 **popen()** および **pclose()** を使用しても、 プログラム間で出力をパイプ処理できます。たとえば、Bourne シェルプログラムの **ed** コマンドの例を次に示します。

```
#!/bin/sh
ed testfile  <<END
a
input from the console command line
abcd
123456
.
w
q
END
```

このコマンドは、最初に出現する区切り行 `END` までの後続の行を入力として使用して、ファイル `testfile` を編集します。区切り記号として "`END`"を使用する必要はなく、任意の単語を使用できます。エディタ **ed** で、次に示す 3 行のテキストを含むファイルを追加します。

```
input from the console command line
abcd
123456
```

ed は行指向のテキストエディタです。Bourne シェルプログラムの **a**、'**.**'、**w** および **q** は、**ed** のコマンドです。コマンド **a** ("append") は、**ed** にテキストの収集を開始するように指示します。'**.**' はテキストの末尾を示すためのコマンドです。コマンド **w** ("write") は、`testfile` ファイルに情報を保存します。コマンド q **q** ("quit") はエディタを終了します。

前述の Bourne シェルプログラムの結果は、次のようにデータをパイプでプロセスに渡すことによって、Ch プログラムを使用して取得できます。

```
#!/bin/ch
#include <stdio.h>
FILE *fp;
string_t command_args="a\ninput from the console command line\
\nabcd\n123456\n.\nw\nq";
fp = popen("ed testfile", "w");
fwrite(command_args, strlen(command_args), sizeof(char), fp);
pclose(fp);
```

ここでは、前述の Bourne シェルプログラムのエディタ **ed** のコマンドおよび入力行を、**string_t** 型の変数 `command_args` によって置換しています。変数 `command_args` の文字列では、コマンドと入力行をまとめて改行文字 '`\n`' で区切っています。関数 **popen()** と **pclose()** は、コマンド `ed testfile` のプロセスにパイプで結合された入出力をそれぞれ開始、終了します。関数 **popen()** の最初の引数は、

シェルのコマンドライン `ed testfile` を含む文字列です。2 番目の引数 `w` は操作がパイプを使用してプロセスにデータを書き込むことを指示する入出力モードです。関数 **fwrite()** では、シェルコマンド `ed testfile` のプロセスのストリーム `fp` によって参照されるストリームにデータを送信します。

4.12 バックグラウンドでのコマンドの実行

既定では、次のプロンプトが表示される前に、コマンドモードの Ch シェルはコマンドの実行の完了を待機します。Ch シェルが待機するコマンドは、フォアグランドコマンドと呼ばれます。バックグラウンドコマンドは、Ch シェル内で非同期に起動されるプロセスです。バックグラウンドコマンドが完了する前に、Ch シェルは次のプロンプトを表示し、直ちにコマンドラインからの入力を受け入れます。プログラムモードでは、プログラムの制御フローはバックグラウンドコマンドが実行を完了する直前に次のステートメントに進みます。末尾が`&`であるコマンドまたはパイプラインは、バックグラウンドコマンドとして処理されます。リダイレクトしない場合、バックグラウンドコマンドの出力は端末に表示されます。

バックグラウンドコマンドは、実行に長い時間がかかるコマンドの処理に役立ちます。また、イベントドリブンのグラフィカルユーザーインタフェースでコマンドを起動する場合にも便利です。たとえば、Windows では、次のようにバックグラウンドコマンドとしてコマンド `notepad` を起動することができます。

```
> notepad &
```

4.13 実行時の式の評価

汎用関数 **streval()** を使用すると、文字列で表された式を実行時に評価できます。式は関数ファイルパス`_fpath` によって指定される関数ファイル内の関数を呼び出すことができます。この関数はポリモーフィックであり、右辺値としてのみ使用できます。次に例を示します。

```
> int i = 1
> float f = 10.1
> string_t s
> char a[10], *p
> i = streval("i*2")  /* i becomes 2 */
2
> s = "f*i"
f*i
> f = streval(s)      /* f becomes 20.20 */
20.20
> strcpy(a, s)
f*i
> strcat(a, "+5")
f*i+5
> f = streval(a)      /* f becomes 45.40 */
```

```
45.40
> p = a
f*i+5
> f = streval(p)       /* f becomes 95.80 */
95.80
> streval("23unknown")
Invalid argument for streval()
```

実行時に式を数値に変換できる場合、汎用関数 **strparse()** は 0 を返します。その他の場合はゼロ以外を返しますので次のように使うことができます。

```
int i =90, status;
status = strparse("i*2");
if(!status)  {
   i = streval("i*2");
   printf("i = %d\n", i);
}
```

一部のアプリケーションでは、引数 **streval()** の文字列値はコマンドインタフェースを介して渡されます。たとえば、プログラム `command` では、文字列 `x` と 10 か、`x+sin(90)*9` と `10` を取得することがあります。次のように、かっこをエスケープすることが必要である場合があります。

```
command x 10
command x+sin\(90\)*9 10
```

4.14 環境変数の処理

Ch の環境変数は、他の Unix シェルや MS-DOS シェルの環境変数と同様です。Ch には関数を処理する 4 つの環境変数があります。関数 **putenv()** ではシステムに環境変数を追加できます。 この関数は、システムのスタートアップファイル **CHHOME/config/chrc** およびユーザーのホームディレクトリにある個々のユーザーのスタートアップファイル **.chrc** (Unix) および **_chrc** (Windows) 内で一般的に使用されます。環境変数が与えられると、関数 **getenv()** は、対応する値を取得できます。 関数 **remenv()** では環境変数を削除できます。 関数 **isenv()** では、記号が環境変数であるかどうかをテストできます。 これらの関数を適用した対話的なコマンド実行を次に示します。

```
> putenv("ENVVAR=value")
0
> getenv("ENVVAR")
value
> isenv("ENVVAR")
1
> remenv("ENVVAR")
> isenv("ENVVAR")
```

```
0
>
```

この例では、コマンド putenv("ENVVAR=value") を使用して、環境変数 ENVVAR に値 value を設定しています。C では、等号 '=' に隣接する空白を挿入できないことに注意してください。その後、コマンド getenv("ENVVAR") は value、つまり環境変数 ENVVAR の値を取得します。ENVVAR は環境変数であるため、関数呼び出し isenv("ENVVAR") は 1 を返します。環境変数から ENVVAR を削除するために関数 **remenv()** を呼び出した後、関数呼び出し isenv("ENVVAR") は 0 を返します。

アプリケーションの例として、リモートマシンで環境変数 **DISPLAY** をローカルマシンの名前 local.domain.com に設定することにより、リモートマシン remote.domain.com が X ウィンドウを使用してネットワーク経由でマシン local.domain.com にグラフィック出力を送信できます。ローカルマシン local.domain.com で次のコマンド

```
> xhost server.domain.com
server.domain.com being added to access control list
>
```

を使用して、アクセス制御リストにリモートマシンを追加する必要があります。リモートマシン remote.domain.com で、次のコマンド

```
> putenv("DISPLAY=local.domain.com:0.0")
>
```

は環境変数 **DISPLAY** を設定して、サーバーがグラフィック出力を local.domain.com に送信するようにします。

セクション10.10で説明するプログラムを使用すると、すべての環境変数とその対応する値を出力できます。また、Unix では、システムコマンド **env** を使用してすべての環境変数を表示することもできます。

環境変数は、現在のシェル内で実行されるすべてのコマンドとプログラムに渡されます。Ch サブシェルは、親シェルから環境のコピーを継承します。一方で、サブシェル内の環境変数の値への変更は親シェルに影響しません。次に例を示します。

```
> putenv("ENVVAR=value")
0
> getenv("ENVVAR")
value
> cat > changeenv.ch
  #!/bin/ch
  printf("%s\n", getenv("ENVVAR"));
  putenv("ENVVAR=value2");
  printf("%s\n", getenv("ENVVAR"));
 (To complete the file and quit from the command cat,
  use Ctrl-D in Unix or Ctrl-Z in Windows)
> changeenv.ch
```

```
value
value2
> getenv("ENVVAR")
value
>
```

この例では、プログラム changeenv.ch は親シェルの ENVVAR を含むすべての環境変数のコピーを含むサブシェル内で実行されています。サブシェルの ENVVAR のコピーの値が value2 に変更されても、親シェルの ENVVAR の値は依然として value のままです。セクション4.4で説明しているように、ドットコマンドを使用すると現在のシェル内でプログラムを実行できます。次の例では、コマンド . changeenv.ch では、現在のシェルの環境変数 ENVVAR を変更します。

```
> getenv("ENVVAR")
value
> . changeenv.ch
value
value2
> getenv("ENVVAR")
value2
>
```

セクション4.7で説明しているように、対話型の Ch シェルのコマンドラインでは、環境変数の値とプログラム内のコマンドステートメントは、変数置換を使用して取得できます。次に示すように $ENVVAR または$(getenv(ENVVAR)) のいずれかを使用して、環境変数 ENVVAR の値を取得できます。

```
> getenv("ENVVAR")
value
> echo $ENVVAR
value
> echo $(getenv("ENVVAR"))
value
>
```

付録 Dで説明しているように、シェルコマンド setenv を使用すると C シェル内の環境変数を設定できます。sh、bash、および ksh シェルでは、シェルコマンド export を使用して設定できます。Ch の次のコマンドを例として示します。

```
putenv("DISPLAY=local.domain.com:0.0")
```

このコマンドは、C シェルでは次のように処理される必要があります。

```
setenv DISPLAY local.domain.com:0.0
```

sh、bash、および ksh シェルでは、このコマンドは次のように設定できます。

```
DISPLAY=local.domain.com:0.0
export DISPLAY
```

4.15 汎用 Ch プログラム

表4.12に示したコマンドは、汎用 Ch プログラムです。

表 4.12: 汎用 Ch プログラム

コマンド	説明
ch	Ch シェル
dirs	dir スタック内のディレクトリを一覧表示します。C シェルと互換性があります。
help	Ch の使い方
popd	dir スタックの最初のディレクトリを取り出し、そのディレクトリに切り換えます。C シェルとの互換性があります。
popd +*n*	dir スタックの *n* 番目のディレクトリを取り出し、そのディレクトリに切り換えます。C シェルとの互換性があります。
pushd	スタック内の 2 番目のディレクトリに切り換えます。C シェルとの互換性があります。
pushd *dirname*	ディレクトリ *dirname* を dir スタックに入れ、そのディレクトリに切り換えます。C シェルとの互換性があります。
pushd +*n*	スタック内の *n* 番目のディレクトリに切り換えます。C シェルとの互換性があります。
chs	セーフ Ch シェル
which [-a]	C シェルの **which** と同じです。トークンの位置を示すために使用します。

コマンド **ch** または **chs** は、通常の Ch またはセーフ Ch を新たに起動するためにそれぞれ使用されます。コマンド **which** は、指定された実行可能プログラムがある場所を表示する場合に使用します。C シェルから流用されたコマンド **dirs**、**pushd**、および **popd** では、複数の作業ディレクトリ間の切り替えができる便利なディレクトリスタックを管理できます。ディレクトリを変更する **cd** の代わりにこれらのコマンドを使用できます。コマンド **pushd** は、スタックの最上部にある 2 つのディレクトリ間を移動するために使用します。コマンド **pushd** `dirname` は `dirname` をスタックに入れ、現在の作業ディレクトリをそのディレクトリに切り替えます。コマンド **popd** `+n` では、現在の作業ディレクトリをスタック上の `n` 番目のディレクトリに変更します。

以下のコマンドは、 Ch シェル内でこれらの汎用プログラムをどのように使用できるかを示しています。

```
> which ch
/usr/ch/bin/ch
> which ch ls
/usr/ch/bin/ch
ls is aliased to ls -F
> which -a ls stdio TERM
ls is aliased to ls -F
/bin/ls
/usr/bin/ls
```

```
/usr/ucb/ls
/usr/ch/include/stdio.h
dtterm
> pwd
/usr/ch
> pushd /usr/ch
0  /usr/ch
> pwd
/usr/ch
> pushd /home/myname
0  /home/myname
1  /usr/ch
> pwd
/home/myname
> pushd
0  /usr/ch
1  /home/myname
> pwd
/usr/ch
> pushd
0  /home/myname
1  /usr/ch
> pwd
/home/myname
> dirs
0  /home/myname
1  /usr/ch
> popd
0  /usr/ch
> dirs
0  /usr/ch
>
```

この例では、まず **which** コマンドが **ch** コマンドの実行可能プログラムの場所を指示します。コマンド **ch** および **ls** で示したように、このコマンドは複数のコマンドを処理できます。この場合、**ls** は別名です。オプション`-a` を使用すると、別名、定義済みの識別子 **_path** で指定されたパスにある実行可能プログラム、**_fpath** が参照するパスにある関数ファイル、環境変数、**_ipath** が参照するパスにあるヘッダーファイルのすべてが表示されます。オプション`-a` を指定しない場合は、使用可能な最初のコマンド、関数ファイル、ヘッダーファイルまたは環境変数の値のみが表示されます。コマンドが見つからない場合、オプション`-v` を指定すると、**_path** のすべてのパスが表示されます。上の例では、**ls** の別名だけなく、パスの使用可能な実行可能コマンドがすべて一覧表示されます。記号 `stdio` はファイル拡張子`.h` を持つヘッダーファイル `stdio.h` です。環境変数 `TERM` の値は `dtterm` です。2 つの作業ディレクトリをスタックに入れた後、コマンド **pushd** を使用して 2 つのディレクトリ間を切

り替えます。

表4.12に示したコマンドは、通常の Ch シェルでのみアクセス可能です。セーフ Ch シェルではアクセスできません。ただし、Ch シェルはセーフ Ch シェルを使用して呼び出すこともできます。

4.16 シェルプログラミング

4.16.1 プログラムでのシェルコマンドの使用

C シェルとは異なり、Ch の構文と制御フローは C と互換性があります。C プログラムは容易に Ch シェルで実行できます。ただし、プログラムの実行速度が大きな問題でないときは、多くの場合、Ch プログラムでシェルコマンドを使用する方が便利です。C コードでは何千行も必要なタスクを、Ch コードでは、既存のコマンドに基づいて、数行のみで実現できる場合もあります。付録Gでは、異なるプラットフォーム間で移植可能なシェルプログラミングのために Ch でよく使用するコマンドの一覧を記載しています。

Ch シェルスクリプトは通常、関数 `main()` または、Windows での `WinMain()` を含みません。Ch シェルでは、システム変数 **_pathext** に指定された名前の拡張子を持つファイルは、ファイルの内容にかかわらず Ch スクリプトファイルとして処理されます。その他のファイルの場合、 Ch は内容を分析します。通常、スクリプトファイルがどのシェル用に記述されているかはファイルの最初の行で示されます。表4.13に、一般的な Unix シェルの最初の行を示します。

表 4.13: さまざまなシェルでのシェルプログラムの最初の行

シェル	最初の行
Ch シェル	`#!/bin/ch`
C シェル	`#!/bin/csh`
Bourne シェル	`#!/bin/sh`
Korn シェル	`#!/bin/ksh`
BASH	`#!/bin/bash`

プログラムの最初の行にステートメント `#!/bin/ch` が含まれる場合、そのプログラムは Ch シェルスクリプトとして処理されます。Ch スクリプトは C シェルや Bourne シェルなどの他のシェルでも実行できます。シェルスクリプトのファイル拡張子がシステム変数 **_pathext** に含まれている場合、スクリプトは、最初の行で Ch プログラムではないことを示している場合でも Ch シェルとして処理されます。この場合、プログラムは Ch プログラムとして正常に実行されない可能性があります。

プログラムのファイル拡張子がシステム変数 **_pathext** に含まれておらず、プログラムの先頭ステートメントが `#!/bin/ch` またはセクション3.3で説明した他のトークンでない場合、Ch はそのプログラムを実行するために他のシェルを呼び出します。

Ch プログラム内で呼び出されたコマンドは、コマンドステートメントと呼ばれます。`cmd` などの**コマンドステートメント**の制約は次のとおりです。

- 有効な識別子または `cmd.ext` などのファイル拡張子を含む識別子である必要があります。

- スコープ内で宣言されている変数ではない必要があります。ただし、コマンド cmd の場合で言えば次に示すように、絶対パスまたは相対パスが前に付いている場合を除きます。

```
/path/cmd
./cmd
../cmd
~/cmd
```

- コマンド名がスコープ内で宣言されている変数でもある場合は、そのコマンドを二重引用符で囲むことができます。コマンドのオプションを引用符内に指定することはできません。また、コマンドに空白が含まれる名前をもつディレクトリ内にある場合にも使用できます。次に例を示します。

```
int ls = 10;
"ls" -1
"/ch/bin/echo" option $PATH
"C:/Program Files/Windows NT/Accessories/wordpad.exe"
```

- コマンドはコマンド名置換を使用して実行時に動的に取得できます。コマンド名置換用の $ 記号の後に続く変数または式のデータ型は、文字列、char へのポインタまたは符号なし char 型である必要があります。次に例を示します。

```
string_t s = "cmd";
$s option
```

  この場合、記号 cmd をコマンド名とは別の変数名として使用することもできます。

- これは、次のようにドット '.' の後に二重引用符で囲んで指定する必要があります。

```
. "cmd"
```

  この場合、コマンド cmd は、プリプロセッサディレクティブ include によって組み込まれた場合と同様に、現在のシェルで実行されます。ただし、**_ipath** ではなくシステム変数 **_path** によってプログラムが検索される点は異なります。以下はこれと同様です。

```
#pragma import "cmd"
```

Windows では、たとえば cmd.ch などのシェルスクリプトが Makefile などの他のプログラムで使用されている場合、次のように Ch によって呼び出される必要があります。

```
ch cmd.ch
```

または

```
ch cmd
```

コマンド実行のこれら 2 つの形式では、スクリプトファイル cmd.ch を実行するために Ch シェルを明示的に起動します。

コマンドステートメントで使用されている変数に、セクション4.7で説明している変数置換を適用することができます。他のプログラムステートメントに必要な末尾を表すセミコロンは、コマンドステートメントには必要ありません。たとえば、シェルスクリプト cpfile1.ch に次のステートメントが含まれているとします。

```
#!/bin/ch
cp /dir/source/*.ch /dir/dest/
```

先頭が Unix コマンド cp であるコマンドステートメントの末尾を表すセミコロンがありません。このコマンドでは、最初の引数で指定したファイルを 2 番目の引数で指定されたディレクトリにコピーします。このシェルスクリプトを

```
> cpfile1.ch
>
```

によって実行すると、ディレクトリ/dir/source にある拡張子.ch を持つすべてのファイルが、ディレクトリ/dir/dest にコピーされます。

セクション8.4.4で説明する **foreach** ループは、シェルプログラミングで非常に役立ちます。たとえば、以下のプログラムは現在のディレクトリのすべてのファイルの名前を出力します。現在のディレクトリ内のファイルの名前をコマンド ls で取得し、foreach ループによって並べ替えます。

```
#!/bin/ch
string_t file, files = `ls ./`;
foreach(file; files) {
  printf("file = %s\n", file);
}
```

第20章で説明するように、プログラムの出力は **fopen()**、**fclose()**、**fprintf()** などの入出力関数によって処理できますが、多くの場合、シェルコマンドを使用してプログラムの出力をファイルに送る方が便利です。たとえば、プログラム4.2では、

現在のディレクトリで読み取り許可があるファイルの名前をファイル newfile に保存します。ファイル newfile が既に存在している場合は、最初に関数 **remove()** で削除されます。file が存在しているかどうかは、access(file, F_OK). の関数呼び出しでテストされます。

ヘッダーファイル **unistd.h** で定義されている関数 **access()** は、最初の引数で参照しているパス名を使用して、指定されたファイルにアクセスできるかどうかを確認します。実効ユーザー ID の代わりに実際のユーザー ID を使用し、実効グループ ID の代わりに実際のグループ ID を使用すると、setuid プロセスにより、このプロセスを実行しているユーザーにこのファイルにアクセスする許可があるかどうかを確認できます。**int** 型である 2 番目の引数の値は、確認対象のアクセス許可 (**R_OK**、**W_OK**、および **X_OK**) のビットごとの包含 **OR**、または存在テスト (**F_OK**) のいずれかです。これらのシンボリック定数については表4.14で説明します。

表 4.14: 関数 **access()** のシンボリック定数

定数	説明
R_OK	読み取りアクセス許可をテストします。
W_OK	書き込みアクセス許可をテストします。
X_OK	実行または検索アクセス許可をテストします。
F_OK	ファイルの有無をテストします。

要求したアクセスが許可される場合は、ゼロが返されます。それ以外の場合は、-1 が返され、エラーを示す errno が設定されます。

プログラム4.2では、file が読み取り可能であるかどうかを access(file, R_OK). の関数呼び出しによって確認します。また、出力リダイレクトを伴うコマンド **echo** により、ファイル名がユーザーのホームディレクトリにあるファイル allfiles に追加されます。

```
#!/bin/ch
#include <stdio.h>
#include <unistd.h>
string_t file, files = `ls ./`;
string_t newfile="newfile";
string_t allfiles= stradd(_home, "/allfiles");
int i;

if (access(newfile, F_OK) == 0) // clear up first
    remove(newfile);
foreach(file; files) {
    if (access(file, R_OK) == 0) {
       i++;
       echo $i $file >> $newfile
       echo $i $file >> $allfiles
    }
}
```

プログラム 4.2: シェルスクリプトを使用する入出力の処理

ファイル名の前には連番が付きます。たとえば、現在のディレクトリが file1、file2、file3 を含む場合、プログラム 4.2 を実行した出力ファイルは次のようになります。

```
1 file1
2 file2
3 file3
```

これらのファイルは、現在のディレクトリ内のファイル newfile にリダイレクトされ、同時にユーザーのホームディレクトリにあるファイル allfiles に追加されます。

シェルプログラミングの場合、セクション20.9では関数 **stat()** とディレクトリ処理関数を使用して、ディレクトリおよびそのサブディレクトリ内にあるファイルのサイズ、アクセス時間、ユーザー ID などの詳細情報を再帰的に取得します。

Ch スクリプトから実行可能プログラムを直接使用できます。プログラム4.3に示したように、sed、awk、wc、grep などのシェルコマンドを C コードと組み合わせて Ch で実行できます。

```
#!/bin/ch
string_t result1;
char result2[50];

grep "test" myfile.txt;
if ( _status == 0 ) {
    printf("'test' is found in myfile.txt\n");
}
else {
    printf("Cannot find 'test' in myfile.txt\n");
}
result1=`wc -l /etc/passwd`;
echo $result1;
strcpy(result2, `wc -l /etc/passwd`);
printf("%s\n", result2);
```

プログラム 4.3: C コードとシェルコマンドの組み合わせ

プログラム4.3の `tnum1` の出力と `tnum2` の出力は同じです。

4.16.2 シェルコマンドへの値渡し

このセクションでは、コマンドライン引数の値を Ch シェルプログラムに渡す方法を説明します。Ch シェルプログラムでは、定義済みの 2 つの識別子**_argc** と**_argv** を使用して、コマンドラインの引数を処理します。これらの引数は、次のように **int** および **char ****の型を使用して内部的に定義されます。

```
int    _argc;
char* _argv[];
```

識別子**_argc** は、コマンド名自体を含むコマンドラインの引数の数を示します。識別子**_argv** はコマンドラインの引数リストを維持します。シェルは**_argv[0]** にコマンド名、**_argv[1]** に最初の引数などを格納します。たとえば、ファイル `argtest.ch` に次のステートメントが含まれているとします。

```
#!/bin/ch
echo $_argc
echo $(_argv[0])
echo $(_argv[1])
printf("%s\n", _argv[2]);
printf("%s\n", _argv[3]);
```

`argtest.ch` の実行は次のとおりです。

```
> argtest.ch -option arg2
3
argtest.ch
-option
arg2
(null)
>
```

この例では、_**argc** の値は 3 で、ファイル名 `argtest.c` および 2 つの引数 `arg1` と `arg2` を含みます。ファイル名は変数_**argv[0]** に格納され、最初の引数 `arg1` は_**argv[1]** に、2 番目の引数 `arg2` は_**argv[2]** に格納されます。コマンドラインでの引数へのアクセスに関する C シェルと Ch の比較を付録 Dに記載しています。

プログラム4.4は、_**argc** および_**argv** を使用するコマンドライン引数の処理例です。これを使用すると、**which** コマンドを実装できます。

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

int main() {
  int i = 0;            // for index of arguments
  int j = 0;            // for index of characters in arguments
  char c;
  int a_option = false; // default, no -a option
  int v_option = false; // default, no -v option

  if(_argc == 1){         // no argument
    fprintf(stderr, "Usage: which [-av] names \n");
    exit(1);
  }

  _argc--; i++; j = 0;
  while(_argc > 0 && _argv[i][j++] == '-') {
      // for every argument beginning with -
      // empty space is not valid option
      for(c = _argv[i][j++]; c&&c!=' '; c = _argv[i][j++]) { // for -av
        switch(c)
        {
          case 'a':
            a_option = true;      // get all possible matches
            break;
          case 'v':
            v_option = true;      // print message
            break;
          default:
            fprintf(stderr,"Warning: invalid option %c\n", c);
            fprintf(stderr, "Usage: which [-av] names \n");
            break;
        }
      }
      _argc--; i++; j = 0;
  }

  if(a_option)
    printf("option -a is on\n");
  if(v_option)
    printf("option -v is on\n");
  while(_argc > 0) { // print out the remaining arguments
      printf("%s\n",  _argv[i]);
      _argc--; i++;
  }
  return 0;
}
```

プログラム 4.4: **_argc** および **_argv** を使用するコマンドライン引数の処理

ここでは、変数 a a_option および変数 v_option は、有効なオプション-a および-v がオンになっているかどうかを示します。これらの値は、既定では false です。コマンドライン引数が指定されていない場合、プログラムはエラーメッセージを出力します。これは、コマンド **which** には少なくとも 1 つの引数、つまり検索対象の名前が必要であるためです。このプログラムの while ループで

は、マイナス記号−で始まるすべての引数を処理します。For ループでは 1 文字ずつこれらの引数文字を分析します。次のステートメント

```
c = _argv[i][j++];
```

では、変数 c を引数_argv[i] 内の j 番目の文字と同じにします。これらの引数に文字 'a' および 'v' がある場合は、変数 a_option と v_option はそれぞれ true に設定されます。他の文字がある場合は、エラーメッセージが出力されます。プログラム4.4の最後には、オプションと残りのコマンドライン引数が出力されます。プログラム4.4のファイル名が commandline.ch であるとして、異なるオプションを指定した場合のプログラム4.4の実行結果を次に示します。

```
> commandline.ch -a -v arg1
option -a is on
option -v is on
arg1
> commandline.ch -av arg1
option -a is on
option -v is on
arg1
> commandline.ch -v arg1 arg2
option -v is on
arg1
arg2
```

ポインタへのポインタを伴うコマンドライン引数を処理する同様のプログラムをセクション10.10に記載しています。

プログラム cpfile2.ch が次のステートメントを含むとします。

```
#!/bin/ch
cp /dir/source/$(_argv[1]) /dir/dest/
```

プログラム cpfile2.ch を使用するとディレクトリ/dir/source にあるファイルをディレクトリ /dir/dest に簡単にコピーできます。たとえば、次のコマンド

```
> cpfile2 file1
> cpfile2 file2
>
```

は、ディレクトリ/dir/source のファイル file1 と file2 をディレクトリ/dir/dest にコピーします。

第5章　プリプロセッサディレクティブ

現在の実装では、Ch はインタープリタです。プリプロセッサなどの独立した翻訳段階はありません。しかし、Ch では、C のプリプロセッサディレクティブの構文がサポートされています。Ch では、便宜上、これらもプリプロセッサディレクティブと呼びます。

プリプロセッサディレクティブは、#のプリプロセッサトークンで始まるプリプロセッサトークンのシーケンスで構成されます。表5.1に、プリプロセッサディレクティブの一覧を示します。本章では、これらのディレクティブに関する詳細を説明します。

表 5.1: プリプロセッサディレクティブ

ディレクティブ	説明
#define	プリプロセッサマクロを定義します。
#elif	先行する#if、#ifdef、#ifndef、または#elif の判定結果が不合格である場合、別の式の値に基づいて、代わりになるテキストを組み込みます。
#else	先行する#if、#ifdef、#ifndef、または#elif の判定が不合格である場合、代わりになるテキストを組み込みます。
#endif	条件付きテキストを終了します。
#error	指定されたメッセージを使用してコンパイル時エラーを生成します。
#if	式の値に基づいて条件付きでテキストを組み込みます。
#ifdef	マクロ名が定義されていれば、テキストを組み込みます。
#ifndef	マクロ名が定義されていなければ、テキストを組み込みます。
#include	別のソースファイルからテキストを挿入します。
#line	メッセージに行番号を付与します。
#pragma	C 標準にない Ch 固有の機能。
#undef	プリプロセッサマクロの定義を削除します。
#	パラメータの値を含む文字列定数でマクロパラメータを置き換えます。
##	隣接する 2 つのトークンから単一のトークンを作成します。
#	Null ディレクティブ。
defined	名前がプリプロセッサマクロとして定義されている場合は 1 に評価され、それ以外の場合は 0 に評価されるプリプロセッサ演算子。#if および#elif で使用されます。

5.1　条件付き組み込み

次のような形式のプリプロセッサディレクティブがあります。

```
# if   expr1
# elif expr2
```

このディレクティブは、制御式が非ゼロに評価されるかどうかをチェックします。条件付きの組み込みを制御する式は、整数式でなければなりません。ただし、宣言された識別子を含むことはできません。次のようなプリプロセッサ演算を含むことができます。

```
defined (identifier)
```

これは、識別子がマクロ名として現在定義されている場合は 1 に評価され、そうでない場合は 0 に評価されます。識別子がマクロ名として定義されている場合とは、つまり、識別子が事前定義されているか、識別子が同じ処理対象の識別子を持つ**#undef** ディレクティブを介さずに**#define** プリプロセッサディレクティブの処理対象になっていた場合です。Ch では、C のプリプロセッサディレクティブが拡張されています。すべての演算子と、strcat()、strcmp()、access() などの汎用関数は、Ch のプリプロセッサディレクティブ`#if` および`#elif` の引数で使用できます。次に例を示します。

```
#if defined(_HPUX_)
  printf("I am using HP-UX\n");
#elif !strcmp(`uname`, "Linux")
  printf("I am using Linux\n");
#endif
```

次のような形式のプリプロセッサディレクティブがあります。

```
# ifdef  identifier
# ifndef identifier
```

このディレクティブは、識別子がマクロ名として現在定義されているかどうかをチェックします。これらは、それぞれ、**#if defined** *(identifier)* および**#if !defined** *(identifier)* に相当します。

各ディレクティブの条件は順番にチェックされます。false (ゼロ) と評価された場合、その条件が制御するグループはスキップされます。入れ子になった条件のレベルを管理するために、ディレクティブはディレクティブを決定する名前を通してのみ処理されます。ディレクティブの残りのプリプロセッサトークンは無視されます。グループ内の他のプリプロセッサトークンも同様に無視されます。制御条件が true (非ゼロ) に評価された最初のグループのみが処理されます。どの条件も true に評価されず、**#else** ディレクティブが指定されている場合は、**#else** で制御されるグループが処理されます。**#else** ディレクティブが指定されていない場合は、**#endif** までのすべてのグループがスキップされます。

5.2 ソースファイルの組み込み

#include ディレクティブは、Ch インタープリタで処理できるヘッダーまたはソースファイルを特定します。次のような形式のプリプロセッサディレクティブがあります。

```
#include <h-char-sequence>
```

これは、区切り文字 < と > ではさんで指定したシーケンスによって一意に識別されるヘッダーを検索し、ヘッダーの内容全体でディレクティブを置き換えます。ヘッダーは、文字列型の事前定義済み識別子 **_ipath** に指定したパスに従って、検索されます。各パスはセミコロンで区切ります。既定で

は、変数_**ipath** には文字列"CHHOME/include;CHHOME/toolkit/include;"が設定されています。CHHOME は Ch ソフトウェアのホームディレクトリです。検索パスの変数_**ipath** は、通常、ユーザーのホームディレクトリにあるスタートアップファイル**.chrc** (Unix の場合) および_**chrc** (Windows の場合) で設定されています。

次のような形式のプリプロセッサディレクティブがあります。

```
#include "q-char-sequence"
```

このディレクティブは、区切り文字"ではさんで指定したシーケンスによって識別されるソースファイルの内容全体でディレクティブを置き換えます。指定した名前のソースファイルは、最初に現在のディレクトリで検索され、次にシステム変数_**ipath** に指定したディレクトリで検索されます。

次の形式のプリプロセッサディレクティブがあります。

```
#include pp-tokens
```

このディレクティブは、前述した 2 つの形式のいずれにも一致しませんが、使用できます。ディレクティブで **include** の後にあるプリプロセッサトークンは、単に通常のテキストと同様に処理されます。マクロ名として現在定義されている個々の識別子は、プリプロセッサトークンの置換リストによって置き換えられます。**全置換の後**に作成されたディレクティブは、前述の 2 つの形式のいずれかに一致します。プリプロセッサディレクティブ**#include** は、別のファイルの**#include** ディレクティブによって読み取られたソースファイルに出現する場合があります。**#include** ディレクティブの入れ子のレベルに制限はありません。

#include プリプロセッサディレクティブの最も一般的な使い方を以下に示します。

```
#include <stdio.h>
#include "myprog.h"
```

以下のコード例は、マクロ置換された#include ディレクティブを示しています。

```
#if VERSION == 3
    #define INCFILE  <version3.h>
#elif VERSION == 2
    #define INCFILE  <version2.h>
#else
    #define INCFILE  <version1.h>
#endif
#include INCFILE
```

5.3 マクロ置換

次のような形式のプリプロセッサディレクティブがあります。

```
#define identifier replacement-list new-line
```

このディレクティブは、オブジェクトに似たマクロを定義します。プログラムの中でその後に現われたマクロ名は、ディレクティブの残りの部分を構成するプリプロセッサトークンの置換リストによって置き換えられます。改行は、**#define** プリプロセッサディレクティブを終了する文字です。

#define の直後の識別子は、**マクロ名**と呼ばれます。**マクロ名**の後には、置換リストと呼ばれるトークンの並びが続きます。2 つの置換リストは、両リストのプリプロセッサトークンが同じ数値、スペル、および空白区切りを持つ場合に限り、同一です。このとき、すべての空白区切りは同一であると見なされます。

単純な形式のマクロは、名前付き定数をプログラムで使用する場合、特に便利です。表の長さなどの数値を 1 箇所だけに記述し、それを他の場所から名前で参照できます。そのため、後から数値を簡単に変更できます。たとえば、マクロ

```
#define BLOCK_SIZE 0x100
```

を定義すると、

```
int size = BLOCK_SIZE;
```

のように記述できるため、次の記述よりも意味がわかりやすくなります。

```
int size = 0x100;
```

次のような形式のプリプロセッサディレクティブがあります。

```
#define identifier( identifier-list-opt ) replacement-list new-line
```

このディレクティブは、関数に似た、引数を使用するマクロを定義します。構文的には関数の呼び出しに似ています。パラメータは、省略可能な識別子リストによって指定します。パラメータのスコープは、識別子リスト内での宣言から、**#define** プリプロセッサディレクティブを終了する改行文字までです。プログラムの中でその後に現われた、関数に似たマクロ名は、それに続く左かっこ'(' と共に次のプリプロセッサトークンとして、定義にある置換リストで置き換えられるプリプロセッサトークンの並びを開始します (マクロの呼び出し)。置換されたプリプロセッサトークンの並びは、対になる閉じかっこ')' のあるプリプロセッサトークンで終了し、間に入る左右対になったかっこのプリプロセッサトークンをスキップします。関数に似たマクロの呼び出しを構成するプリプロセッサトークンの並び内では、改行は通常の空白文字と見なされます。

たとえば、引数を 2 つ持つマクロ mul が

```
#define mul(x,y) ((x)*(y))
```

のように定義されている場合、ソースプログラムの行

```
result = mul(5, a+b);
```

は次のように置き換えられます。

```
result = ((5)*(a+b));
```

マクロの定義では、かっこが重要であることに注意してください。マクロ mul() が、

```
#define mul(x,y) x*y
```

のようにかっこなしで定義されている場合、ステートメント

```
result = mul(5, a+b);
```

は次のようになりなります。

```
result = 5*a+b;
```

変数の引数リストを持つマクロは、引数内で省略記号を使用します。置換リストに出現する識別子**__VA_ARGS__**は、パラメータのように扱われ、変数の引数が置換に使用されるプリプロセッサトークンを形成します。たとえば、次のようなコード例があるとします。

```
#define debug(...)    printf(__VA_ARGS__)
#define debug2(fp, ...)   fprintf(fp, __VA_ARGS__)
debug("x = %d\n", x);
debug2(stderr, "x = %d\n", x);
```

結果は、次のようになります。

```
printf("x = %d\n", x);
fprintf(stderr,  "x = %d\n", x);
```

5.4 トークンから文字列への変換

マクロ定義の中に現れる#トークンは、単項式の文字列化演算子として認識されます。置換リストでパラメータが#プリプロセッサトークンの直後にある場合、該当する引数のプリプロセッサトークンの並びのスペルを含む、単一文字の文字列リテラルプリプロセッサトークンによって両方が置き換えられます。次に例を示します。

```
> #define TEST(a) #a
> printf("%s",TEST(abcd))
abcd
>
```

マクロパラメータ abcd は、文字列定数"abcd"に変換されました。

引数のプリプロセッサトークンの間に出現する個々の空白は、文字列リテラル内では単一の空白文字になります。引数を構成している最初のプリプロセッサトークンより前にある空白と、最後のプリプロセッサトークンより後にある空白は削除されます。それ以外の場合には、引数に含まれる個々のプリプロセッサトークンの元のスペルが、文字列リテラル内に保持されます。文字列リテラルと文字定数のスペルでは、文字\が、文字定数または文字列リテラルの個々の"および\文字の前に挿入され(区切りの"文字を含む)、特別に処理されます。

次に例を示します。

```
> #define TEST(a) #a
> printf("1%s2",TEST(  a    b  ))
1a b2
> printf("1%s2\n", TEST( a\\b ))
1a\b2
> printf("1%s2\n", TEST(" a \\ b "))
1" a \\ b "2
>
```

ここでは、引数が文字列定数 “a b”に変更されています。a の前と b の後の空白は削除され、a と b の間の連続する空白は、単一の文字に置き換えられます。

5.5 マクロ拡張で結合するトークン

Ch の新しいトークンを形成するトークンの結合は、マクロ定義の結合演算子##の存在によって制御されます。オブジェクトに似たマクロの呼び出し、および関数に似たマクロの呼び出しの両方で、置換対象である追加のマクロ名が置換リストにあるかどうかを再検査する前に、置換リスト (引数からではない) にある##プリプロセッサトークンが削除され、先行するプリプロセッサトークンが後続のプリプロセッサトークンと連結されます。新しいトークンは、関数名、変数名、データ型名、キーワード名のいずれかであってもかまいません。または、別のマクロの名前である場合があり、その場合は展開されます。

連結の一般的な使用目的は、2 つの名前を連結してより長い名前にすることです。‘1.5’ や ‘e3’ のように 2 つの数字や 1 つの数字と名前を連結して、1 つの番号にすることもできます。また、連結によって、‘+=’ のように複数の文字の演算子を形成できます。次に例を示します。

```
> #define CONC2(a, b) a ## b
> #define CONC3(a, b, c) a ## b ## c
> CONC2(1, 2)
12
> CONC3(3, +, 4)
7
>
```

マクロ CONC2(1, 2) は 2 つの数字 1 および 2 を連結して 12 にし、CONC2(3, +, 4) はこれら 3 つの引数を連結して 3+4 とし、Ch コマンドラインに 7 を出力します。

Ch は、マクロが調べられるよりも前に、コメントを空白に変換します。“/* コメント文字列 */” の文字列はすべて、数個の空白として解釈されます。コメントは最初に空白に変換され、その後連結処理で破棄されるため、マクロ定義または連結される実際の引数内の”##”の横にコメントを使用することができます。次に例を示します。

```
> #define CONC2(a, b) a ## b
> CONC2(1, /*this is a comment */2)
12
>
```

2 番目の引数内のコメントは、連結時に破棄されます。
##プリプロセッサトークンは、どの形式のマクロ定義においても、置換リストの先頭または末尾に記述することはできません。

5.6 行の制御

#line ディレクティブを使用して、ソースコードに割り当てられる行番号を変更できます。このディレクティブは後続の行に新たな行番号を付与し、それはその後、これに続く行の行番号を得るためにインクリメントされます。このディレクティブは、プログラムのソースファイルに対して新規のファイル仕様を指定することもできます。これは、他のプログラムによって Ch コードへとプリプロセッサされた元のソースファイルを参照する際に便利です。
次の形式のプリプロセッサディレクティブがあります。

```
#line digit-sequence new-line
```

このディレクティブは、後続のソース行の行番号を、数字文字列 (10 進数の整数として解釈される) によって指定される行番号を持つソース行で始まる場合と同様に動作させるための実装を行います。数字文字列では、ゼロおよび 2147483647 を超える数字は指定できません。行番号は、事前定義されたマクロ **__LINE__**に内部的に格納されます。
次の形式のプリプロセッサディレクティブがあります。

```
#line digit-sequence "s-char-sequence-opt" new-line
```

このディレクティブは、前のディレクティブと同様に行番号を指定された値に設定し、かつソースファイルの名前を指定された文字列に設定します。ソースファイルの名前は、事前に定義されたマクロ **__FILE__**に内部的に格納されます。
ファイル名 `pre_line.c` を使用する以下のプログラムを例に示します。

```
int main () {
   printf("before line directive, line number is %d \n", __LINE__);
   printf("the FILE predefined macro = %s\n", __FILE__);
   #line 200 "newname"
   printf("after line directive, line number is %d \n", __LINE__);
   printf("the FILE predefined macro = %s\n", __FILE__);
   return 0;
}
```

出力は次のようになります。

```
before line directive, line number is 2
the FILE predefined macro = pre_line.c
after line directive, line number is 200
the FILE predefined macro = newname
```

5.7 エラーディレクティブ

次の形式のプリプロセッサディレクティブがあります。

```
#error pp-tokens-opt new-line
```

このディレクティブは、指定したプリプロセッサトークンの並びを含む診断メッセージを生成し、解釈を停止するための実装を行います。
たとえば、以下のプログラム pre_err.c

```
int main () {
#error from preprocessing error directive
   printf("after error directive\n");
   return 0;
}
```

が Ch で実行されると、出力は次のようになります。

```
ERROR: #error:  from preprocessing error directive
ERROR: syntax error before or at line 2 in file pre_err.c
  ==>: #error from preprocessing error directive
  BUG: #error from preprocessing error directive <== ???
WARNING: cannot execute command 'pre_err.c'
```

5.8 NULL ディレクティブ

次の形式のプリプロセッサディレクティブがあります。

```
#new-line
```

このディレクティブはプログラムに影響しません。行は無視されます。

5.9 プラグマディレクティブ

次の形式のプリプロセッサディレクティブがあります。

```
# pragma pp-tokens-opt new-line
```

このディレクティブはプラグマディレクティブと呼ばれます。C 標準では、プラットフォームに依存する機能を実装する手段として`#pragma` を定義しています。C 標準では、ディレクティブ内でどのマクロ置換にも先んじてプリプロセッサトークン STDC が pragma の直後にない場合、つまり (`#pragma STDC`) でディレクティブが始まっていない場合、実装定義の機能を追加できます。Ch では、特別な機能を実装するために、いくつかの`#progma` ステートメントを定義しています。Ch で定義されている、C 標準に準拠したプラグマディレクティブのプリプロセッサトークン名を表5.2に示します。

次の例では、変数 `var1` と var2 を最初に int として宣言します。後で変数を削除し、異なるスコープで float として再宣言します。

表 5.2: 有効なプラグマ

プラグマ名	数の値
`exec expr`	解析時に式を実行します。
`remvar(`*arg*`)`	グローバル変数または最上位変数 *arg* を削除します。
`remkey(`*arg*`)`	キーワード *arg* を削除します。
`import "`*filename*`"`	*filename* で指定したファイルを取り込みます。最初に現在のディレクトリでファイルを検索します。次に、**_path** で指定されたディレクトリを検索します。
`import` <*filename*>	*filename* で指定したファイルを取り込みます。**_path** で指定したディレクトリのみでファイルを検索します。
`importf "`*filename*`"`	filename で指定したファイルを取り込みます。最初に現在のディレクトリでファイルを検索します。次に、**_fpath** で指定されたディレクトリを検索します。
`importf` <*filename*>	*filename* で指定したファイルを取り込みます。**_fpath** で指定したディレクトリのみでファイルを検索します。
`pack()`	構造体フィールドの自動配置。
`pack(pop)`	構造体フィールドの自動配置。
`pack(push,` *n*`)`	構造体の *n* バイトのパッキングをオンにします。
`pack(`*n*`)`	構造体の *n* バイトのパッキングをオンにします。
`package` <*pname*>	`$(_ppath)/pname/bin` を **_path** に、`$(_ppath)/pname/lib` を **_fpath** に `$(_ppath)/pname/include` を **_ipath** に、`$(_ppath)/pname/dl` を **_lpath** に追加します。
`package "`*/u/dir/pname*`"`	`/u/dir/pname/bin` を **_path** に、`/u/dir/pname/lib` を **_fpath** に、`/u/dir/pname/include` を **_ipath** に、`/u/dir/pname/dl` を **_lpath** に追加します。
`package _fpath` <*path*>	`$(_ppath)/`*path* を **_fpath** に追加します。
`package _fpath "`*/u/dir/path*`"`	*/u/dir/path* を **_fpath** に追加します。
`package _ipath` <*path*>	`$(_ppath)/`*path* を **_ipath** に追加します。
`package _ipath "`*/u/dir/path*`"`	*/u/dir/path* を **_ipath** に追加します。
`package _lpath` <*path*>	`$(_ppath)/`*path* を **_lpath** に追加します。
`package _lpath "`*/u/dir/path*`"`	*/u/dir/path* を **_lpath** に追加します。
`package _path` <*path*>	`$(_ppath)/`*path* を **_path** に追加します。
`package _path "`*/u/dir/path*`"`	*/u/dir/path* を **_path** に追加します。
`_fpath` <*path*>	`CHHOME/toolkit/lib/`*path* を **_fpath** に追加します。
`_fpath "`*/u/dir/path*`"`	*/u/dir/path* を **_fpath** に追加します。
`_ipath` <*path*>	`CHHOME/toolkit/include/`*path* を **_ipath** に追加します。
`_ipath "`*/u/dir/path*`"`	*/u/dir/path* を **_ipath** に追加します。
`_lpath "`*/u/dir/path*`"`	*/u/dir/path* を **_lpath** に追加します。
`_path "`*/u/dir/path*`"`	*/u/dir/path* を **_path** に追加します。

```
int var1;
int var2;
..
#pragma remvar(var1)
float var1;
int main(){
   #pragma remvar(var2)
   float var2;
   ...
}
```

次の例では、キーワード **int** をシステムから削除し、後で変数識別子として使用します。

```
#pragma remkey(int)
float int;
int = 10;
```

プログラムが解析されるとき、ディレクティブ`#pragma exec expr`にある式`expr`が評価されます。expr は汎用関数を含むことができますが、関数ファイル内にある関数は含むことができません。たとえば、汎用関数`getenv()`によって取得されたホームディレクトリが/home/myname であると想定し、次のディレクティブがあるとします。

```
#pragma exec _fpath=stradd(_fpath, getenv("HOME"), "/chfunc;");
```

このディレクティブは、解析時および実行時の両方に、関数ファイル用のシステム変数 **_fpath** にディレクトリ`/home/myname/chfunc`を追加します。別の例として、次のように、解析時に汎用関数 **clock()** を呼び出すことによって、ヘッダーファイルとブロック内のコードの解析にかかる CPU 時間 (秒単位) を取得できます。

```
#include <time.h>
#pragma exec clock();
#include <headerfiles.h>
/* other code */
#pragma exec printf("CPU: %f\n", (double)clock()/CLOCKS_PER_SEC);
```

ディレクティブ`#pragma pack`は、構造体、共用体又はクラスのメンバのパッキングした配置、すなわちメンバをどのようなバイト境界に配置するかを指定します。次のディレクティブ

```
#pragma pack()
#pragma pack(pop)
```

はともに、パッキングした配置を自動的に設定します。次のディレクティブ

```
#pragma pack(push, n)
#pragma pack(n)
```

はともに、パッキングに使用する値をバイト単位で指定します。有効な値は 1、2、4、8 及び 16 です。メンバは次のバイト境界のいずれかに配置されます。すなわち、パラメータに指定された n の値の整数倍かまたはメンバのバイトサイズの整数倍のいずれか小さい方です。

表 5.3: C と Ch の両方で定義されているマクロ

マクロ名	説明
`__LINE__`	10 進の整数定数として表される現在のソースプログラム行の行番号。
`__FILE__`	文字列定数として表される現在のソースファイルの名前。
`__DATE__`	`"Mmm dd yyyy"`という形式の文字列定数として表されるカレンダー日付。`Mmm` は **asctime()** によって生成されるのと同一です。
`__TIME__`	**asctime()** によって返されるのと同様の、`"hh:mm:ss"`という形式の文字列定数として表される現在の時刻。
`__STDC__`	10 進定数の 1。
`__STDC_VERSION__`	10 進定数の 199901L.

表 5.4: Ch に定義されているマクロ

マクロ名	説明
`_CH_`	10 進定数の 1。
`_CHDLL_`	ダイナミックリンクライブラリがサポートされている場合は、10 進定数の 1。 それ以外の場合は、未定義です
`_GLOBALDEF_`	定義済みマクロがプログラムスコープ内にある場合は、10 進定数の 1。プログラム、ドットファイル、および関数ファイル内の定義済みマクロが相互に関連していない場合は、未定義にします。デフォルトでは、1 と定義されています。
`_M64_`	10 進定数の 1。64 ビットマシンでのみ定義されています。
`_SCH_`	Ch がセーフシェルとして呼び出される場合は、10 進定数の 1。それ以外の場合は、未定義です。

5.10 事前定義済みのマクロ

C と Ch の両方にあらかじめ定義されているマクロ名を表5.3に、 Ch だけの事前定義マクロ名を表5.4に、プラットフォームに依存しないマクロを表5.5に示します。プラットフォームに依存しないマクロである`_HPUX_`、`_LINUX_`、`_LINUXPPC_`、`_SOLARIS_`、`_WIN32_`、`_DARWIN_`、`_FREEBSD`、`_QNX_`、`_AIX_`は、主にスタートアップファイルとヘッダーファイルで使用するために定義されています。移植可能な Ch アプリケーションプログラムでこれらのマクロを使用することは避けてください。

表 5.5: Ch に定義されているプラットフォームに依存しないマクロ

マクロ名	説明
__ppc__	Darwin で Mac OS X の PowerPC が使用されている場合は、10 進定数の 1。それ以外の場合は、未定義です。
__i386__	Intel x86 32 ビットマシンに対しては、 10 進定数の 1。それ以外の場合は、未定義です。
__x86_x64__	Intel x86 64 ビットマシンに対しては、 10 進定数の 1。それ以外の場合は、未定義です。
__BIG_ENDIAN__	Darwin の Mac OS X、または Solaris の Sparc でビッグエンディアンマシンが使用されている場合は、10 進定数の 1。それ以外の場合は、未定義です。
__LITTLEENDIAN__	QNX のリトルエンディアンマシンが使用されている場合は、10 進定数の 1。それ以外の場合は、未定義です。
AIX	AIX が使用されている場合は、10 進定数の 1。それ以外の場合は、未定義です。
DARWIN	Darwin で Mac OS X が使用されている場合は、10 進定数の 1。それ以外の場合は、未定義です。
FREEBSD	FreeBSD OS が使用されている場合は、10 進定数の 1。それ以外の場合は、未定義です。
HPUX	HP-UX OS が使用されている場合は、10 進定数の 1。それ以外の場合は、未定義です。
LINUX	Linux OS が使用されている場合は、10 進定数の 1。それ以外の場合は、未定義です。
LINUXPPC	PowerPC 用 Linux OS が使用されている場合は、10 進定数の 1。それ以外の場合は、未定義です。
QNX	QNX OS が使用されている場合は、10 進定数の 1。それ以外の場合は、未定義です。
SOLARIS	Solaris OS が使用されている場合は、10 進定数の 1。それ以外の場合は、未定義です。
WIN32	Windows OS が使用されている場合は、10 進定数の 1。それ以外の場合は、未定義です。
_X86_	Windows と Mac OS X x86 で Intel x86 プロセッサが使用されている場合は、10 進定数の 1。それ以外の場合は、未定義です。

Solaris の一部のプログラムでは、以下のコマンドによってマクロ__STDC__を値 0 で再定義する必要があります。

```
#define __STDC__ 0
```

第6章　型と宣言

Ch は豊富なデータ型を持つゆるやかに型指定された言語です。自動的な型変換を禁止する Pascal などの言語と異なり、コンテキストにおいて合理的である場合は、Ch のデータ型を他のデータ型に自動的に変換することができます。オブジェクトに格納される値または関数によって返される値の意味は、その値にアクセスするために使用する式の型によって決まります。オブジェクトとして宣言される識別子は、最も単純な式です。型は識別子の宣言で指定されます。型は、オブジェクトを記述するオブジェクト型、関数を記述する関数型、オブジェクトを記述するがサイズを特定するために必要な情報を持たない不完全な型に分けられます。コンピュータのメモリに格納される値の形式は、使用しているマシンのアーキテクチャによって異なります。本章では、さまざまな型の識別子を宣言する方法と、Ch 内部での操作のために値がコンピュータシステムでどのように表現されるかについて説明します。本章の説明は、SUN Sparc ワークステーションに採用されている RISC プロセッサのアーキテクチャに基づきます。

6.1　データ型

6.1.1　整数型のデータ

整数はすべてのコンピュータ言語の基本データ型です。Ch では整数を次のいずれかのデータ型で表現できます。

```
char
signed char
unsigned char
short
signed short
unsigned short
int
signed int
unsigned int
long
long int
signed long
signed long int
unsigned long
unsigned long int
long long
```

```
long long int
signed long long
signed long long int
unsigned long long
unsigned long long int
```

Ch での char および int の数値操作は、C に定義されているルールに従います。

char のデータ表現

char データは、文字や句読点などの文字を格納するために使用します。char の配列を使用すると文字列を格納できます。文字は、実際には、ASCII コードなど特定の数値コードに従って整数として格納されます。このコードでは、特定の整数が特定の文字を表します。標準の ASCII コードは 0〜127 の範囲を持ち、表現するために 7 ビットしか必要としません。Ch では、char 型変数は CHAR_MIN から CHAR_MAX までの 範囲を持つ符号付き整数です。C の標準ヘッダー **limits.h** で定義されているマクロ CHAR_MIN および CHAR_MAX は、Ch ではシステム定数です。通常、char 型の定数または変数は 1 バイト単位メモリを占有します。ビット 8 は符号ビットです。符号付き 1 バイト表現の最大の正の整数は、127 または 2 進形式 01111111 です。負の数は 2 進の補数形式で格納されます。数値の 2 の補数を生成するには、2 進のすべてのビット (char では 8 ビット) を反転し、結果に 1 を足します。たとえば、10 進値の 2 は、2 進数では 8 ビットの char 整数を使用して 00000010 と表現されます。10 進値の −2 は、次のように、1 バイトの 2 の補数形式である 11111110 という 2 進値で表現されます。

$$
\begin{aligned}
(-2)_{10} &= \text{complement}(00000010)_2 + (1)_2 \\
&= (11111101)_2 + (1)_2 \\
&= (11111110)_2
\end{aligned}
$$

ここで、2 と 10 の添字は、関連付けられている数値の基数を示します。符号付き char の最小の整数値は、−128 または 2 進形式の 10000000 です。したがって、char の整数の範囲は、−128 から +127 までとなります。

符号なし char のデータ表現

Ch では、符号なし char 型変数は 0 から UCHAR_MAX までの符号なし int に相当します。C の標準ヘッダー **limits.h** で定義されているマクロ UCHAR_MAX は、Ch ではシステム定数です。通常、符号なし char 型変数は符号ビットを含まない 1 バイト単位メモリを占有し、パラメータ UCHAR_MAX は 255 または 2 進数形式で 11111111 となります。

short のデータ表現

short 型変数の範囲は SHRT_MIN から SHRT_MAX までです。C の標準ヘッダー **limits.h** で定義されているマクロ SHRT_MIN および SHRT_MAX は、 Ch ではシステム定数です。Ch では、short 型データは符号用の 1 ビットを含む 2 バイト (16 ビット) を使用して格納されます。負の数値は 2 バイ

トの 2 の補数形式で格納されます。そのため、パラメータ SHRT_MIN と SHRT_MAX は、それぞれ $-32768(2^{15})$ と $32767(2^{15}$ - 1) です。

符号なし short のデータ表現

符号なし short 型変数の範囲は、0 から USHRT_MAX までです。C の標準ヘッダー **limits.h** で定義されているマクロ USHRT_MAX は、Ch ではシステム定数です。Ch では、符号なし short 型変数は符号ビットを含まない 2 バイト単位メモリを占有し、パラメータ USHRT_MAX は $65535(2^{16}$-1) となります。

int のデータ表現

int 型データは、Ch では符号付き整数です。int 型の数値は整数であり、負数、正数、または 0 とすることができます。int 型の範囲は INT_MIN から INT_MAX までです。 C の標準ヘッダー **limits.h** で定義されているマクロ INT_MIN および INT_MAX は、Ch ではあらかじめ計算されたシステム定数です。int 型データが 2 バイトのみを占有する一部の C の実装とは異なり、Ch の int 型データは、符号用の 1 ビットを含む 4 バイト (32 ビット) を使用して格納されます。負数は 4 バイトの 2 の補数形式で格納されます。そのため、INT_MIN と INT_MAX の値は、それぞれ -2147483648 (2^{31}) と 2147483647 になります。たとえば、Ch では次のステートメントが有効です。

```
> char c[2][3], *cptr;
> int i, *iptr;
> c[0][1] = 'a'; // c[0][1] becomes 'a'
a
> i = c[0] [ 1];   // i becomes 97, ASCII number for 'a'
97
> c[1][2] = i+1  // c[1][2] becomes `b', ASCII number for `b' is 98
b
> i += c[1][2];  // i becomes 195 = 97 + 98
195
> iptr = &i;     // iptr points to address of i
4005ec50
> *iptr /= 2;    // i becomes 97 = 195/2
97
>
```

Ch では、`c[i][j]` によって配列の宣言およびアクセスができます。`i = c[0] [ 1]` などのステートメントに含まれる空白とタブ文字は、Ch プログラム内では無視されます。ただし、`"ab cd"` のように文字列内に含まれる文字は例外です。

符号なし int のデータ表現

符号なし int 型変数の範囲は、0 から UINT_MAX までです。C の標準ヘッダー **limits.h** で定義されているマクロ UINT_MAX は、Ch ではシステム定数です。Ch では、符号なし int 型変数は符号ビットを含まない 4 バイト単位メモリを占有し、パラメータ UINT_MAX は 4294967295 (2^{32} - 1) となります。

long のデータ表現

Ch では、long 型および long int 型のデータの表現は、C の場合と同じです。たとえば 32 ビットマシンの場合 long は例によって、int と同じになります。64 ビットマシンでは、Linux 64 ビットの場合、long は long long と同じであり、Windows 64 ビットの場合、int と同じになります。

long long のデータ表現

Ch では、long long 型および符号なし long long 型の整数データは 64 ビットを含みます。これらの型のデータ表現は、int および符号なし int のデータ型と同様です。次に例を示します。

```
> long long l
> l = 10LL
> printf("l = %lld", l);
l = 10
> sizeof(l)
8
> scanf("%lld", &l);
11
> printf("l = %lld", l);
l = 11
>
> unsigned long long ul
> ul = 10ULL
> printf("ul = %llu", ul);
ul = 10
>
```

6.1.2 浮動小数点型

整数データ型は、ソフトウェア開発プロジェクト (特にシステムのプログラミング) で役に立ちます。ただし、科学計算では、浮動小数点数が幅広く使用されます。浮動小数点数は、整数と整数の間にある数値を含む実数に対応します。Ch では、このような数値を float 型または double 型で定義します。これらは、Fortran の実数型および倍精度浮動小数点型にそれぞれ相当します。浮動小数点数は科学的な表記での数値表現と似ています。整数演算と比較して、浮動小数点演算は複雑です。

浮動小数点演算の最も一般的な実装は、IEEE 754 標準に基づいています。この標準では float または double は次の形式で表されます。

$$(-1)^{sign}2^{exponent-bias}1.f \tag{6.1}$$

ここで $1.f$ は仮数部です。1 は暗黙ビットであり、f は正規化された数値の小数部分のビットを表します。この正規化された浮動小数点数には "隠された" ビット '1' が含まれます。そのため、この表現は他の場合よりも 1 ビット分精度が高くなっています。

float のデータ表現

float データ型は記憶領域として 32 ビットを使用します。float 型データの処理結果を式で表すと次のとおりです。

$$(-1)^{sign}2^{exponent-127}1.f \tag{6.2}$$

ビット 31 は符号ビットであり、負数の場合、このビットは 1 です。ビット 23〜30 は**指数**部ビットです。指数部は 127 の補正値であり、1 刻みの数値を使用できます。表6.1に示すように、すべての指数部ビットが 0 である表現と、すべての指数部ビットが 1 である表現は、メタ数値 Inf、–Inf、NaN 用に予約されています。ビット 0〜22 は仮数の小数部分を定義します。正規化された仮数の先行する整数は必ず 1 となるため、格納する必要がありません。2 進小数では、最上位ビットは 0.5 を表し、次のビットは 0.25、0.125 などを表します。float 型数値の一部について、16 進表現を表6.1に示します。

たとえば、(6.2) の式では、float 型数値 1.0 および –2.0 は、それぞれ $(-1)^0 2^{127-127}1.0 = 1.0$ および $(-1)^1 2^{128-127}1.0 = 2.0$ で求めることができます。正規化された仮数の小数部は 2 進小数として格納されることに注意してください。float 型数値 3.0 は、$(-1)^0 2^{128-127}(1.1)_2 = 2*(1.1)_2 = 2*(1.5)_{10} = (3.0)_{10}$ で計算できます。ここで、添字は浮動小数点数の基数を示します。IEEE 754 標準では、浮動小数点数としての $+0.0$ と –0.0 を区別することに注意してください。

float データ型で表現可能な最大の有限浮動小数点数として C の標準ヘッダー **float.h** に定義されているマクロ FLT_MAX は、Ch ではあらかじめ計算されたシステム定数です。数値が FLT_MAX より大きい場合は、**オーバーフロー**と呼ばれます。FLT_MAX より大きい数値は、すべての指数部ビットが 1 に設定されます。これは、算術記号の無限大 ∞ に相当するメタ数値 Inf で表される必要があります。これは有限数のゼロ除算などの多くの演算の結果です。ただし、IEEE マシンでは厳密でない例外が発生する可能性があります。–FLT_MAX 未満の数値は、負の無限大の算術記号 $-\infty$ に相当するメタ数値 –Inf で表される必要があります。

パラメータ FLT_MIN の値は、正規化された最小の正の浮動小数点数値として、C の標準ライブラリヘッダー **float.h** に定義されています。数値が FLT_MIN 未満である場合は、**アンダーフロー**と呼ばれます。IEEE 754 標準には、**段階的アンダーフロー**が用意されています。

数値が小さすぎるために正規化した表現にできない場合は、先行するゼロが仮数内に配置されて、非正規化表現が生成されます。**非正規化数**は、正規化されない非ゼロの数値であり、その指数部は格納する型での最小の指数です。表現可能な最大の正の非正規化 float は、表6.1に示すように、Ch では FLT_MINIMUM として定義されています。FLT_MINIMUM には最後の桁に 1 つの単位だけがあるため、*ulp* と一般的に呼ばれます。ほとんどすべての浮動小数点の実装では、FLT_MINIMUM (IEEE マシンの場合) および FLT_MIN (IEEE 以外のマシン) より小さい値を値 0 に置き換えます。

表 6.1: 一部の実数値の 16 進表現

value	float	double
0.0	00000000	0000000000000000
−0.0	80000000	8000000000000000
1.0	3F800000	3FF0000000000000
−1.0	BF800000	BFF0000000000000
2.0	40000000	4000000000000000
−2.0	C0000000	C000000000000000
3.0	40400000	4080000000000000
−3.0	C0400000	C080000000000000
Inf	7F800000	7FF0000000000000
−Inf	FF800000	FFF0000000000000
NaN	7FFFFFFF	7FFFFFFFFFFFFFFF
FLT_MAX	7F7FFFFF	
DBL_MAX		7FEFFFFFFFFFFFFF
FLT_MIN	007FFFFF	
DBL_MIN		000FFFFFFFFFFFFF
FLT_MINIMUM	00000001	
DBL_MINIMUM		0000000000000001

ただし、 Ch で定義されている算術演算および数学関数では、FLT_MIN より小さい FLT_MINIMUM とゼロとの間には質的な違いがあります。本書では、0.0 という値は、小さい数値ではなくゼロを意味します。Ch の式では 0. と 0.00 と .0 はいずれも 0.0 と同じです。同様に、 Ch の浮動小数点の定数式 −0.0、−0.、−0.00、および −.0 は等価です。

数学的には、0.0/0.0 というゼロによるゼロ除算、および ∞/∞ という無限大による無限大除算は不定です。これらの演算の結果は、非数を表す NaN という記号で表現されます。IEEE 754 標準では、**シグナルなし**の NaN と**シグナルあり**の NaN を区別することに注意してください。シグナルありの NaN は、シグナルを生成するか、例外を発生させます。Ch では、すべての NaN はシグナルなしの NaN として処理されます。さらに、IEEE 754 標準では NaN の符号を解釈しません。

Ch では、算術および関数の実行結果として −NaN が生成されることはありません。ただし、float 型変数のメモリ上の場所に対するビットパターン操作によって作成することができます。Ch では式 −NaN は NaN として解釈されます。メタ数値は単に通常の浮動小数点数として処理されます。表6.1では、float 型のメタ数値の内部 16 進表現も示しています。

double のデータ表現

Ch では、より広い範囲を表現可能な浮動小数点数として、double 型データを使用します。double データ型は記憶領域として 64 ビットを使用します。double 型データの処理結果を式で表すと次のとおりです。

$$(-1)^{sign}2^{exponent-1023}1.f \tag{6.3}$$

ビット 63 は符号ビットであり、負の数の場合、このビットは 1 です。ビット 52〜62 の 11 ビットの指数部は、1023 でバイアスされます。11 ビットがすべてゼロの値およびすべて 1 の値は、メタ数値として予約されています。

ビット 0〜51 は正規化された仮数の小数部です。float と同様に、正規化された仮数の 1 という整数値は隠されます。表6.1では、代表的な double 型数値の 16 進表現も示しています。double の指数部の幅とバイアス値は、float のものとは異なることに注意してください。

そのため、単に小数部にゼロを埋め込むだけで float を double に変換することはできません。一方、double 型データを float にキャストする場合、単にビット 0〜31 の値を無視するだけで結果を得ることはできません。

内部表現は異なりますが、float Inf と double Inf に外部的な違いはないことに注意してください。メタ数値 –Inf および NaN についても同じです。float と同様、パラメータ DBL_MAX、DBL_MIN、および DBL_MINIMUM は Ch ではシステム定数です。特殊な double 型の有限浮動小数点数の内部メモリ表現も表6.1に示しています。なお、浮動小数点数表現の精度は有限であるため、π などの無理数の正確な値は、float であるか double であるかにかかわらず、コンピュータシステムでは表現できません。

6.1.3 集合体の浮動小数点型

実数の拡張である複素数には、科学および工学の分野において幅広い用途があります。複素数型の変数は、`complex` および `double complex` という 2 つの型指定子によって宣言できます。宣言の後、 Ch で複素数コンストラクタ **complex**(x, y) によって複素数を作成できます。x は実部、y は虚部です。次に例を示します。

```
> complex z1;           // a double complex variable
> double complex z;     // a double complex variable
> float complex z2;     // a float complex variable
> z1 = complex(1, 2);   // z1 becomes 1 + i2
complex(1.00,2.00)
>
```

単純な複素数型の変数だけでなく、複素数へのポインタ、複素数の配列、および複素数へのポインタの配列なども宣言することができます。このような複素数型の変数の宣言は、他のデータ型の宣言と同じです。次に例を示します。

```
> complex *zptr1;
> complex z2[2], z3[2][3]; // declared as pointer to complex variable
> complex *zptr[2][4];      // array of pointer to complex
> zptr1 = &z1;              // zptr1 point to the address of z1
4005e748
> *zptr1 = complex(2, 3);   // z1 becomes 2 + i3
complex(2.00,3.00)
```

```
> z1
complex(2.00,3.00)
>
```

入出力操作、データ変換ルール、関数などの複素数の詳細については、第13章を参照してください。

6.1.4 ポインタのデータ型

ポインタは、別の変数のアドレスまたは動的に割り当てられるメモリのアドレスを格納する変数として定義されます。Ch では配列、構造体、関数、クラス、および単純なデータ型へのポインタを明示的に使用します。変数名の前に演算子'*' を付けると、他のデータ型の変数と同様に、ポインタ型の変数を宣言できます。単項演算子'&' は、"変数のアドレス"を指定します。たとえば、次のコードがあるとします。

```
int i, *p1, **p2;
p1 = &i;
p2 = &p1;
```

このコードでは、`p1` および `p2` という 2 つのポインタを宣言します。`p1` は整数 `i` のアドレスを格納し、`p2` は `p1` のアドレスを格納します。ポインタの詳細については、第9章を参照してください。単純なデータ型へのポインタの他に、Ch では配列および関数へのポインタも使用できます。配列および関数へのポインタの詳細については、それぞれ第 14章およびセクション10.8を参照してください。

6.1.5 配列型

配列の次元数は、配列の**ランク**と呼ばれます。1 つの次元にある要素の数は、その次元での配列の**エクステント**と呼ばれます。配列の**形状**はベクトルであり、ベクトルの各要素は対応する次元のエクステントです。

Ch では、計算配列はファーストクラスオブジェクトです。Ch Professional Edition および Student Edition では、計算配列の型修飾子 `array` がヘッダーファイル **array.h** にマクロとして定義されています。配列の宣言を次に示します。

```
#include <array.h>
int a1[3][4];        // array of integer
int *a2[3][4];       // array of pointer
array int a3[3][4]; // computational array
```

ここで、`a1` は整数の配列として宣言されます。`a2` は整数へのポインタの配列であり、`a3` は計算配列です。この宣言では、型修飾子 **array** によって a3 を計算配列にしています。計算配列は、線形代数および行列計算のファーストクラスオブジェクトとして処理できます。形状無指定配列、形状引継ぎ配列、形状引継ぎ配列へのポインタ、および参照の配列を含む可変長の配列がサポートされます。このようなさまざまな配列定義の概念を明らかにする例を次に示します。

```
void funct(int a[:][:], (*b)[:], c[], d[&], n, m){
/* a: assumed-shape array */
/* b: pointer to array of assumed-shape */
/* c: incomplete array completed by function call */
/* d: array of reference */
/* n, m: integers */
  int e[4][5];      // fixed-length array
  int f[n][m];      // deferred-shape array
  int (*g)[:];      // pointer to array of assumed-shape
  extern int h[];   // incomplete array completed by external linkage
  int i[] = {1,2}; // incomplete array completed by initialization
  f[1][2] = a[2][3];
}
int A[3][4], B[5][6], C[3], D[4];
funct(A, B, C, D, 10, 20);
funct(B, A, C, D, 85, 85);
```

引数 a は、さまざまなエクステントを含む配列を渡すことのできる形状引継ぎ配列として宣言されます。引数 b は形状引継ぎ配列へのポインタとして宣言されます。c は関数呼び出しによって完了する不完全配列として宣言されます。d はさまざまなデータ型の配列を処理できる参照の配列です。

次に示す for ループでは、異なるサイズを持つ配列 a が宣言し直されると、メモリが関数 **realloc()** によって内部的に再割り当てされます。

```
int i;
for (i = 0; i<10; i++) {
  int a[i];
  ...
}
```

配列のインデックスを表す添字の範囲は調整できます。次に例を示します。

```
int a[1:10], b[-5:5], c[0:10][1:10], d[10][1:10];
int e[n:m], f[n1:m1][1:m2];
extern int a[1:], b[-5:], c[0:][1:10];
int funct(int a[1:], int b[1:10], int c[1:][3], int d[1:10][0:20]);
a[10] = a[1]+2; /* OK */
a[0] = 90;      /* Error: index out of range */
```

ここで、添字 a の範囲は 1〜10、b の範囲は-5〜5 です。c の最初の次元の範囲は 0〜10 であり、2 番目の次元の範囲は 1〜10 です。d の最初の次元の範囲は 0〜9 であり、2 番目の次元の範囲は 1〜10 です。

参照の配列には、さまざまな形状とデータ型の配列を渡すことができます。

次に例を示します。

```
float a[3][4];
double b[5][6];
void func(double a[&][&]);
func(a);
func(b);
```

ここでは、関数 func の引数 a を参照の配列として宣言しており、さまざまなエクステントとデータ型を持つ配列を渡すことができます。

構造体または共用体のメンバを形状引継ぎ配列へのポインタとして使用できます。次に例を示します。

```
int a[4][5], b[7][8];
struct tag_t {
  int n;
  int (*A)[:];
} s;
s.A = a; /* s.A[i][j] == a[i][j] */
...
s.A = b; /* s.A[i][j] == b[i][j] */
```

ここでは、構造体 s のメンバ A を、さまざまなエクステントを持つ配列を割り当てることのできる形状引継ぎ配列へのポインタとして宣言しています。

ポインタと配列の関係の詳細については第 14章を、計算配列の詳細についてはセクション6.2.1と第16章を参照してください。

6.1.6 構造体型

Ch の構造体型は C++と似ています。構造体型は、さまざまな型を持つことができるメンバの集合です。次に例を示します。

```
struct tag_t {
  data_type1 field1;
  data_type2 field2;
} name1;
tag_t name2, *name3;
struct tag_t name4;
name3 = &name2;
```

ここで、tag_t というタグ名の構造体に、field1 および field2 という 2 つのメンバがあります。構造体 tag_t の 3 つのオブジェクト (name1、name2、name4) を異なる 3 とおりの方法で宣言しています。name1 は構造体の定義後に直接宣言します。name2 はタグ名だけで宣言し、name4 はオプションのキーワード構造体を使用して宣言します。変数 name3 を構造体へのポインタとして宣言し、name2 のアドレスを割り当てます。

C の構造体には 2 つの名前空間があります。1 つは構造体タグであり、1 つはメンバ変数です。しかし、C++の構造体には 1 つと半分の名前空間があります。1 つは構造体タグの名前空間であり、半分はメンバ変数の名前空間です。Ch の構造体は C++の構造体と同様に処理されます。Ch ではタグは型指定された名前として暗黙的に処理されます。Ch の構造体タグと構造体変数は同じ名前空間を共有します。タグ名を変数として明示的に使用すると、この暗黙的な処理が無効になります。次に例を示します。

```
struct tag1_t{
    struct tag2_t {
    ....
    };
  ...
};
tag1_t s;           /* OK */
int tag1_t;         /* OK, tag name is used */
struct tag1_t s2;   /* OK */
tag1_t s3;          /* Not valid in Ch and C++ */
```

C++と同様に、Ch の構造体のメンバは関数であってかまいません。構造体の詳細については、第 18章を参照してください。

6.1.7 クラス型

Ch または C++のクラスは、構造体が自然に進化したものです。C++と同様に、Ch のクラスと構造体の両方が関数のメンバを持つことができます。既定では、クラスのメンバは private ですが、構造体のメンバは public です。

クラスの定義例を次に示します。

```
class Student {
    int id;
    char *name;
  public:
   void setName(char *n);
}

void Student::setName(char *n) {
  ...
}
```

クラス `Student` には 3 つのメンバがあります。2 つの `private` メンバとして `id` と `name`、および public メンバ関数である `setName()` です。id には生徒の ID 番号を格納し、`name` は生徒の名前であるとします。また、関数 `setName()` を使用して生徒の名前を設定するとします。クラスを定義した後、次のようにプログラム内でクラスを使用できます。

```
int main() {
    class Student s1;
    s1.setName("Bob");
    ...
}
```

ここで、s1 はクラス Student の**オブジェクト**または**インスタンス**と呼ばれます。クラス型の詳細については、第19章を参照してください。

6.1.8 ビットフィールド

C と同様に、Ch では単語内での直接的な定義とアクセスの機能を持つビットフィールドを提供します。次のコード例があります。

```
struct tag{
  data_type1 a:4;
  data_type2 b:4;
} name1 {1,1};
struct tag name2 = {1,1};
name2.a = 2;
```

タグ a と b という 2 つのメンバは、メモリを 8 ビットのみ (それぞれが 4 ビットずつ) 使用します。ビットフィールドの詳細については、第18章を参照してください。

6.1.9 共用体型

共用体型は、重なっている空でないメンバオブジェクト集合を記述します。共用体は同時にメンバの 1 つのみを保持できます。概念的には、メンバは同一のメモリ内で重なります。共用体の各メンバは共用体の先頭に配置します。たとえば、次のコードは共用体型を定義する方法を示しています。

```
union tag{
  data_type1 fields1;
  data_type2 fields2;
} name1;
tag name2, *name3;
union tag name4;
name3 = &name2;
```

メンバ fields1 および fields2 は同一のメモリを共有します。一度に 1 つのメンバのみを使用できます。C++と同様に、既定では、タグは型定義された名前空間に配置されます。共用体の詳細については、第18章を参照してください。

6.1.10 列挙型

列挙型は列挙定数で表現される一連の整数値です。たとえば、次のコードがあるとします。

```
enum tag_t{bad, good=1, ugly} x;
enum tag_t y;
x = good;
y = x;
```

このコードは、タグ名 tag_t によって示される新しい列挙型を定義します。x や y などの tag_t の変数により、bad、good、および ugly という 3 つの列挙定数を割り当てることができます。列挙型の詳細については、第18章を参照してください。

6.1.11 Void 型

void 型は主に、未定の型へのポインタとして、または引数リストや戻り値がない関数に対して使用されます。

ポインタで void の型を参照できます。他の型のポインタを void へのポインタに割り当てたり、void へのポインタから割り当てたりできます。また、void へのポインタと比較することができます。さらに、情報を失うことなく、任意のオブジェクトへのポインタを void 型に変換できます。ただし、元のポインタの参照先オブジェクトに正しくアクセスするためには、変換後のポインタを元のポインタ型に変換し直す必要があります。

関数の定義時にキーワード void を関数名の前に指定すると、関数が戻り値を持たないことを示します。たとえば、次に定義されている関数 funct1() には戻り値がありません。

```
void funct1(int i) {int i; ...; };       /* no return value */
```

関数の引数リストにキーワード void を指定すると、関数が引数を持たないことを示します。たとえば、次に定義されている関数 funct2() には引数がありません。

```
int funct2(void){int i; ...; return i;} /* no argument */
```

6.1.12 参照型

C++の場合と同様に、Ch では記号 ‘&’ で宣言した参照はオブジェクトの代替名です。構文は同じです。たとえば、次の宣言があるとします。

```
> int i
> int &j = i
> i = 5
5
> j
5
>
```

この宣言で変数 j は i の参照であることを示します。これらの変数はシステム内の同じメモリ空間を共有しており、したがって同一のものとして使用できます。値 i に変更を加えると、値 j にも反映されます。Ch では、char、short、int、float、double などの単純なデータ型の参照や、符号付き、符号なし、long、および複素数などの修飾されたデータ型に加えて、ポインタ型の参照も宣言できます。

Ch では、関数に引数を渡す既定の方法は値呼び出しですが、'&' 記号を使用することで参照呼び出しが可能です。たとえば、次に示す関数のプロトタイプ swap では、

```
void swap(int &n, int &m); /* the same as in C++ */
```

引数 n と m を int に対する参照として宣言しています。つまり、呼び出された関数 swap() 内で n および m が変更されると、呼び出し側の関数内の元の値に反映されます。参照型の詳細については、第 11章を参照してください。

6.1.13 文字列型

文字列は型指定子 string_t を持つ Ch のファーストクラスオブジェクトです。関数内では、char へのポインタ型の引数を文字列型の引数によって置き換えることができます。安全なネットワークコンピューティングのためには、char へのポインタではなく char の文字列または配列を使用する必要があります。文字列型変数のメモリ割り当てと割り当て解除は、Ch によって自動的に処理されます。

```
string_t s1, s2, s, a[3];
s1 = "Hello, ";
s2 = "world!";
s = s2;
int i = strlen(s1);
strcat(s1,s2);   /* s1 becomes "Hello, world!" */
strcpy(a[0],s1); /* a[0] becomes "Hello, world!" */
```

次のプログラムに示すように、 Ch では参照型の文字列がサポートされています。

```
string_t stringcat(string_t &s1, s2)
{
  string_t s;
  s = strcat(s1, s2);
  /* s = stradd(s1, s2); */
  s1 = s;
  return s1;
}
string_t s1 = "string1", s2 = "string2";
stringcat(s1, s2);
printf("%s\n", s1); // print out string1 and string2
```

関数 stringcat() で文字列 s1 の末尾に文字列 s2 を追加します。これは C の標準関数 **strcat()** に相当します。

また、次のコードに示すように文字列へのポインタを宣言することもできます。

```
string_t stringcat2(string_t *s1, s2) {
  *s1 = strcat(*s1, s2);
  return *s1;
}
string_t s1 = "string1", s2 = "string2";
stringcat2(&s1, s2);
printf("%s\n", s1); // print out string1 and string2
```

関数 `stringcat2()` の働きは関数 `stringcat()` と同様です。

文字列型は以下のように、空白をともなった文字列を入力関数 **scanf()** をとおして取得するために使用されます。

```
> string_t str;
> scanf("%s", &str);
abcd 1234
> str
abcd 1234
```

関係演算子`==`、`!=`、<、および > は、一方のオペランドが内蔵文字列型 **string_t** であり、他方のオペランドが **string_t**、文字列テラル "something"、char へのポインタ、unsigned char へのポインタである、2 つの文字列の比較に使用されます。2 つの文字列が同じである場合、演算 == の結果は 1、演算 `!=` の結果は 0 となります。そうでない場合、演算 == の結果は 0、演算 `!=` の結果は 1 となります。

> と < の結果は関数 **strcmp()** からの 1 と −1 の結果と同様です。演算子> と < は、お使いのマシンの文字セットの順にしたがって、2 つの文字列をバイト単位で比較します。演算 `s1>s2` の結果は、比較される文字列において異なっている最初のペアのバイト値間で、文字列 s1 が文字列 s2 より大きい場合は 1 となり、そうでない場合は 0 となります。演算 `s1<s2` の結果は、比較される文字列において異なっている最初のペアのバイト値間で、文字列 s1 が文字列 s2 より小さい場合は 1 となり、そうでない場合は 0 となります。

文字列のオペランドを使用したシンボリック計算の演算を表6.2に示します。

表 6.2: 文字列を使用したシンボリック計算の演算

定義	Ch 構文
加算	s1 + s2
減算	s1 − s2
乗算	s1 ∗ s2
除算	s1 / s2

シンボリック計算の例を次に示します。

```
int i = 2;
float x = 10, y;
string_t a="sin(x)", b="x", s;
```

```
s = i*a/b+1;
s = s+s
printf("s = %s\n", s); /* output is 2*(2*sin(x)/x+1) */
y = streval(s);        /* y = 1.782 = 2*(2*sin(10)/10+1) */
printf("y = %f\n", y); /* output is y = 1.782 */
```

Ch のシンボリック計算の機能は、現時点の実装ではまだ極めて暫定的なものです。文字列型の詳細については、第17章を参照してください。

6.1.14 関数型

Ch の標準関数は C 標準に準拠しています。C の考え方に従って、入れ子にされた関数の Ch での関数定義は次の形式になります。

```
return_type function_name(argument declaration)
{
  statements
  function_definitions
}
```

または

```
return_type function_name(argument declaration)
{
  function_definitions
  statements
}
```

ここで、ステートメントは任意の有効な Ch ステートメントとすることができ、ローカル関数を他のローカル関数内に定義できます。Ch で入れ子にする関数の数に制限はありません。次に例を示します。

```
int func1() {
   int func2() {
     int func3() { ...}
   }
   /* ...  */
   func2();
}
```

ローカル関数の定義は関数内の任意の場所に配置できます。ローカル関数を定義の前に呼び出す場合は、プログラム6.1に示すように、ローカル関数プロトタイプを使用する必要があります。

```
void funct1()
{
  __declspec(local) float funct2(); // local function prototype
  funct2();
  float funct2()        // definition of the local function,
  {
    return 9;
  }
}
```

プログラム 6.1: 宣言__declspec(local) で funct2() をローカル関数として修飾

プログラム6.1では、関数 funct2() を定義の前に使用しているため、関数プロトタイプが必要です。これはローカル関数であるため、型修飾子__declspec(local) を使用して、最上位レベルの標準 C 関数とローカル関数を区別します。

引数リストのパラメータを関数内で使用しない場合、関数定義でそれらのパラメータを無視できます。次に例を示します。

```
int func(int i, int /* not_used */, int /* no_used */) {
   return i*i;
}
func(10, 20, 30);
```

6.2 型修飾子

Ch の型修飾子を表6.3に示します。型修飾子 array は計算配列に、restrict は制限付き関数に使用します。

表 6.3: 型修飾子

修飾子	関数
array	計算配列
const	(現在は無視されます。今後修正される予定です)
inline	(無視されます)
operator	(発生する可能性のある演算子のオーバーロードに備えて予約されています)
restrict	制限付き関数。引数リスト内に指定した場合は無視されます。
virtual	(現在は無視されます。C++での仮想関数のために予約されています)
volatile	(無視されます)

6.2.1 計算配列

型修飾子 array によって修飾した配列は**計算配列**と呼ばれ、Ch Professional Edition および Student Edition で使用できます。この型修飾子は、ヘッダーファイル array.h にマクロとして定義されています。計算配列はファーストクラスオブジェクトとして処理されます。次に例を示します。

```
array float a[10][10], b[10][10];
a += b+inverse(a)*transpose(a);
```

これは $a = a + b + a^{-1} * a^T$ の場合です。計算配列は関数の引数として使用できます。通常の完全な C 配列を計算配列の引数に渡すことができます。またその逆も可能です。計算配列の詳細については、第16章を参照してください。

6.2.2 制限付き関数

Ch では、関数定義内または戻り値の型の宣言の前に型修飾子 restrict を指定すると、関数は制限付き関数として処理されます。セキュリティのため、セーフ Ch プログラムで制限付き関数を呼び出すことはできません。たとえば、次のように制限付き関数 restricted_function() を宣言し、セーフ Ch で実行できないようにすることができます。

```
restrict int restricted_function(int i);
```

C の標準ライブラリにある **fopen()** などの一部の関数は、Ch では制限付き関数として定義されています。関数の引数リストに型修飾子 restrict を指定した場合は、無視されます。セーフ Ch の詳細については、第 21章を参照してください。

6.3 定数

このセクションでは、前のセクションで説明したデータ型の外部表現について説明します。宣言される変数およびシステム定義パラメータの他に、Ch のさまざまなデータ型では、プログラマが自由に対応する定数を使用できます。Ch の定数には 4 種類あります。文字、文字列、整数、浮動小数点数です。

6.3.1 文字定数

文字定数は整数として格納され、'x' のように 1 文字を一重引用符で囲んで記述できます。文字定数は char 型の変数に割り当てることができます。次に例を示します。

```
> char c = 'x'
> c
x
>
```

複数の文字またはエスケープ文字を含む文字定数はマルチバイト文字と呼ばれます。また、 Ch では文字 L を前に付けたワイド文字定数を使用できます。ソースプログラム内のアポストロフィ、バックスラッシュ、および読み取りにくい改行などの一部の文字を文字定数内に含めるには、後で説明するエスケープ文字を使用する必要があります。文字の詳細については、第17章を参照してください。

ワイド文字およびマルチバイト文字

Ch では、ロケール固有の文字を含む拡張文字セットを処理することができます。ロケール固有の文字は、サイズが 1 バイトである単一の char 型オブジェクトで表すには、常に長すぎます。このような文字を格納するために、Ch はワイド文字とマルチバイト文字の両方をサポートしています。”ワイド文字”は、ヘッダーファイル **stddef.h** に定義されている 整数型 **wchar_t** のオブジェクトに拡張文字コードが収まるようにする内部表現方法です。拡張文字の文字列は、wchar_t[] 型のオブジェクトまたは wchar_t *型のポインタとして表すことができます。たとえば、次のコードは Ch でワイド文字 wc を宣言します。

```
wchar_t wc = L'a';
```

文字 a の前に指定されている L は、文字 a がワイド文字であることを示します。一方、複数の文字列またはエスケープを含む “マルチバイト文字”は、Ch がサポートする外部表現方法です。マルチバイト文字は、ワイド文字に相当する一連の通常の文字列です。現在のロケールでマルチバイト文字を表すときに使用される最大バイト数は、ヘッダーファイル **stddef.h** に定義されているマクロ **MB_CUR_MAX** によって示されます。ワイド文字列はマルチバイト文字列によって外部的に表現できます。マルチバイト文字はコメント、文字列、および文字定数内で使用できます。

マルチバイト文字セットでは状況依存エンコーディングを指定することができます。このエンコーディングでは一連の各マルチバイト文字の先頭は初期シフト状態で始まり、シーケンス内で特殊なマルチバイト文字が出現したときに、その他のローカル固有のシフト状態に移行します。初期シフト状態にある間は、すべての単一バイト文字は通常の解釈を保持し、シフト状態を変更しません。シーケンス内の後続バイトの解釈は、現在のシフト状態の関数です。

また、Ch では C で定義されているマルチバイト文字とワイド文字間の変換を実装する機能もサポートしています。たとえば、ファイル **stdlib.h** で宣言されている関数 **mbtowc()** は、マルチバイト文字をワイド文字に変換し、関数 **wctomb()** ではその反対に変換します。

エスケープ文字

一部の特殊文字および出力デバイスの特定の動作をソースプログラムに直接入力することはできません。Ch では、エスケープ文字をサポートしています。エスケープ文字は、このような文字や動作を表すための先頭がバックスラッシュ文字 ‘\’ であるエスケープコードです。エスケープコードは、表 6.4に示す文字である文字エスケープコードか、最大 3 桁の 8 進数または任意の桁数の 16 進数である数値エスケープコードです。

表 6.4: 文字エスケープコード

エスケープコード	変換
\a	(アラート) 音声によるアラートまたは視覚的なアラートを生成します。アクティブな位置は変更されません。
\b	(バックスペース) アクティブな位置を現在の行の前の位置に移動します。アクティブな位置が行の初期位置である場合、動作は未指定です。
\f	(改ページ) アクティブな位置を次の論理ページの先頭の初期位置に移動します。
\n	(改行) アクティブな位置を次の行の初期位置に移動します。
\r	(復帰) アクティブな位置を現在の行の初期位置に移動します。
\t	(水平タブ) アクティブな位置を現在の行の次の水平タブ位置に移動します。アクティブな位置が、最後に定義されている水平タブ位置またはそれを過ぎた位置にある場合、動作は未指定です。
\v	(垂直タブ) アクティブな位置を次の垂直タブ位置の初期位置に移動します。アクティブな位置が、最後に定義されている垂直タブ位置またはそれを過ぎた位置にある場合、動作は未指定です。
\\	(バックスラッシュ) バックスラッシュ文字\を生成し、アクティブな位置を次の位置に移動します。
\'	(単一引用符) 単一引用符' を生成し、アクティブな位置を次の位置に移動します。
\"	(二重引用符) 二重引用符"を生成し、アクティブな位置を次の位置に移動します。
\?	(疑問符) 疑問符?を生成し、アクティブな位置を次の位置に移動します。

通常、文字エスケープコード\a は、スピーカーからアラートとしてビープ音を発します。アクティブな位置は、関数 **fputc()** または **fputwc()** による次の文字出力が表示されるディスプレイデバイス上の位置です。出力する文字を (**isprint()** 関数または **iswprint()** 関数での定義に従って) ディスプレイデバイスに書き出す目的は、その文字のグラフィック表現をアクティブな位置に表示して、現在の行の次の位置にアクティブな位置を進めることです。

コード\b は、アクティブな位置を現在の行の前の位置に移動します。コード\f は改ページを表し、アクティブな位置を次の論理ページの先頭の初期位置に移動します。コード\n は、最も一般的に使用されるエスケープコードであり、アクティブな位置を次の行の初期位置に移動します。一方、\r はアクティブな位置を現在の行の初期位置に移動します。コード\t および\v は、アクティブな位置を次の水平タブの位置および次の垂直タブの位置にそれぞれ移動します。

コード\\は、エスケープコードの先行文字ではないバックスラッシュを表します。文字定数内に指定した一重引用符は、文字定数の末尾のアポストロフィと誤る可能性があります。この場合、コード\' を使用すると、文字定数内で一重引用符を表すことができます。同様に、コード\"は、セクション6.3.2で説明するように、文字列定数内の二重引用符を表すことができます。コード\?を使用すると、セクション2.1.1で説明した三連文字の一部と間違えられる可能性がある場合に疑問符を生成でき

ます。次のコードは、文字エスケープの使用方法を示しています。

```
> printf("abcdefd");
abcdefd
> printf("abcd\befd");  // backspace
abcefd
> printf("abcd\tefd");  // horizontal tab
abcd    efd
> printf("abcd\"efd");  // double quote
abcd"efd
> printf("%c", '\'');   // single quote
'
> printf("??!")         // trigraph
|
> printf("?\?!")        // question mark
??!
>
```

数値エスケープコードには 2 つのバリエーションがあります。8 進数のエスケープコードと 16 進数のエスケープコードです。8 進数のエスケープコードは、バックスラッシュ文字\の後に指定する最大 3 桁の 8 進数で構成されます。たとえば、ASCII エンコーディングでは、文字'a' は'\141' と記述でき、文字列を終了させるために使用する null 文字は'\0' と記述できます。16 進数のエスケープコードは、'\x' 文字の後に指定する任意の桁数の 16 進数で構成されます。たとえば、文字'a' は 16 進数のエスケープコード'\x61' として記述できます。

これらの各エスケープシーケンスは、単一の **char** オブジェクトに格納できる一意の値を生成します。8 進数のエスケープコードは、8 進数の数字ではない最初の文字が出現した場合、または 3 桁の 8 進数が使用されている場合に終了します。そのため、文字列"\1111"は'\111' および'1' という 2 つの文字列を表し、文字列"\182"は'\1'、'8'、および'2' という 3 つの文字を表します。16 進数のエスケープシーケンスは任意の長さであり、非 16 進文字によってのみ終了されるので、文字列内の 16 進数のエスケープを停止するには、文字列を分割します。たとえば、コード'\x61' および'a' は 2 つの文字ですが、16 進のエスケープコード'\x61a' には、2 つの文字'a' ではなく 1 つの文字だけが含まれます。

6.3.2 文字列リテラル

文字の文字列リテラルは、"xyz"のように二重引用符で囲まれた一連の 0 個以上のマルチバイト文字です。文字の文字列リテラルまたはワイド文字列リテラル内のシーケンスの各要素には、整数文字定数またはワイド文字定数内で使用する場合と同じ考慮事項が適用されます。例外として、単一引用符' は、' またはエスケープシーケンス \' のどちらでも表現できますが、二重引用符"はエスケープシーケンス \"によって表す必要があります。

隣接する文字およびワイド文字列リテラルのトークンで指定されるマルチバイト文字シーケンスは、単一のマルチバイト文字シーケンスに連結されます。いずれかのトークンがワイド文字列リテラ

ルトークンである場合、結果のマルチバイト文字シーケンスはワイド文字列リテラルとして処理されます。そうでない場合は、文字の文字列リテラルとして処理されます。

バイトまたは値 0 のコードは、文字列リテラルまたは複数のリテラルの処理結果として得られる個々のマルチバイト文字シーケンスに追加されます。次に、マルチバイト文字シーケンスを使用して、静的な記憶期間とシーケンスを格納するために十分なだけの長さを持つ配列を初期化します。文字の文字列リテラルの場合、配列要素は **char** 型であり、マルチバイト文字シーケンスの個別のバイトを使用して初期化されます。

静的な記憶期間を持つ配列は異なります。たとえば、次の 1 組の隣接する文字の文字列リテラルは、

```
"A" "3"
```

'A' および'3' という値の 2 つの文字を含む単一文字の文字列リテラルを生成します。文字列の詳細については、第17章を参照してください。

ワイド文字列

ワイド文字列リテラルは、L"xyz"のように、二重引用符で囲まれ、文字 L が先頭に付く、一連の 0 個以上のマルチバイト文字です。

ワイド文字列リテラルの場合、配列要素は **wchar_t** 型であり、ワイド文字のシーケンスを使用して初期化されます。ワイド文字列はマルチバイト文字列によって外部的に表現できます。マルチバイト文字はコメント、文字列、および文字定数内に指定できます。通常の文字で構成される文字列内と同様に単一の null 文字 '\0' は、マルチバイト文字の文字列で終端子の役割をします。すべてのビットが 0 である 1 バイトは null 文字と解釈されるため、マルチバイト文字列の 2 番目のバイトまたはそれより後のバイトには指定されません。関数 **mbstowcs()** はマルチバイト文字をワイド文字の文字列に変換し、関数 **wcstombs()** はその反対に変換します。ワイド文字列の詳細については、第17章を参照してください。

6.3.3 整数定数

12345 などの 10 進数の整数定数は、int です。また、整数は 10 進数の代わりに 2 進数、8 進数または 16 進数でも指定できます。整数定数の先行する 0(ゼロ) は 8 進整数を示し、一方、先行する 0x または 0X は 16 進数を意味します。 また、 Ch および C99 では先行する 0b または 0B を含む 2 進定数をサポートしています。 たとえば、10 進数の 30 は 8 進では 036、16 進では 0X1e または 0x1E、および 2 進では 0b11110 または 0B11110 と記述できます。029 や 0b211 のような表現は無効であり、 Ch で検出できます。

Ch では、0 の値は整数のゼロを意味します。実数とは異なり、int では 0_ はありません。そのため、 Ch では数値 −0 は 0 に等しくなります。実数域 [−FLT_MAX, FLT_MAX] は整数域 [−INT_MIN, INT_MAX] より大きく、−Inf を含む INT_MIN より小さい実数は整数に変換され、結果は INT_MIN です。Inf を含む INT_MAX より大きい実数の場合、変換後の整数値は INT_MAX です。整数の変数に NaN(非数) が割り当てられると、システムは警告メッセージを出力し、結果の整数値はメモリマップが NaN(非数) のメモリマップと同じである INT_MAX になります。

10 進、8 進、および 16 進の整数定数に加えて、入出力用の 2 進整数定数および 2 進形式指定子がサポートされています。2 進定数の先頭はプレフィックス `0b` または `0B` です。形式指定子は`%b` です。次に例を示します。

```
/* Bit-map using binary constants */
#include<stdio.h>
int H[] = {
    0b00000000000000000000000000000000,
    0b00000000000000000000000000000000,
    0b00011111100000000000011111100000,
    0b00000110000000000000000110000000,
    0b00000110000000000000000110000000,
    0b00000110000000000000000110000000,
    0b00000110000000000000000110000000,
    0b00000110000000000000000110000000,
    0b00000110000000000000000110000000,
    0b00000110000000000000000110000000,
    0b00000111111111111111111110000000,
    0b00000111111111111111111110000000,
    0b00000110000000000000000110000000,
    0b00000110000000000000000110000000,
    0b00000110000000000000000110000000,
    0b00000110000000000000000110000000,
    0b00000110000000000000000110000000,
    0b00000110000000000000000110000000,
    0b00000110000000000000000110000000,
    0b00011111100000000000011111100000,
    0b00000000000000000000000000000000,
    0b00000000000000000000000000000000
}
int main() {
  int i, size;
  int I=0b00000110000000000000000110000000;

  size = sizeof(H)/sizeof(int);

  for (i=0; i<size; i++) {
    printf("H[%2d] = 0X%8x\n", i, H[i]);
  }
  /* H becomes II */
  H[10] = I;
  H[11] = I;
  for (i=0; i<22; i++) {
```

```
    printf("%32b\n", H[i]);
  }
  return 0;
}
```

6.3.4 浮動小数点定数

実数の定数

K&R C では、式内のすべての float は評価前に double に変換されます。結果として、浮動小数点のオペランドを使用する演算は、2 つのオペランドが float であっても double の結果を生成します。これは、プログラムの速度とメモリが重要である多くの科学計算に対しては適切ではありません。

C の初期設計における無差別変換ルールのため、3.5 や 3e7 などのすべての浮動小数点定数は double として取得されます。浮動小数点定数に対するこの既定の double モードは、標準 C に継承され、Ch でサポートされています。2.4、$2e+3$、$-2.E-3$、$+2.1e3$ など、すべての浮動小数点定数は既定では double 型定数です。ただし、C では float 型定数を指定するメカニズムが用意されています。サフィックス F または f は float 型定数を示し、D または d は double を示します。

たとえば、定数 $3.4e3F$、$3E-3f$、および $3e+3F$ は float ですが、定数 $3.4e3D$、$3E-3d$、および $3e+3D$ は double です。ただし、定数メタ数値 ±Inf および NaN は、double 型変数の値である場合を除き、常に float と解釈されます。これらの機能は Ch でもサポートされています。この設計により、表現可能な浮動小数点数の範囲を自動的に拡張できます。たとえば、SUN SPARCStations の FLT_MAX および DBL_MAX の値は、それぞれ $3.4e38$ と $1.8e308$ です。次のような Ch プログラムがあるとします。

```
printf("pow(10.0F, 39) < Inf is %d \n", pow(10.0F, 39) < Inf);
printf("pow(10.0, 39) < Inf is %d \n", pow(10.0, 39) < Inf);
```

出力は次のようになります。
pow(10.0F, 39) < Inf is 0
pow(10.0, 39) < Inf is 1

このプログラムの最初のステートメントでは、pow(10.0F, 39) によって計算された 10^{39} の値は、FLT_MAX より大きいため Inf としてオーバーフローされます。double 型データで **pow**(10.0, 39) によって計算される 10^{39} の値は、まだ −DBL_MAX < **pow**(10.0, 39) < DBL_MAX という表現可能な範囲内です。2 番目のステートメントでは、メタ数値 Inf を DBL_MAX より大きい double の無限大として拡張します。

16 進浮動小数点の定数

Ch は、C99 における 16 進浮動小数点の定数をサポートしています。たとえば、以下のとおりです。

```
> 0X2P3
16.0000
> 0x1.1p0
```

```
1.0625
> 0x1.1p1F
2.12
```

複素数の定数

複素数の定数は複素数コンストラクタ **complex**(x, y) によって形成できます。x は複素数の実数部で、y は虚数部です。関数 **complex()** の両方の引数が float 型または整数型である場合、結果の複素数は float の複素数です。1 つまたは 2 つの引数が double 型である場合は、結果の複素数は double 型の複素数です。次に例を示します。

```
complex z = complex(1, 3);          // complex(1, 3) is float complex
double complex z = complex(1.0, 3);// complex(1.0, 3) is
                                    // double a complex
```

また、Ch では、複素数の無限大および複素数の非数に該当する複素数のメタ数値 ComplexInf および ComplexNaN を使用できます。

ポインタの定数

ポインタに定数 0 および NULL を割り当てることができます。Ch では、定数 NULL は 2 つの目的を持つ組み込みシンボルであり、整数型とポインタ型の両方の変数に割り当てることができます。ゼロの代わりに使用します。たとえば、次のコードがあるとします。

```
> int i, *p
> p = &i
4005e758
> p = NULL
00000000
>
```

このコードは、最初に整数 i のアドレスをポインタ p に割り当て、次にそのポインタに定数 NULL を割り当てます。

6.4 初期化

変数の宣言には、変数の値を指定する初期化子が指定されている場合があります。初期化子は、変数の存続期間の開始時での変数の値を指定します。C での初期化のすべてのルールは、Ch に適用できます。ただし、3 次元より多い配列は Ch では初期化できません。
オブジェクトが自動または静的な記憶期間を持つ場合、そのオブジェクトが明示的に初期化されなければ、次のようになります。

- オブジェクトがポインタ型である場合、null ポインタに初期化されます。

- オブジェクトが算術型である場合、(正または符号なしの) ゼロに初期化されます。
- オブジェクトが集合体である場合、すべてのメンバはこのルールに従って (再帰的に) 初期化されます。
- オブジェクトが共用体である場合、最初の名前付きメンバがこれらのルールに従って (再帰的に) 初期化されます。

自動記憶期間を持つオブジェクトが明示的に初期化されない場合の Ch と C の違いは、C では値が不定ですが、Ch では上記の初期化ルールが適用されます。スカラの初期化子は、単一の式である必要があります。これはオプションでかっこで囲むことができます。オブジェクトの初期値が式 (変換後) の初期値です。単純な割り当ての場合と同じ型の制約と変換が適用され、スカラの型は宣言された型の修飾されないバージョンと解釈されます。次に例を示します。

```
> int i = 3.0/2
> i
1
>
```

変数 `i` は式 `3.0/2` の結果によって初期化され、その型は float から int に変換されます。

文字型の配列は、オプションの中かっこで囲まれた文字の文字列リテラルによって初期化できます。文字の文字列リテラルの連続する文字列 (余地がある場合または配列のサイズが不明である場合の終端の null 文字を含む) では配列の要素を初期化します。同様に、**wchar_t** と互換性のある要素型を含む配列は、オプションの中かっこで囲まれたワイド文字列リテラルによって初期化できます。たとえば、サイズが 80 バイトである配列 `str1` を文字列リテラル`"this is a string"`で初期化します。

```
> char str1[80] = "this is a string"
> str1
this is a string
>
```

サイズの不明な配列を初期化する場合は、明示的な初期化子を使用して、最大のインデックス付き要素によってサイズを決定します。初期化子リストの終わりになると、配列に不完全な型はもうありません。次に例を示します。

```
> char str2[] = "this is a string"
> str2
this is a string
>
```

char の配列のサイズは、文字列 “`this is a string`”に終端の null である 1 を足した長さと同じです。

集合体型または共用体型のオブジェクトの初期化子は、要素または指定されたメンバの初期化子を中かっこで囲んだリストである必要があります。たとえば、変数 `s1` は構造体のオブジェクトであり、中かっこで囲まれたリスト`{1 ,2}`によって初期化されます。メンバ `s1.a` は 1 に設定され、`s1.b` は 2 に設定されます。

```
> struct {int a, b;} s1 = {1, 2};
> s1
.a = 1
.b = 2
>
```

初期化は初期化子リストの順に実行されます。各初期化子は、特定のサブオブジェクト用に提供されます。このサブオブジェクトは、同じサブオブジェクトに対して前にリストされていた初期化子をオーバーライドします。明示的に初期化されないすべてのサブオブジェクトは暗黙的に初期化される必要があります。

集合体または共用体に含まれる要素またはメンバが集合体または共用体である場合、これらのルールは、サブ集合体または内側の共用体に再帰的に適用されます。サブ集合体または含まれる共用体の初期化子の先頭が左中かっこである場合、その中かっこおよび対応する右中かっこで囲まれた初期化子は、サブ集合体または内側の共用体の要素またはメンバを初期化します。その他の場合、リストにある初期化子のうち、サブ集合体の要素またはメンバあるいは内側の共用体の最初のメンバに対して十分な初期化子のみが考慮されます。残りの初期化子は、現在のサブ集合体または内側の共用体がその一部になっている集合体の次の要素またはメンバを初期化するために残されます。たとえば、次の宣言があるとします。

```
int y[3][3] = {
    { 1, 3, 5 },
    { 2, 4, 6 },
    { 3, 5, 7 },
};
```

この宣言は、完全にかっこで囲まれた初期化を使用した定義です。1、3、および 5 は、最初の行 `y`、つまり配列オブジェクト `y[0]` を初期化します。同様に、次の 2 行では、`y[1]` と `y[2]` を初期化します。次の宣言

```
int y[3][3] = {
  1, 3, 5, 2, 4, 6, 3, 5, 7
};
```

では、`y[0]` の初期化子の先頭が左中かっこではないため、リストの 3 つの項目を使用します。同様に、次の 3 つが `y[1]` および `y[2]` に対して連続して取得されます。これには、前の完全にかっこで囲まれた初期化と同じ効果があります。次のコマンドがあるとします。

```
> struct {int a[3], b;} s2[] = {{1}, 2}
> s2[0]
.a = 1 0 0
.b = 0
> s2[1]
.a = 2 0 0
.b = 0
>
```

ここでは、初期化子の一つ（{1}）にのみ中かっこを使用しています。２つの構造体からなる配列 s2 を定義し、s2[0].a[0] の値は 1、s2[1].a[0] の値は 2、その他すべての要素の値はゼロです。

中かっこで囲まれたリスト内の初期化子が集合体の要素またはメンバより少ないか、既知のサイズの配列の初期化に使用する文字列リテラル内の文字数が配列の要素数より少ない場合、集合体の残りの部分は、静的な記憶期間を持つオブジェクトと同様に、暗黙的に初期化される必要があります。

```
> struct {int a, b;} s3 = {1}
> s3
.a = 1
.b = 0
>
```

最初のメンバが 1 として初期化されると、その他は暗黙的に 0 として初期化されます。

Ch では、非定数の式、汎用関数、および関数ファイルで定義されている関数を、静的な持続期間を持つオブジェクトおよび動的な持続期間を持つオブジェクトの両方の初期化子として使用できます。ただし、1 つ例外があります。関数ファイルで定義されている関数を静的変数の初期化子として使用することはできません。関数またはブロックの有効範囲を以下のコードに示します。関数ファイル hypot.chf で定義されている関数 hypot() は、関数の有効範囲内の静的変数である識別子 d1 の初期化には使用できません。

```
#include <math.h>
int main () {
    static double d = hypot(3,4);  // Error: hypot is not generic function
}
```

第7章　演算子と式

Ch で使用される演算子を表7.1に示します。各演算子は、それより下のレベルにある他の演算子に優先します。同じレベルにある演算子の優先順位は同じです。優先順位が同じ演算子は、それぞれの結合性に従ってオペランドを結合します。単項演算子、三項条件演算子、およびコンマ演算子は右結合です。それ以外のすべての演算子は左結合です。

表 7.1: 演算子の優先順位と結合性

<table>
<tr><th>演算子</th><th>結合性</th></tr>
<tr><td>::</td><td></td></tr>
<tr><td>( ) [ ]</td><td>左から右</td></tr>
<tr><td>function_name( )</td><td>右から左</td></tr>
<tr><td>. -></td><td>左から右</td></tr>
<tr><td>` ! ~ ++ -- + - * & (type)</td><td>右から左</td></tr>
<tr><td>* / % .* ./</td><td>左から右</td></tr>
<tr><td>+ -</td><td>左から右</td></tr>
<tr><td><< >></td><td>左から右</td></tr>
<tr><td>< <= > >=</td><td>左から右</td></tr>
<tr><td>== !=</td><td>左から右</td></tr>
<tr><td>&</td><td>左から右</td></tr>
<tr><td>^</td><td>左から右</td></tr>
<tr><td>|</td><td>左から右</td></tr>
<tr><td>&&</td><td>左から右</td></tr>
<tr><td>^^</td><td>左から右</td></tr>
<tr><td>||</td><td>左から右</td></tr>
<tr><td>?:</td><td>右から左</td></tr>
<tr><td>= += -= *= /= %= &= |= <<= >>=</td><td>右から左</td></tr>
<tr><td>,</td><td>左から右</td></tr>
</table>

Ch におけるさまざまな演算子の演算の優先順位は、標準 C とまったく同じです。Ch では、排他 OR 演算子^^、コマンド置換演算子`、配列乗算演算子'.*'、および配列除算演算子'./' が導入されています。標準 C 同様、 Ch における浮動小数点数の演算で使用されるアルゴリズムと演算後のデータ型は、オペランドのデータ型に依存します。Ch における char、int、float、および double の変換規則は、標準 C に定義された型変換規則に従います。使用メモリ領域が少ないデータ型は、情報を失うことなく使用メモリ領域の多いデータ型に変換できます。

たとえば、char 整数は、問題なく int または float にキャストできます。ただし、逆に変換すると、情報が失われる可能性があります。Ch における実数の順位は、char、int、float、double です。char データ型が最も低く、double データ型が最も高くなります。C と同じように、 Ch における演算のアルゴリズムと結果のデータ型は、オペランドのデータ型によって決まります。加算、減算、乗算、除算などの二項演算では、結果のデータ型は、2 つのオペランドの順位が高い方のデータ型になります。

たとえば、2 つの float 型数値の加算結果は float 型数値になり、float 型数値と double 型数値の加算結果は double 型数値になります。

Ch における通常の実数とメタ数値の演算規則を表7.2〜7.12に示します。表7.2〜7.12では、x、x1、および x2 は、float または double での通常の正の正規化された浮動小数点数です。メタ数値 0.0、−0.0、Inf、−Inf、および NaN は、定数、float 型変数の値、または double 型変数の値です。既定では、定数のメタ数値は float 型定数です。

表 7.2: 否定の結果

否定 −							
オペランド	−Inf	−x1	−0.0	0.0	x2	Inf	NaN
結果	Inf	x1	0.0	−0.0	−x2	−Inf	NaN

表 7.3: 加算の結果

加算 +							
左オペランド	右オペランド						
	−Inf	−x1	−0.0	0.0	x2	Inf	NaN
Inf	NaN	Inf	Inf	Inf	Inf	Inf	NaN
y2	−Inf	y2−x1	y2	y2	y2+x2	Inf	NaN
0.0	−Inf	−x1	0.0	0.0	x2	Inf	NaN
−0.0	−Inf	−x1	−0.0	0.0	x2	Inf	NaN
−y1	−Inf	−y1−x1	−y1	−y1	−y1+x2	Inf	NaN
−Inf	−Inf	−Inf	−Inf	−Inf	−Inf	NaN	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

表 7.4: 減算の結果

減算 −							
左オペランド	右オペランド						
	−Inf	−x1	−0.0	0.0	x2	Inf	NaN
Inf	Inf	Inf	Inf	Inf	Inf	NaN	NaN
y2	Inf	y2+x1	y2	y2	y2−x2	−Inf	NaN
0.0	Inf	x1	0.0	0.0	−x2	−Inf	NaN
−0.0	Inf	x1	0.0	−0.0	−x2	−Inf	NaN
−y1	Inf	−y1+x1	−y1	−y1	−y1−x2	−Inf	NaN
−Inf	NaN	−Inf	−Inf	−Inf	−Inf	−Inf	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

表 7.5: 乗算の結果

乗算 ∗							
左オペランド	右オペランド						
	−Inf	−x1	−0.0	0.0	x2	Inf	NaN
Inf	−Inf	−Inf	NaN	NaN	Inf	Inf	NaN
y2	−Inf	−y2∗x1	−0.0	0.0	y2∗x2	Inf	NaN
0.0	NaN	−0.0	−0.0	0.0	0.0	NaN	NaN
−0.0	NaN	0.0	0.0	−0.0	−0.0	NaN	NaN
−y1	Inf	y1∗x1	0.0	−0.0	−y1∗x2	−Inf	NaN
−Inf	Inf	Inf	NaN	NaN	−Inf	−Inf	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

表 7.6: 除算の結果

除算 /							
左オペランド	右オペランド						
	−Inf	−x1	−0.0	0.0	x2	Inf	NaN
Inf	NaN	−Inf	NaN	NaN	Inf	NaN	NaN
y2	−0.0	−y2/x1	−Inf	Inf	y2/x2	0.0	NaN
0.0	−0.0	−0.0	NaN	NaN	0.0	0.0	NaN
−0.0	0.0	0.0	NaN	NaN	−0.0	−0.0	NaN
−y1	0.0	y1/x1	Inf	−Inf	−y1/x2	−0.0	NaN
−Inf	NaN	Inf	Inf	−Inf	−Inf	NaN	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

7.1 算術演算子

表7.2に示した否定演算では、結果のデータ型は、オペランドのデータ型と同じであり、実数ではその記号が否定演算によって変更されます。Ch には −NaN は存在しません。$+57864 - x$ のような式に含まれる先行するプラス記号 '+' (sign 演算子) は無視されます。正の整数ゼロの否定は、やはり正のゼロであることに注意してください。前述した負の整数を 2 の補数で表す表現に基づいて、Inf と NaN は int データ型で表すことができません。

IEEE 754 規格に従い、一部の演算は丸めモードに依存します。たとえば、ゼロに向かう丸めの場合、オーバーフローすると、Inf ではなく、適切な符号を付けた FLT_MAX が得られます。この丸めモードは、Fortran を実装するため、および無限大がないマシンのために必要です。丸めモードが $-\infty$ に向かって丸められる場合、$-0.0 + 0.0$ と $0.0 - 0.0$ では、両方とも 0.0 ではなく −0.0 が得られます。科学プログラミングでは、一貫性と確定性が不可欠です。現在、 Ch は、オーバーフロー時に Inf になるように、近似値に丸めるという既定の丸めモードを使用して実装されているので、表7.3と7.4に示すように、$-0.0 + 0.0$ と $0.0 - 0.0$ では、両方とも 0.0 が渡されます。Ch のモジュロ演算子%は、C と互換性があります。

表7.3〜7.6に示す加算、減算、乗算、除算の各演算では、2 つのオペランドのいずれかが double であれば、結果のデータ型は double になります。それ以外の場合は float 型になります。$\infty - \infty$、$\infty * 0.0$、∞/∞、0.0/0.0 などの数学的に不定である式の結果は NaN になります。値 ±0.0 は、乗算演算と除算演算で重要な役割をします。たとえば、正の有限値 x2 を 0.0 で除算した場合、結果は正の無限大 $+\infty$ になり、−0.0 で除算した場合は負の無限大 $-\infty$ になります。二項算術演算子のいずれかのオペランドが NaN の場合、結果は NaN です。

2 つの計算配列の要素に関する乗算と除算は、配列乗算演算子'`.*`' と配列除算演算子'`./`' をそれぞれ使用することで実行できます。配列乗算演算子'`.*`' と配列除算演算子'`./`' の計算配列用のオペランドについては、第16章を参照してください。

7.2 関係演算子

表7.7〜7.12に示す関係演算子の結果は、常に、それぞれ TRUE または FALSE に該当する論理値 1 または 0 である整数になります。これらの値は、事前定義されたシステム定数です。IEEE 754 規格に従って、浮動小数点数の +0.0 と −0.0 は区別されます。

Ch では、値 0.0 は、実数線に沿って正数側からゼロに近づいていき、ゼロになっていることを意味します。値 −0.0 は、数直線に沿って負数側からゼロに近づいていき、多くの場合、限りなく 0.0 より小さいことを意味します。Ch プログラムでの符号付きの +0.0 と −0.0 は、それぞれ適切な符号付きの無限小量 0_+ と 0_- のように動作します。

ただし、多くの演算では、浮動小数点数の −0.0 と 0.0 は、IEEE 754 規格に従って区別されますが、比較では関係演算で −0.0 と 0.0 が同等になるように、ゼロの符号は無視するものとします。

signbit(x) や copysign(x、y) などの関数 を使用して式の符号を処理できます。

Ch の比較演算では、値 −0.0 は 0.0 と異なると見なすことができます。C コードを Ch に移植するための便宜上、ゼロは比較演算では符号が付きません。メタ数値の同等性は、 Ch では意味が異なります。2 つの同一のメタ数値は、互いに等しいと見なされます。

その結果、2 つの Inf または 2 つの NaN を比較すると、論理 TRUE が得られます。数学的に、無限大 ∞ と非数値の NaN は互いに比較できない未定義値なので、これは単なるプログラミング上の都合

です。Ch のメタ数値 Inf、−Inf、および NaN は、算術演算、関係演算、および論理演算では、通常の浮動小数点数として処理されます。

表 7.7: より小さい比較の結果

より小さい比較 <							
左オペランド	右オペランド						
	−Inf	−x1	−0.0	0.0	x2	Inf	NaN
Inf	0	0	0	0	0	0	0
y2	0	0	0	0	y2 < x2	1	0
0.0	0	0	0	0	1	1	0
−0.0	0	0	0	0	1	1	0
−y1	0	−y1 < −x1	1	1	1	1	0
−Inf	0	1	1	1	1	1	0
NaN	0	0	0	0	0	0	0

表 7.8: 以下比較の結果

以下比較 <=							
左オペランド	右オペランド						
	−Inf	−x1	−0.0	0.0	x2	Inf	NaN
Inf	0	0	0	0	0	1	0
y2	0	0	0	0	y2 <= x2	1	0
0.0	0	0	1	1	1	1	0
−0.0	0	0	1	1	1	1	0
−y1	0	−y1 <= −x1	1	1	1	1	0
−Inf	1	1	1	1	1	1	0
NaN	0	0	0	0	0	0	0

表 7.9: 等しい比較の結果

等しい比較 ==							
左オペランド	右オペランド						
	−Inf	−x1	−0.0	0.0	x2	Inf	NaN
Inf	0	0	0	0	0	1	0
y2	0	0	0	0	y2 == x2	0	0
0.0	0	0	1	1	0	0	0
−0.0	0	0	1	1	0	0	0
−y1	0	−y1 == −x1	0	0	0	0	0
−Inf	1	0	0	0	0	0	0
NaN	0	0	0	0	0	0	0

表 7.10: 以上比較の結果

以上比較 >=							
左オペランド	右オペランド						
	−Inf	−x1	−0.0	0.0	x2	Inf	NaN
Inf	1	1	1	1	1	1	0
y2	1	1	1	1	y2 >= x2	0	0
0.0	1	1	1	1	0	0	0
−0.0	1	1	1	1	0	0	0
−y1	1	−y1 >= −x1	0	0	0	0	0
−Inf	1	0	0	0	0	0	0
NaN	0	0	0	0	0	0	0

表 7.11: より大きい比較の結果

より大きい比較 >							
左オペランド	右オペランド						
	−Inf	−x1	−0.0	0.0	x2	Inf	NaN
Inf	1	1	1	1	1	0	0
y2	1	1	1	1	y2 > x2	0	0
0.0	1	1	0	0	0	0	0
−0.0	1	1	0	0	0	0	0
−y1	1	−y1 > −x1	0	0	0	0	0
−Inf	0	0	0	0	0	0	0
NaN	0	0	0	0	0	0	0

表 7.12: 等しくない比較の結果

等しくない比較 !=							
左オペランド	右オペランド						
	−Inf	−x1	−0.0	0.0	x2	Inf	NaN
Inf	1	1	1	1	1	0	1
y2	1	1	1	1	y2 != x2	1	1
0.0	1	1	0	0	1	1	1
−0.0	1	1	0	0	1	1	1
−y1	1	−y1 != −x1	1	1	1	1	1
−Inf	0	1	1	1	1	1	1
NaN	1	1	1	1	1	1	1

片方のオペランドが内蔵文字列型 **string_t** である 2 つの文字列に対する関係演算子==、!=、<、および > については、セクション 6.1.13で説明されています。

7.3 論理演算子

Ch には!、&&、||、および^^という 4 つの論理演算子があり、それぞれ NOT、AND、包含 OR、および排他 OR に対応します。 Ch での演算!、||、&&は、標準 C に従います。演算子^^は、論理排他 OR 演算子とビットごとの排他 OR 演算子が直交するように、プログラミング上の便宜を考慮して Ch に導入されています。

C と同じように、Ch では、&&演算と||演算の両方で、左オペランドがそれぞれ TRUE および FALSE と評価された場合のみ、右オペランドを評価します。^^演算子では、最初のオペランドが TRUE でも FALSE でも、第 2 のオペランドによっては排他 OR 演算で TRUE が返される可能性があるので、この”短絡的”な動作は存在しません。

演算子^^の優先順位は、||演算子より高く、&&より低くなります。この演算の優先順位は、次のセクションで説明するビットごとの演算子&、|、および^の優先順位に類似しています。論理演算には TRUE または FALSE という 2 つの値しかないので、値 ±0.0 は論理 FALSE として処理され、メタ数値 −Inf、Inf、および NaN は論理 TRUE と見なされます。たとえば、!(−0.0) と!NaN を評価すると、それぞれ値 1 と 0 が得られます。

7.4 ビットごとの演算子

Ch には、ビットごとの演算子として&、|、^、<<、>>、および~の 6 つがあり、それぞれビットごとの AND、包含 OR、排他 OR、左シフト、右シフト、および補数に対応します。Ch のこれらの演算子は、標準 C と完全に互換性があります。Ch の現在の実装では、これらの演算子は char と int の整数データに対してのみ適用できます。返されるデータ型は、オペランドのデータ型に依存します。

単項演算子˜の結果は、そのオペランドのデータ型を維持します。二項演算子&、|、および^の結果は、2 つの演算子の優先順位の高い方のデータ型になります。二項演算子<<と>>は、左オペランドのデータ型を返します。

ただし、C では未定義であるいくつかの動作が Ch には定義されています。演算子<<と>>について、右オペランドには、内部的に int に変換できる限り任意のデータ型を使用できますが、C の場合、右オペランドは正の整数でなければなりません。 Ch では、浮動小数点データから変換される場合がある負の整数値が右オペランドであるとき、シフト方向が逆転します。

たとえば、 Ch では、式 $7 << -2.0$ は $7 >> 2.0$ に相当します。したがって、Ch では、これらのシフト演算子は片方だけが必要です。Ch プログラミングでは、演算子<<の使用をお勧めします。1 つの演算子で二方向のシフトを使用する方が、2 つの演算子それぞれに一方向のシフトを使用する場合よりもわかりやすい可能性があります。

7.5 代入演算子

通常の代入ステートメントに加え、9 つの代入演算子 (+=、−=、*=、=、&=、|=、˜˜=、<<=、および>>=) があります。これらの代入演算子は C と互換性があります。

lvalue は、代入ステートメントの左辺に出現する任意のオブジェクトです。lvalue は、関数や定数ではなく、変数やポインタなどのメモリを参照します。

Ch 式 `lvalue op= rvalue` は、`lvalue = lvalue op rvalue` と定義されます。ここで、`lvalue` は、複素数を含む任意の有効な lvalue であり、一度だけ評価されます。

たとえば、`i += 3` は `i = i+3` と同等であり、**real**`(c) *= 2` は `real(c) = real(c)*2` と同じです。ただし、ステートメント `*ptr++ += 2` とステートメント `*ptr++ = *ptr++ +2` は、値 `*ptr++`に増分演算子が含まれているので同じではありません。演算子+、−、*、/、&、|、^、<<、および>>の演算規則については、前のセクションに説明があります。

7.6 条件演算子

条件演算子'`?:`' は、Ch に条件式を導入します。次の条件式

```
r = op1 ? op2 : op3;
```

は以下と同等です。

```
if(op1 != 0)
  r = op2;
else
  r = op3;
```

条件式では、1 番目と 2 番目のオペランドは疑問符'`?`' によって分割され、2 番目と 3 番目のオペランドはコロン'`:`' によって分割されます。条件式の実行は以下のように進行します。

1. 1 番目のオペランドが評価されます。

2. 2 番目のオペランドは、1 番目のオペランドの評価が 0 にならない場合にのみ評価されます。3 番目のオペランドは、1 番目のオペランドの評価が 0 になる場合にのみ評価されます。
3. 結果は、評価された 2 番目または 3 番目のオペランドの値になります。

条件式の最初のオペランドは、スカラ型でなければなりません。2 番目と 3 番目のオペランドは、以下のいずれかに該当する必要があります。

1. 両方のオペランドが算術型である。結果の型は、通常の算術変換によって決定されます。
2. 両方のオペランドが、互換性のあるクラス、構造体、または共用体型である。結果は、クラス、構造体、または共用体型です。
3. 両方のオペランドが **void** 型である。結果は、**void** 型になります。
4. 両方のオペランドが、互換性のある型へのポインタである。結果は、複合型へのポインタです。
5. 片方のオペランドがポインタであり、他方が NULL である。結果は、NULL ではない型のオペランドです。
6. 片方のオペランドがオブジェクトへのポインタまたは不完全な型であり、他方が **void** へのポインタである。結果は、**void** へのポインタです。One operand is a pointer to an object or incomplete type and the other is a pointer to **void**. The result is a pointer to **void**.
7. 両方のオペランドが、同形の計算配列である。結果は、2 つのオペランドの優先順位が高い方のデータ型の計算配列です。

条件式は、右結合です。次に例を示します。

```
op1 ? op2 : op3 ? op4 : op5 ? op6: op7
```

これは、次のように処理されます。

```
op1 ? op2 : (op3 ? op4 : (op5 ? op6: op7))
```

以下のコマンドは、オペランドが計算配列型である条件式の例です。

```
> 5 ? 1 : 2
1
> 0 ? 1 : 0 ? 3 : 4 // right-association
4
> 0 ? 1.0 : 2       // data type conversion
2.0000
> 1 ? (array float [2][3])1 : (array int [2][3])2
1.00 1.00 1.00
1.00 1.00 1.00
> 0 ? (array float [2][3])1 : (array int [2][3])2
2.00 2.00 2.00
2.00 2.00 2.00
```

プログラム7.1では、引数として条件式の結果が渡される関数 func() が、メイン関数の中で呼び出されます。

```
struct tag {
  int ii;
  int *pp;
} s1, s2, *ps1, *ps2;

int func(int i) {
  printf("i = %d\n", i);
  return 0;
}

int main() {
  int i = 1;
  int *p1 = NULL, *p2 = &i;

  func(i? 5 : 8);         // passed as argument of function

  p1  =  (p1)? p1 : p2;   // oparands is pointers
  printf("p1 = %p\n", p1);

  ps1 = &s1;
  ps2 = &s2;
  s1.ii = 10;
  s2.pp = &s1.ii;
  i  = (1 ? s1 : s2).ii;      // operands of structure
  p1  = (0 ? ps1 : ps2)->pp;
  printf("i = %d\n", i);
  printf("*p1 = %d\n", *p1);
}
```

プログラム 7.1: データ型が異なるオペランドを使用する条件式の例

条件式では、ポインタ p1 と p2 がオペランドとして使用されます。

```
p1  =  (p1)? p1 : p2;
```

次に、条件式の中で、構造体 s1 と s2 がオペランドとして使用されます。

```
i  = (1 ? s1 : s2).ii;
```

プログラム7.1の出力が、プログラム7.2に表示されます。

```
i = 5
p1 = 40063528
i = 10
*p1 = 10
```

プログラム 7.2: プログラム7.1の出力

7.7 キャスト演算子

7.7.1 キャスト演算子

Ch では、C であれば明示的な型変換を必要とする場合でも、多くの場合、明示的な型変換は必要ありません。たとえば、
`aptr[3] = malloc(90)` は、Ch では有効です。

ただし、ある型の値を別のデータ型に明示的に変換しなければならない場合があります。これは、従来の C のキャスト演算 `(type)expr` によって実現できます。この `expr` は Ch 式であり、`type` は、`char`、`int`、`float`、`double` などの単一のオブジェクトのデータ型、または `char *`、`double *`、`complex *`などの任意のポインタ宣言識別子です。たとえば、(**int**)9.3、(**float**)ptr、(**double**)9、(**float***)&i、および (**complex***)iptr は、有効な Ch 式です。

sizeof() 関数でも、型識別子を使用できます。次に例を示します。

```
ptr = malloc(5+sizeof(int*)+sizeof((int)2.3) + sizeof((int)float(90)+7))
```

これは、有効な Ch ステートメントです。

C の重要な機能の 1 つは、コンピュータのメモリ上の特定の場所にアクセスすることによるハードウェアインタフェース機能です。これは、メモリ上の特定の場所またはレジスタをポインタで参照することによって実現します。Ch にも、このハードウェアインタフェース機能があります。たとえば、以下のステートメントは、メモリアドレス $(68FFE)_{16}$ の整数値を変数 i に代入し、メモリアドレス $(FF000)_{16}$ のバイトを $(01101001)_2$ に設定します。

```
char *cptr;
int i, *iptr, j;
iptr = (int *)0X68FFE;    // point to the memory location at 0X68FFE
i = *iptr;                // i equals the value at 0X68FFE;
cptr = (char *)0XFF000;   // point to the memory location at 0XFF000
*cptr = 0B01101001;       // 0B01101001 is assigned to 0XFF000
cptr = (float *)cptr + 1;// cptr  points to 0XFF004, not 0XFF001.
                          // note: (float *)cptr++ is (float *)(cptr++)
j = int(cptr);            // j becomes 0XFF004
```

整数値は、明示的に型キャストを行わないとポインタ変数に代入できないことに注意してください。逆も場合も同じです。コンピュータのメモリの下位セグメントは、通常はオペレーティングシステムとシステムプログラムのために予約されています。これらの保護されたメモリセグメントがポインタによって破壊された場合、アプリケーションプログラムは例外処理によって終了させられます。

7.7.2 関数型キャスト演算子

Ch には、単一オブジェクトのデータ型では `type(expr)` 形式、複合型などの集約データ型では `type(expr1, expr2, ...)` 形式の関数型キャスト演算が追加されています。この関数型キャスト演算では、`type` はポインタデータ型であってはなりません。

たとえば、 **int**(9.3)、**complex**(**float**(3), 2)、および **complex**(**double**(3), 2) は、有効な Ch 式です。オペランドとして使用する場合、**float**() 演算と **real**() 演算は同じです。ただし、関数 **real**() は lvalue とし

て使用できますが、関数 **float**() は使用できません。関数 **real**() の詳細については、セクション13.6を参照してください。関数型キャスト演算の例を以下に示します。

```
char char(double)
char char(complex)
char char(pointer_type)
complex complex(float, float)
double complex complex(double, float)
double complex complex(float, double)
double complex complex(double, double)
double double(double)
double double(complex)
double double(pointer_type)
float float(double)
float float(complex)
float float(pointer_type)
int int(double)
int int(complex)
int int(pointer_type)
long long(double)
long long(complex)
long long(pointer_type)
short short(double)
short short(complex)
short short(pointer_type)
signed signed(double)
signed signed(complex)
signed signed(pointer_type)
unsigned unsigned(double)
unsigned unsigned(complex)
unsigned unsigned(pointer_type)
```

7.8 コンマ演算子

Ch では、コンマ演算子',' を使用するコンマ式が導入されています。コンマ式は、コンマで区切られた 2 つの式で構成されます。次に例を示します。

```
a = 1, ++a;
```

コンマ演算子は、構文的には左結合です。次の式

```
a = 1, ++a, a + 10;
```

は以下と同等です。

```
((a = 1), ++a), a + 10;
```

コンマ演算子の左オペランドが、最初に void 式として評価されます。次に、右オペランドが評価されます。結果は、その型と値になります。次に例を示します。

```
> a = 1, ++a, a + 10
12
```

コンマ演算子は、コンマが関数の引数リストなどで項目を分離するために使用される場所では、そのままでは使用できません。このような場合は、かっこで囲んで使用します。次に例を示します。

```
int func(int i1, int i2);
int t;
...
func((t = 1, t + 2), 2);
```

7.9　単項演算子

7.9.1　アドレスと間接演算子

単項演算子&は、オブジェクトのアドレスを与えます。演算子&は C と互換性があり、有効な lvalue に対してのみ適用できます。

単項間接演算子*は、ポインタに適用された場合、そのポインタの参照先オブジェクトにアクセスします。ポインタと整数は、加算または減算できます。

たとえば、ポインタ型の変数 ptr、ptr1、および ptr2 と整数値 n の場合、式 ptr+n は、ptr の現在の参照先オブジェクトから n 番目のアドレスを与えます。ポインタ ptr+n と ptr のメモリ上での場所は、n*sizeof(*ptr) バイト離れています。つまり、n は、ポインタ変数 ptr の宣言に従って、n*sizeof(*ptr) バイトに調整されます。

データ型が同じポインタのポインタ減算は許可されます。ptr1 > ptr2 の場合、ptr1 − ptr2 は、ptr2 と ptr1 の間にあるオブジェクトの数を与えます。ポインタの配列も宣言できます。ポインタは、宣言時にゼロに初期化されます。

プログラムでは、ゼロの代わりにシンボリック定数 NULL を使用できます。ptr が NULL の場合、式のオペランド*ptr は、ゼロと評価されます。ptr が NULL である*ptr が lvalue として使用された場合、**sizeof**(*ptr) のメモリが、ポインタ ptr に対して自動的に割り当てられます。どちらの場合も、システムは警告メッセージを出力します。有効なメモリ上の場所を参照していないポインタにメモリを自動的に割り当てることによって、システムクラッシュを回避できます。

関係演算<、<=、==、=、>、および!=の中で、2 つのポインタと定数 NULL を使用できます。代入演算と関係演算では、明示的な型変換なしで、型が異なるポインタを一緒に操作できます。たとえば、以下は有効な Ch プログラムです。

```
int *iptr;
float *fptr;
iptr = (int *)malloc(90);
fptr = malloc(80); // fptr = (float *)malloc(80)
```

```
if(iptr != NULL && iptr != fptr)
  free(iptr);
iptr = fptr;
```

Ch では、すべての変数の宣言時に変数がゼロに初期化されるだけではなく、関数 **malloc**()、**calloc**()、または **realloc**() のいずれかによって割り当てられるメモリもゼロに初期化されます。これにより、予期しないエラーを回避できます。

C では、関数 **malloc**() と **realloc**() によって割り当てたメモリの内容はランダムな値になります。さらに、3 つのメモリ割り当て関数 **malloc**()、**calloc**()、および **realloc**() のキャスト演算は、Ch では省略できます。使用可能なメモリがない場合、これらの関数は NULL を返し、システムはエラーメッセージを出力します。

関数 **free**(ptr) は、これら 3 つの関数によって割り当てられたメモリの割り当てを解除し、ポインタ `ptr` を NULL に設定します。C では、`ptr` が参照しているメモリの割り当てが解除されても、ptr に NULL は設定されません。この宙に浮いたぶら下がりメモリは、C プログラムのデバッグを非常に困難にします。その理由は、割り当てを解除されたメモリがプログラムの他の部分によって再度要求されるまで、問題が表面化しないためです。 Ch におけるその他のメモリ操作関数 (**memcpy**() など) は、C と互換性があります。

前に説明したように、NaN、Inf、FLT_MAX、INT_MIN、FLT_EPSILON などのさまざまなシステム定義パラメータがあります。これらのパラメタの値は、予期しない変更を回避できるように、lvalue として使用することはできません。ただし、どうしても必要な場合は、ポインタを通してメモリ上の場所にアクセスすることによって、これらのパラメタの値を変更できます。たとえば、数値アルゴリズムが、FLT_EPSILON パラメータと Inf パラメータに依存するとします。次の Ch コードにより、FLT_EPSILON の値を 10^{-4} に、Inf の値を FLT_MAX に変更できます。

```
float *fptr;
fptr = & FLT_EPSILON; *fptr = 1e-4;
fptr = &Inf; *fptr = FLT_MAX;
```

このコードは、実質的に、基礎になる数値アルゴリズムを変更する可能性があります。

7.9.2 増分演算子と減分演算子

C は、構文が簡潔であることがよく知られています。増分演算子++と減分演算子--は、C 独自の演算子です。Ch におけるこれらの 2 つの演算子は、C と互換性があります。増分演算子++はオペランドに 1 を加算し、減分演算子--は 1 を減算します。++または--を前置演算子として使用した場合、式のオペランドは、値が使用される前にそれぞれ加算または減算されます。後置演算子として使用した場合は、値が使用された後に演算が実行されます。

単一の + は、状況に応じて加算演算子または単項プラス演算子として処理されます。同様に、単一の − は、減算演算子または単項否定演算子になることができます。たとえば、以下は有効な Ch コードです。

```
i = +(-9);             // unary plus and negation operators
i++;                   // i = i+1
```

```
j = ++i--;             // i = i+1; j = i; i = i-1;
j = ++i;               // i = i+1; j = i;
j = i--;               // j = i; i = i-1;
i = (*ptr++)++;        // ptr = ptr + 1; i = *ptr; *ptr = *ptr + 1;
```

定義により、++lvalue は lvalue = lvalue + 1 と式 lvalue + 1 を意味し、lvalue--は式 lvalue - 1 と lvalue = lvalue - 1 に相当します。++演算子と--演算子は、lvalue が内部データ変換規則に従って整数値 1 を加算または減算できる限り、整数変数だけではなく、任意の有効な lvalue に適用できます。以下は有効な Ch コードです。

```
int i, a[4], *aptr[5];
complex z, *zptr;       // declare complex variable and complex pointer
z = z++;                // z = z + 1; z is a complex variable
zptr = (complex *)malloc(sizeof(complex)*90);
aptr[3] = malloc(90);   // aptr[3] = (int *)malloc(90);
/* imag(z)=complex(0.0, 4.0); zptr=zptr+1; *aptr[3]=1; i=i-1 */
imag(z) = ++real(++++*(zptr+++2*(int)real(++*aptr[3+i--])));
real(z)++;              // real(z) = real(z) + 1;
--imag(*zptr);          // imag(*zptr) = imag(*zptr) - 1;
a[--i] = a[2]++;        // i = i - 1; a[i] = a[2]; a[2] = a[2] + 1;
```

Ch における複素数、関数 **real()**、および関数 **imag()** の詳細については、セクション13.6を参照してください。関数 **malloc()** によって割り当てられるメモリは、ゼロに初期化されることに注意してください。

7.9.3 コマンド置換演算子

コマンド置換演算子`は、コマンドからの出力を文字列として返します。次に例を示します。

```
  string_t s;
s = `ls`;
```

2 つのコマンド置換演算子を一緒に使用すると、コマンドの出力の改ページ文字、改行文字、復帰文字、水平タブ文字、および垂直タブ文字は、空白のスペース文字に置換されます。次に例を示します。

```
  string_t s;
s = ``ls``;
```

7.10 メンバ演算子

. 演算子と−>演算子は、メンバ演算子と呼ばれます。クラス、構造体、または共用体のメンバは、これら 2 つのメンバ演算子によって参照されます。**.** 演算子の最初のオペランドは、クラス、構造体、または共用体のいずれかの型でなければならず、2 番目のオペランドは、その型のメンバを指定する必要があります。
−>演算子の最初のオペランドの型は、“クラスへのポインタ”、“構造体へのポインタ”、または “共用体へのポインタ” でなければならず、2 番目のオペランドは、参照されているメンバを指定する必要

があります。
次に例を示します。

```
struct tag {
    int i;
    double d;
} s, *p;
s = &p;
s.i = 10;
p->i += s.i;
```

第8章　ステートメントと制御フロー

ステートメントは、実行する動作を指定します。

指定された場合を除いて、ステートメントは連続して実行されます。完全式とは、別の式または宣言子の一部ではない式です。初期化子、式ステートメント内の式、選択ステートメント (if または switch) の制御式、**while** または **do** ステートメントの制御式、**for** ステートメントの (オプションの) 各式、**return** ステートメントの (オプションの) 各式などは、それぞれが完全式です。完全式の終わりはシーケンスポイントです。

8.1　単純ステートメントと複合ステートメント

複合ステートメントは、一対の中かっこに囲まれたブロックです。ブロックにより、宣言とステートメントのセットが 1 つの構文単位にまとめられます。自動記憶期間を持つオブジェクトの初期化子、およびブロックスコープを持つ通常の識別子の可変長配列宣言子は、ステートメントの場合と同様、宣言が実行順序に達するたびに、各宣言内の宣言子が現れる順番で評価され、値はオブジェクトに格納されます (初期化子を持たないオブジェクトの未定義の値の格納を含む)。次に例を示します。

```
int i;  // simple statement
{       // compound statement
  int i;
  i =90;
  ...
}
```

8.2　式ステートメントと空ステートメント

式ステートメントには式のみが含まれます。式は副作用のため void 式[1]として評価されます。セミコロンのみから成る空ステートメントでは、実行される操作はありません。

関数呼び出しが副作用のためにのみ式ステートメントとして評価される場合は、以下に示すように、キャストによって式を void 式に変換することで、明示的に値を破棄することができます。

```
int p(int);
/* ... */
(void)p(0);
```

[1]訳注：値が破棄される式。

以下のプログラムに示すように、空のステートメントを使用して、空のループ本体を繰り返しステートメントに指定できます。

```
char *s;
/* ... */
while(*s++ != '\0')
        ;
```

また、空ステートメントを使用して、複合ステートメントの閉じる } の直前のラベルを指定することもできます。

```
while(loop1) {
  /* ... */
  do {
    /* ... */
    if(want_out)
       goto end_loop1;
    /* ... */
  } while (loop2);
  /* ... */
  end_loop1: ;
}
```

8.3 選択ステートメント

選択ステートメントは、制御式の値に応じて、ステートメントのセットからステートメントを選択します。
選択ステートメントは、自身を囲んでいるブロックのスコープの厳密なサブセットであるスコープを持つブロックです。また、関連するそれぞれのサブステートメントは、選択ステートメントのスコープの厳密なサブセットのスコープを持つブロックです。

8.3.1 If ステートメント

if ステートメントの構文は以下のとおりです。

```
if(expression)
  statement
```

if ステートメントの制御式にはスカラ型を使用しなければなりません。ステートメントは、式の比較が 0 と等しくない場合に実行されます。
Ch では、C99 で追加され、ブール型 **bool** が定義されているヘッダーファイル **stdbool.h** をサポートします。マクロ **true** および **false** がブール値を処理するために定義されています。

マクロ **true** は 1、マクロ **false** は 0 として定義されます。以下のコード例は、条件式で **bool** 型を使用する方法を示します。

```
#include <stdbool.h>
bool i = true;
/* ... */
if (i) {
  i = false;
}
```

8.3.2 If-Else ステートメント

if-else ステートメントの構文は以下のとおりです。

```
if(expression)
  statement1
else
  statement2
```

if ステートメントの制御式にはスカラ型を使用しなければなりません。式の比較が 0 と等しくない場合は、最初のサブステートメントが実行されます。式の比較が 0 と等しい場合は、2 番目のサブステートメントが実行されます。最初のサブステートメントにラベルを通じて達した場合、2 番目のサブステートメントは実行されません。

8.3.3 Else-If ステートメント

else-if ステートメントの構文は以下のとおりです。

```
if(expression1)
  statement1
else if (expression2)
  statement2
else if (expression3)
  statement3
else
  statement4
```

意味的には、else-if ステートメントの構文は、前の if-else ステートメントの拡張になります。**else** は、構文で許可された最も近い場所にある前の **if** に関連付けられます。上のステートメントは、次のように書き換えられます。

```
if(expression1)
  statement1
else
```

```
    if (expression2)
       statement2
    else
       if (expression3)
          statement3
       else
          statement4
```

8.3.4 Switch ステートメント

switch ステートメントの構文は以下のとおりです。

```
switch(expression) {
  case const-expr1:
     statement1
     break;
  case const-expr2:
     statement2
     break;
  default:
     statement
     break;
}
```

switch ステートメントの制御式には、整数または文字列型を使用しなければなりません。それぞれの **case** ラベルの式は整数の定数式または文字列でなければならず、 一つの switch ステートメントの中の 2 つの **case** 定数式が変換後の同一の値を持つことはできません。**switch** ステートメントには **default** ラベルを 1 つまで使用できます。**switch** ステートメントでは、制御式の値と、**switch** 本体の **default** ラベルの有無および **case** ラベルの値に応じて、*switch* 本体であるステートメント内に、またはステートメントを通り越して制御がジャンプします。**case** または **default** ラベルには、それらを囲んでいる最も近接した **switch** ステートメント内部からのみアクセスできます。switch ステートメントの **case** 値の数に制限はありません。

制御式で、整数の上位変換が実行されます。各 **case** ラベルの定数式は、制御式の上位変換型に変換されます。変換された値が上位変換された制御式の値と一致する場合は、一致した **case** ラベルに続くステートメントに制御がジャンプします。一致しない場合は、**default** ラベルがあれば、このラベルのステートメントに制御がジャンプします。変換された **case** 定数式が一致せず、**default** ラベルもない場合、実行される switch 本体の部分はありません。

以下のようなコード例

```
switch (expr) {
   int i = 10;
   f(i);
case 0:
```

```
        i = 20;
        /* falls through into default code */
    default:
        printf("%d\n", i);
    }
```

において、自動記憶期間を持つ識別子 **i** のオブジェクトがブロック内にありますが、これが初期化されることはありません。このため、制御式がゼロ以外の値である場合、**printf** 関数への呼び出しが不定の値にアクセスします。同様に、関数 **f** への呼び出しに達することができません。

switch ステートメントの制御式では、以下の例に示すように、整数の代わりに文字列を使用できます。したがって、そのような switch ステートメントでは、すべての case 定数式も文字列でなければなりません。

```
string_t str;
str = `hostname`;  // get host name from command 'hostname'
char *s="host2";
switch (str) {      // or switch (s)
  case "host1":
    printf("s = host1\n");
    break;
  case "host2":
    printf("s = host2\n");
    break;
  default:
    break;
}
```

8.4 繰り返しステートメント

繰り返しステートメントでは、制御式の比較が 0 に等しくなるまで、*loop* 本体と呼ばれるステートメントが繰り返し実行されます。

繰り返しステートメントのループ本体はブロックです。

8.4.1 While ループ

`while` ステートメントの構文は以下のとおりです。

```
while(expression)
  statement
```

制御式の評価は、ループ本体の実行が開始するたびに行われます。
制御式の比較が 0 と等しくなるまで、ループ本体が繰り返し実行されます。
たとえば、以下のコード例

```
int i =0;
while(i<5) {
  printf("%d ", i);
  i++;
}
```

からは、次の出力が生成されます。

```
0 1 2 3 4
```

8.4.2 Do-While ループ

do-while ステートメントの構文は以下のとおりです。

```
do
  statement
while(expression);
```

制御式の評価は、ループ本体の実行が終了するたびに行われます。制御式の比較が 0 と等しくなるまで、ループ本体が繰り返し実行されます。

たとえば、以下のコード例

```
int i =0;
do {
  printf("i = %d\n", i);
  i++;
} while(i<5);
```

からは、次の出力が生成されます。

```
0 1 2 3 4
```

以下のコード例

```
int i = 10;
do {
  printf("i = %d\n", i++);
} while(i<5);
```

からは、次の出力が生成されます。

```
10
```

この例に示すように、ループ本体は制御式が評価される前に実行されます。以下の while ループのコード例では、while ステートメントの制御式が最初に 0 に評価されるため、出力はありません。

```
int i =10;
while(i<5)
  printf("%d ", i++);
```

8.4.3 For ループ

for ステートメントの構文は以下のとおりです。

```
for(expression1; expression2; expression3)
  statement
```

式 *expression1* は、制御式の最初の評価の前に void 式として評価されます。式 *expression2* は、ループ本体の各実行の前に評価される制御式です。式 *expression3* は、ループ本体の各実行の後に void 式として評価されます。*expression1* と *expression3* は両方とも省略できます。省略された *expression2* はゼロ以外の定数に置き換えられます。

for ループは、以下の while ループと意味的に同じです。

```
expression1;
while(expression2) {
  statement
  expression3;
}
```

たとえば、以下のコード例

```
int i;
for(i=0; i<5; i++)
  printf("%d \n", i);
```

から生成される出力は次のようになります。

```
0 1 2 3 4
```

この出力は、以下の while ループと同じです。

```
int i =0;
while(i<5) {
  printf("%d ", i);
  i++;
}
```

for ループステートメントでは、以下に示すように、より複雑な式を使用できます。

```
int i, j=10;
for(i=0, j=10; i<10&&j>0; i++, j--) {
  printf("i=%d\n", i);
  printf("j=%d\n", j);
}
```

8.4.4 Foreach ループ

foreach ステートメントの構文は以下のとおりです。

```
foreach(token; expr1; expr2; expr3)
  statement
foreach(token; expr1; expr2)
  statement
foreach(token; expr1)
  statement
```

foreach ループは、文字列型または char へのポインタの条件に基づく繰り返しを処理するために使用されます。式 *expr1*、*expr2*、および *expr3* は、文字列型または char へのポインタを保持します。また、識別子 *token* も文字列型または char へのポインタを保持します。各繰り返しでは、変数 `token` は元の式 *expr1* から区切り記号 *expr3* で分離されたトークンを受け取ります。ループ本体は、`token` が NULL ポインタまたは `expr2` と等しくなるまで繰り返し実行されます。この処理は、制御式 `(token==NULL || expr2!=NULL && !strcmp(token,expr2))` を 0 と比較することで達成されます。省略された *expr2* と *expr3* は、それぞれ NULL と";"に置き換えられます。

以下のコード例

```
char *token, *str="ab:12 cd ef", *cond="cd", *delimit=" :";
foreach(token; str; cond; delimit)
  printf("token= %s\n", token);
printf("after foreach token = %s\n", token);
printf("after foreach cond = %s\n", cond);
printf("after foreach delimi= %s\n", delimit);
```

からは、次の出力が生成されます。

```
token= ab
token= 12
after foreach token = cd
after foreach cond = cd
after foreach delimi=  :
```

この例では、プログラムの変数 `delimit` の値が示すように、トークンの区切り記号はスペースとコロンの文字です。以下のコードは、現在のディレクトリ内に 3 つのディレクトリ *dir1*、*dir2*、および *dir3* を作成します。

```
string_t token, str="dir1 dir2 dir3";
foreach(token; str) {
  mkdir $token
}
```

8.5 分岐ステートメント

分岐ステートメントは別の場所に無条件にジャンプします。ある関数から別の関数にジャンプするには、ヘッダーファイル setjmp.h 内の関数 **setjmp()** および **longjmp()** を使用する必要があります。

8.5.1 Break ステートメント

break ステートメントでは、for、while、do-while、foreach ループおよび switch をすぐに終了できます。break によって、最も内側を囲むループまたはスイッチから直ちに抜け出します。**break** ステートメントは switch 本体またはループ本体だけに使用されます。たとえば、以下のコード例

```
int i;
for(i=0; i<5; i++) {
   if(i == 3) {
       break;
   }
   printf("%d \n", i);
}
```

からは、次の出力が生成されます。

```
0 1 2
```

8.5.2 Continue ステートメント

continue ステートメントでは、囲んでいる for、while、do-while、foreach ループの次の繰り返しが開始されます。continue ステートメントはループ本体内で、またはループ本体としてのみ使用できます。各ステートメントは以下のようになります。

```
while (/* ... */) {          do {
   /* ... */                    /* ... */
   continue;                    continue;
   /* ... */                    /* ... */
   contin: ;                    contin: ;
}                            } while (/* ... */);

for(/* ... */) {             foreach (/* ... */)
   /* ... */                    /* ... */
   continue;                    continue;
   /* ... */                    /* ... */
   contin: ;                    contin: ;
}                            }
```

この **continue** ステートメントは、ステートメント内部で解釈されるような、囲まれた繰り返しステートメント内に置かれなければ、**goto contin;** と同等になります。
たとえば、以下のコード例

```
int i;
for(i=0; i<5; i++) {
   if(i == 3) {
      continue;
   }
   printf("%d \n", i);
}
```

からは、次の出力が生成されます。

```
0 1 2 4
```

8.5.3 Return ステートメント

return ステートメントは現在の関数の実行を終了し、制御を呼び出し元に返します。関数には、任意の数の **return** ステートメントを使用できます。式を持つ **return** ステートメントを実行した場合は、関数呼び出し式の値として式の値が呼び出し元に返されます。式の型が、その式を含む関数の戻り値の型と異なる場合は、関数の戻り値の型を持つオブジェクトへの代入が行われたかのように値が変換されます。式を持つ **return** ステートメントは、戻り値の型が **void** の関数に使用してはなりません。式を持たない **return** ステートメントは、戻り値の型が **void** の関数にのみ使用できます。

8.5.4 Goto ステートメント

goto ステートメントでは、囲んでいる関数内にある名前ラベルのプレフィックスが付けられたステートメントに無条件でジャンプします。goto ステートメントは関数内の前方または後方に制御を移すことができます。次に例を示します。

```
for (/* ... */)
  for(/* ... */) {
    /* ... */
    if(emergency)
       goto hospital;
}
/* ... */
hospital:
   emergenceaction();

 void funt1(int j)
 {
```

```
    int funt2(int j)
    {
      if(j>10)
        goto label1;
      j = 10;
    }
    funct2(j)
    label1: exit(1);
  }
```

入れ子にされた関数では、制御の流れは内側の関数から、ラベルが定義されている、囲んでいる外側の関数までジャンプできます。しかし、囲んでいる外側の関数から内側の関数にはジャンプできません。次に例を示します。

```
int task() {
  int task1() {
    /* ... */
    if(student)
      goto school;
    /* ... */
  }
  int task2() {
    /* ... */
    if(tolean)
      goto school;
    /* ... */
  }
  school:
    study();
}

 void funt1(int j)
 {
   if(j>10)
       goto label1; // Error: going INTO scope of inner function
   j = 10;
   int funt2(int j)
   {
     label1:
     /* ... */
   }
   funct2(j)
 }
```

goto ステートメントは、不定の変更される型を持つ識別子のスコープ外部からその識別子のスコープ内部にはジャンプできません。ただし、スコープ内のジャンプは許可されます。

```
goto lab3;            // Error: going INTO scope of VLA
{
      double a[n];
      a[j] = 4.4;
lab3:
      a[j] = 3.3;
      goto lab4;      // OK, going WITHIN scope of VLA
      a[j] = 5.5;
lab4:
      a[j] = 6.6;
}
goto lab4;            // Error: going INTO scope of VLA
```

8.6 ラベル付きステートメント

ラベル付きステートメントの構文は以下のとおりです。

```
labeled-statement:
        identifier : statement
        case constant-integral expr : statement
        case string-expr: statement
        default : statement
```

case または **default** ラベルは、**switch** ステートメントにのみ使用可能です。ラベル名は関数内で一意でなければなりません。ラベル名として識別子を宣言するプレフィックスは、任意のステートメントの前に置くことができます。ラベルによって制御の流れが変更されることはありません。ラベル名は、関数スコープを持ちます。

第9章　ポインタ

ポインタは、別の変数のアドレスまたは動的に割り当てられたメモリのアドレスを含む変数として定義されます。整数へのポインタ型のポインタ変数がある場合は、整数変数、または整数型の配列要素を指していると考えられます。C および Ch のプログラミングではポインタが不可欠です。また、ハードウェアとの対話でも役立ちます。

Ch のポインタには、C との互換性があります。Ch では配列、構造、関数、クラス、および単純なデータ型に対し明示的にポインタを使用します。ポインタの 2 つの基本的な演算子があります。それらは、間接演算子 '*' とアドレス演算子 '&' です。以下のコンテキストで使用されます。

1. 名前の前に演算子 '*' を追加して、ポインタを宣言します。
2. 名前の前に演算子 '&' を追加して、変数のアドレスを取得します。
3. ポインタ名の前に演算子"*"を追加して、変数の値を取得します。

ポインタ型の変数は、他のデータ型の変数と同じように宣言できます。次に例を示します。

```
int *p, i;
```

これは、整数へのポインタ `p` と整数 `i` を宣言します。式`*p` は整数型です。任意の型の変数のポインタを使用できます。ポインタを特定の型に関連付ける必要があることに注意してください。例外が 1 つあります。"void へのポインタ"は、任意の型のポインタを格納するのに使用できますが、それ自身を逆参照することはできません。

単項演算子 '&' では "変数のアドレス"が取得されます。式`&i` は、変数 `i` のアドレスを意味します。間接演算子"*"では、"ポインタが指しているオブジェクトの内容"が取得されます。式`*p` は、変数 `p` が指している場所に格納された値を表します。これは乗算演算子とは異なり、また、ポインタ型変数の宣言での使用とも異なります。このため、次のようなプログラミングステートメントがあるとします。

```
p = &i;
```

これは、ポインタ `p` に `i` のアドレスを設定します。これを実行すると、`*p == [i` が格納する値]、となります。

9.1　ポインタ演算

上で示したように、ポインタはスカラ型の単純な変数を指す必要はありません。また、ポインタは配列要素を指すことができます。たとえば、次のように書くことができます。

```
int *p;
int a[10];
p = &a[3];
```

これにより、p は配列 a の 4 番目の要素を指すようになります。既定では配列インデックスが 1 ではなく 0 から開始することに注意してください。次のような状況になります。

```
 a[0] a[1] a[2] a[3] a[4]  ...  a[9]
                 |
                 p
```

ポインタ p は、前のセクションのポインタと同様に使用できます。式*p は p が指す内容を取得するため、この場合は a[3] の値になります。

配列要素または動的に割り当てられたメモリを指すポインタがある場合は、ポインタ演算を実行できます。*p* が a[3] へのポインタであるとすると、*p* に 1 を加算できます。

```
p + 1
```

Ch および C では、1 を加算するとポインタが次のセルに移動します。以下のコードは、この新しいポインタを別のポインタ変数 p2 に割り当てます。

```
int *p2;
p2 = p + 1;
```

こうすると、ポインタと配列の関係は次のようになります。

```
a[0] a[1] a[2] a[3] a[4] ... a[9]
                |    |
                p    p2
```

次のプログラミングステートメントがあります。

```
*p2 = 4;
```

これにより、a[4] は 4 に設定されます。以下に示すように、新しく計算したポインタ値をすぐに使用できます。

```
*(p + 1) = 5;
```

この例では、再び a[4] を変更して、5 に設定しました。単項演算子*の優先度は加算演算子より高いため、かっこが必要です。かっこを使わずに*p + 1 と書いた場合は、p が指す値を取得し、その値に 1 を加算することになります。

1 を加算する以外にも、ポインタに対し任意の数を加算または減算できます。引き続き、p が a[3] を指しているとします。

```
*(p + 5) = 7;
```

これで、a[8] は 7 に設定されます。

```
*(p - 2) = 4;
```

また、a[1] は 4 に設定されます。

インクリメント演算子 '++' とデクリメント演算子 '−−' を使用すると、一度に 2 つの処理が簡単に行えます。*p++のような式では、p が指す内容にアクセスし、同時に p が増分されるため、ポインタは次の要素を指すようになります。事前インクリメント形式の*++p は、p を増分してから、指している内容にアクセスします。(*p)++は、p が指している内容を増分することに注意してください。

文字へのポインタは一般的に使用されます。Ch では、次のように文字列を定義します。

```
char * str1;
string_t str2;
```

以下の例は、ポインタを使用して文字列を処理する方法を示します。

```
char dest[100], src[100];
char *dp = dest, *sp = src;

strcpy(src, "abcd");
/* copy src to dest */
while(*sp != '\0')
    *dp++ = *sp++;
*dp = '\0';
```

上の例では、char へのポインタを使用して、配列 src の文字列をコピーしています。

ポインタ演算を実行する場合は、必ず有効な範囲内で実行するようにしてください。たとえば、10 個の要素を持つ配列では有効な添字の範囲が既定で 0 から 9 であるため、配列 a の要素が 10 個である場合、a[10] または a[-1] にアクセスすることはできません。

明示的なポインタを使用する以外にも、配列名自体を使用して配列の要素にアクセスできます。これは、C および Ch では配列名が配列内の最初の要素へのポインタであるためです。したがって、次のようになります。

```
p = a;
```

このステートメントは、次のステートメントと同等です。

```
p = &a[0];
```

これらの 2 つのステートメントはどちらも、ポインタ p が配列 a の最初の要素を指しています。さらに、以下のようにして、配列 a の 3 番目の要素にアクセスできます。

```
int aa2 = *(a+2);  // obtain the value of the third element
*(a+2) = 5;        // assign 5 to the third element of a
```

9.2 メモリの動的な割り当て

固定サイズの配列を使用する際の問題は、特別なケースを処理するには小さすぎるか、または大きすぎればリソースが無駄になることです。可変長配列を使用せずにこの問題を解決するには、標準関数 **malloc()**、**calloc()**、 または **realloc()** と演算子 **new** を使用して、 メモリを動的に割り当てます。

calloc()、**malloc()**、および **realloc()** の各関数の連続呼び出しで割り当てられる記憶域の順序と隣接は不明です。後続の割り当ての適切な場所が確保されるとポインタが返され、(領域が明示的に解放されるか再割り当てされるまでの間)、これを任意の型のオブジェクトへのポインタに代入し、割り当てられた領域を占めるそれらのオブジェクトまたはオブジェクト配列にアクセスできるようになります。こうした割り当てのたびに、他のオブジェクトから分離されたオブジェクトへのポインタが生成されます。返されたポインタは、割り当て領域の開始位置 (最低バイトアドレス) を指します。領域を割り当てることができない場合は、null ポインタが返されます。要求された領域のサイズがゼロである場合の動作は、プラットフォームに依存します。null ポインタが返されるか、または、返されたポインタを使用してオブジェクトにアクセスできないこと以外は、サイズがゼロ以外のなんらかの値を持つ場合と同じように動作します。解放された領域を参照するポインタの値は不定です。

以下の例では、少量のメモリを割り当て、関数 **strcpy()** で文字列をコピーしています。

```
char *str = "abcd", *copy;
...
/* +1 for NULL terminator */
copy = (char *)malloc(strlen(str) + 1);
strcpy(copy, str);
```

すべての文字列の末尾に付けられる ‘`\0`’ 文字は **strlen()** に含まれないことを思い出してください。文字列 `str` のバイト数は `strlen(str)+1` ではなく、`strlen(str)` です。

Ch には、変数または型のバイト単位のサイズを計算する **sizeof** 演算子があります。これは、サイズが不明な変数のメモリを割り当てるのに役立ちます。整数 100 個の領域を割り当てるには、次のようにします。

```
int *p = (int *)malloc(100 * sizeof(int));
```

無限のメモリを使用できるコンピュータなどどこにもありません。malloc(1000000000) を呼び出すか、malloc(10) を 1 億回呼び出せば、システムはおそらくメモリを使い果たします。関数 **malloc()** は、要求されたメモリを割り当てることができない場合、**NULL** ポインタを返します。したがって、**malloc()** の呼び出しでは常に戻り値をチェックしてから使用することが重要です。関数 **malloc()** の呼び出しとエラーチェックは、以下のように行います。

```
int *p = (int *)malloc(100 * sizeof(int));
if(p == NULL)
{
    printf("out of memory\n");
    exit(1);
}
```

関数 **malloc()** で **NULL** が返された場合は、エラーメッセージの出力後、呼び出し元に戻るか、または完全にプログラムを終了する必要があります。p が指すメモリを使用しているコードを続行することはできません。第¿18章で説明していますが、動的なメモリ割当てに関する適切なアプリケーションの例は、リンクリストを作成することです。

自動持続変数とは異なり、動的に割り当てられたメモリは関数が戻るときに自動的に消滅しません。関数 **malloc()** を使用してメモリを割り当てるタイミングと量を厳密に制御できるのと同様、割り当てを解除するタイミングも厳密に制御できます。事実、多数のプログラムは一時的にメモリを使用します。メモリをいくらか割り当て、しばらくはそれを使用しますが、ある時点でその特定の断片が必要なくなります。メモリは無尽蔵ではないため、使用しなくなったメモリの割り当てを解除する (つまり、解放する) ことは良い考えです。

動的に割り当てたメモリの解除には、関数 **free()** を使用します。第19章で説明しますが、演算子 **new** を使用して動的に割り当てられたメモリは、演算子 **delete** を使用して割り当てを解除できます。p には、前に関数 **malloc()** で返されたポインタが格納されているとします。この場合、次の関数を呼び出します。

```
free(p);
```

これで、動的に割り当てられたメモリが解放されます。`free(p)` を呼び出した後も、C では、p が引き続き同じメモリを指していることがしばしばあります。ただし、Ch では、割り当てが解除された後の p は NULL に設定されます。p が NULL 以外の値であるかどうかを再使用前にチェックしている限り、ポインタ p を通じてメモリの使用を間違えることはありません。

Ch ではポインタを返す関数がサポートされています。これは関数内でのメモリの割り当てに役立ちます。以下は、整数へのポインタを返す関数の簡単な例です。

```
int *fn1() {
    int *p = (int *)malloc(sizeof(int));
    .....

    *p = 5;
    ...

    return p;
}
```

関数 `fn1()` で関数 **malloc()** により動的に割り当てられたメモリは、呼び出し元関数で解放できます。

以下のコードは無効ですので、注意してください。

```
int *fn2() {
    int k;
    ...

    k = 5;
    ...
```

```
    /* return address of k*/
    return &k;
}
```

関数 fn2() はローカル変数 k のアドレスを返そうとします。関数 fn2() が戻るとき、変数 k は自動的にメモリの割り当てが解除されます。

9.3 ポインタ配列

C と同様、Ch では、以下のようにポインタ自身が変数であるため、ポインタ配列がサポートされています。

```
int (*p1)[3], a1[2][3], a2[3][3];
p1 = a1;        // p1[i][j]<=>a1[i][j]
...
p1 = a2;        // p1[i][j]<=>a2[i][j]
int *p2[3];     // declares an array of 3 pointers to ints.
```

ポインタ配列が非常に役立つ場合があります。以下のコード例について考えます。

```
char m1[7][10] = {"Sunday", "Monday", "Tuesday", "Wednesday",
                  "Thursday", "Friday", "Saturday"};
char *m2[7] =    {"Sunday", "Monday", "Tuesday", "Wednesday",
                  "Thursday", "Friday", "Saturday"};
```

変数 m1 は char の二次元配列ですが、m2 は char へのポインタの配列です。図9.1と図9.2に、m1 と m2 のメモリレイアウトをそれぞれ示します。

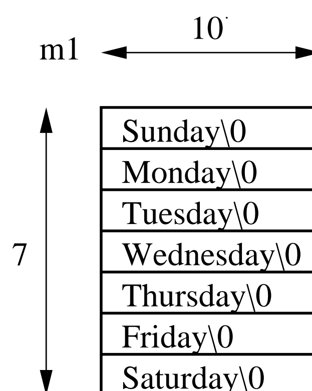


図 9.1: 二次元配列

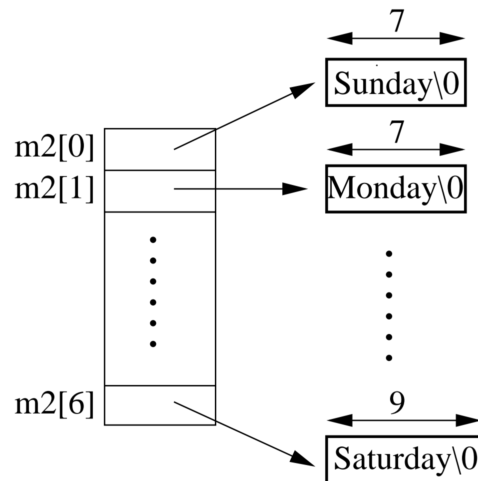


図 9.2: ポインタの配列

m2 を使用する利点は、各ポインタが 10 バイトの固定長配列ではなく、長さが異なる配列を指すことができる点です。以下のコードで説明します。

```
/* three text lines */
char *p[3] = {"ABC", "HIJKL", "EF"};
char *tmp;
...
tmp  = p[1];
p[1] = p[2];
p[2] = tmp;
```

この例は、ポインタ配列を使用することで、複雑なメモリの管理と行移動のオーバーヘッドがどのように削減されるかを示します。図9.3に、p[0]、p[1]、p[2] の各ポインタが指し示す、長さの異なる元の文字列を示します。図9.4に示すように、これらの文字列の文字を移動またはコピーせずに、ポインタの値を交換することで、p[1] と p[2] のポインタが指す内容が交換されます。

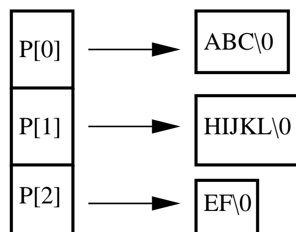


図 9.3: テキストの交換前

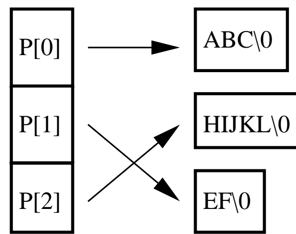


図 9.4: テキストの交換後

9.4 ポインタへのポインタ

異なる型を持つポインタ自身が変数であるため、Ch では、任意の型のポインタへのポインタを処理できます。

以下のコードについて考えます。

```
char ch;            // a character
char *p = &ch;      // a pointer to ch
char **pp = &p;     // a pointer to p
```

図に表すと図9.5のようになります。**pp は*p のメモリアドレスを参照し、*p は変数 ch のメモリアドレスを参照します。

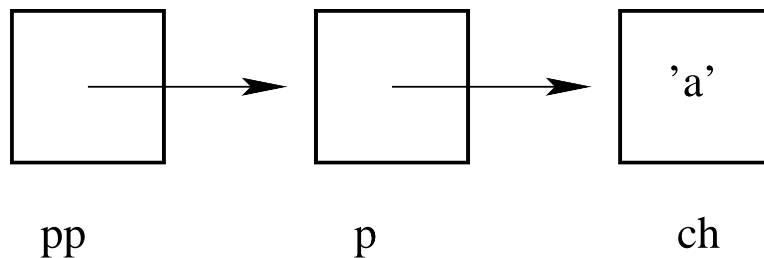


図 9.5: ポインタへのポインタ

Ch では **char ***を使用して NULL で終わる文字列を参照するため、一般的で便利な 1 つの方法として、char へのポインタへのポインタを宣言します。たとえば、次のコードがあるとします。

```
char *p = "ab"; // a string
char **pp = &p; // a pointer to p
```

図9.6で示すように、ポインタ p を宣言し、char へのポインタへのポインタ pp を宣言します。

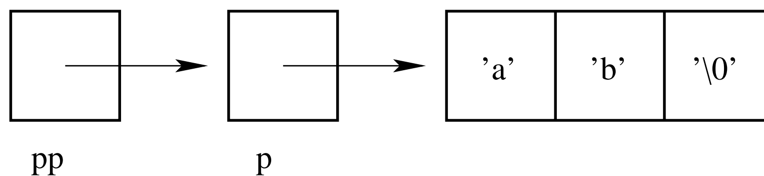


図 9.6: 文字列へのポインタ

さらに、Ch では、以下のコマンドが示すように、二重ポインタが指す複数の文字列がサポートされています。

```
> char **pp;
> pp = (char**)malloc(3*sizeof(char*));
4006c8d0
> pp[0] = "ab";
ab
> pp[1] = "py";
py
> pp[2] = NULL;
00000000
```

図9.7に、上のコードのメモリレイアウトを示します。

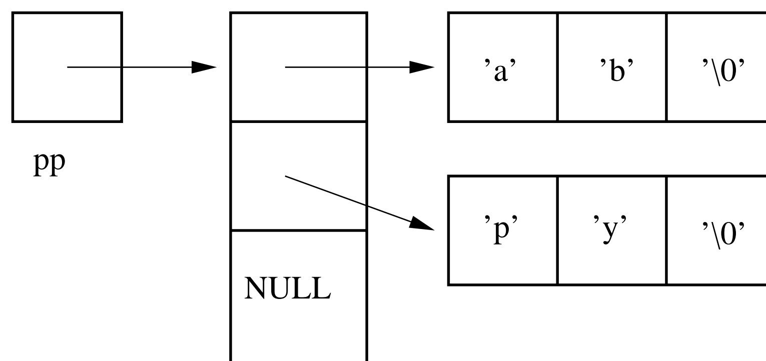


図 9.7: 複数の文字列へのポインタ

`pp[0]` および `pp[1]` で、個々の文字列を参照できます。これは、宣言 `char *pp[]` と意味的に同じです。二重ポインタは関数 **main()** のコマンドライン引数の処理に役立ちます。

ポインタへのポインタは、メモリの動的割当てにも役立ちます。以下のプログラムがあるとします。

```
void fn3(int **p) {
    *p = (int *)malloc(sizeof(int));
    **p = 5;
}
int main() {
    int *p;
    ....

    fn3(&p);
    ....
}
```

呼び出し元関数 **main()** のポインタ `p` のメモリは、関数 `fn3()` で割り当てられます。

以下に示すような、ポインタと二重ポインタを使用したプログラミングステートメントの対話的な実行でポインタが動作するしくみを理解するには、対話型 Ch シェルが特に役立ちます。

```
> int i, *p
> p = &i    // assign address of i to p
1c4228
> *p = 90
90
> printf("i = %d\n", i);
i = 90
> int **p2
> p2 = &p
1c3c38
> printf("**p2 = %d\n", **p2)
**p2 = 90
> i**p   // i * (*p)
8100
>
```

第10章　関数

Ch プログラムは、通常、数多くのプログラミングステートメントを直後に含む関数セットで作成されています。関数を使用して、規模の大きい処理タスクを小さな処理タスクに分割できます。これにより、開発者は、一からではなく、他のプログラマが作成したものを基にしてアプリケーションプログラムを作成することができます。プログラミング言語にとって、関数の性能とユーザーが使い易いインタフェースは重要な要素です。Ch では、関数へのすべての関数呼び出しがプロトタイプによって統制されること、同じ関数のすべてのプロトタイプに互換性があること、数多くのファイルに分割されたプログラムの場合でも、すべてのプロトタイプが関数定義を満たすことが保証されます。

メイン関数 `main()` など、C の関数はすべて同じレベルであるため、他の関数内で関数を定義できません。言い換えると、C には内部プロシージャがないということです。

Ch は、入れ子にされた関数で C を拡張したものです。Ch の関数は再帰的にできるだけでなく、入れ子にすることもできます。これは、関数が関数自身を呼び出せるだけでなく、関数内で他の関数を定義できることを意味します。関数を入れ子にすることで、ある関数モジュールの詳細を、入れ子にされた関数を関知する必要のない他のモジュールから隠すことができます。

他のモジュールから切り離して、各モジュールに取り組むことができます。ソフトウェアのメンテナンスは、プログラムのコストの大きな部分を占めます。ほとんどの場合、プログラムのメンテナンスを担当するのは、当初の設計に関わらなかった人々です。入れ子にされた関数でプログラムをモジュール化すると、プログラム全体が見通せるようになり、他のプログラマが書いたモジュールを変更する場合の作業負荷も小さくなります。関数の入れ子は、必要な情報だけを表示することができ、モジュラー型プログラミングでは非常に有用です。

入れ子にされた関数を C に追加すること自体はプログラミング言語にとってそれほど大きな拡張ではありませんが、言語規格に何か新しい機能を追加する場合は、言語全体に及ぼす影響の可能性について注意深く検討する必要があります。新しい機能を追加して C を拡張することは、とりわけ、いわゆる「C の精神」に適うという意味でも、無理のない形で行われるべきです。したがって、現行の既存のコードを完全に壊すようなことがあってはなりません。

入れ子にされた関数を使用すると、他の関数内でローカル関数を定義できます。C の精神に従い、Ch の関数は入れ子にできるだけでなく、再帰的呼び出しも可能です。言い換えると、関数は直接的または間接的のいずれかで関数自身を呼び出しできるということです。関数ファイルを書く上で、これは特に重要です。関数ファイルの中で定義された関数は、Ch 言語の環境ではシステムに組み込みの関数であるかのように扱われます。本章では、最初に、標準 C に準拠した関数の処理方法を説明し、次に、C の精神で Ch に現在実装されている入れ子にされた関数の新しい特徴をプログラム言語の観点から説明します。

10.1 値呼び出しと参照呼び出し

引数を関数に渡す場合、通常は**値呼び出し**または**参照呼出し**のどちらかの方法がとられます。値呼び出しでは、実引数の値は、呼び出された関数のローカルの仮引数にコピーされます。仮引数を lvalue (割り当てステートメントの左辺で指定可能なオブジェクト) として使用すると、引数のローカルコピーだけが変更されます。一方、参照呼び出しの場合は、引数のアドレスが関数の仮引数にコピーされます。呼び出された関数内で、このアドレスは、呼び出し元の関数で使用された実際の引数にアクセスするために使用されます。これは、仮引数を lvalue として使用すると、関数呼び出しに使用された変数が引数の影響を受けることを意味します。FORTRAN では参照呼び出しを使用しますが、C では通常、値呼び出しを使用します。呼び出された関数によって呼び出し元の関数の実引数が変更されることを C で行うには、引数のアドレスを明示的に渡す必要があります。ただし、C++と Ch では、次の章で説明する参照型を使用して引数を参照渡しすることができます。

10.2 関数の定義

関数は次の形式で定義できます。

```
return_type function_name(argument declaration)
{
  statements
}
```

上記の関数定義の一部は、指定を省くことができます。`return_type` には有効な型指定子のいずれかを指定できます。Ch の関数定義では、int を返す関数の場合でも、定義の先頭に型指定子を指定する必要があります。

K&R C として知られる古い関数定義も Ch でサポートされます。今では使われない手法ですが、K&R の C の記述方法では、関数定義内のパラメータ識別子は宣言リストで切り離されます。

宣言ステートメントと実行可能なステートメントとを混在させることができます。たとえば、次のコード例

```
int funct(int i)
{
  i = 3;
  int j;
  return i;
}
```

では、変数 `j` は実行ステートメント `i = 3` の後で宣言されています。関数の引数に関するパラメータ変数のレキシカルレベルは、関数内で定義されたローカル変数のレキシカルレベルより低くなります。同一の識別子を関数のパラメータ変数とローカル変数の両方に使用すると、パラメータ変数をローカル変数として宣言する宣言ステートメントの前では、変数は関数の引数として扱われます。宣言ステートメントの後では、変数は関数内でローカル変数になります。したがって、C の場合と異な

り、Ch では、次の例に示すように、[1]関数の引数とローカル変数の両方に同じ識別子を使用することができます。

```
int funct(int i, j)
{
  printf("i = %d \n", i);// use i as the argument parameter
  int j=1, i=1;          // i and j are initialized to 1
  j = i +8 +j;           // use i and j as local variables
  return j;              // return the local variable j with 10
}
```

Ch では、変数は、宣言されると必ずゼロに初期化されます。上記の関数 funct() では、引数パラメータの値を含む識別子 i は、出力関数 printf() によって出力されます。その後、識別子 i は宣言ステートメント int j, i の後でローカル変数になります。

名前 j は、関数の引数とローカル変数の両方に使用されています。変数 j は、ローカル変数として宣言された後、関数内で呼び出されます。呼び出し元の関数から渡された値は、関数内では使用されません。つまり、ローカル変数は関数の引数パラメータを表示しません。

上記の例で、関数 funct() は、C では int funct(int i, int j) と定義する必要があることに注意してください。[2]以降に出現する引数の型宣言子は、前出のものと同じ型であれば、Ch では省略できます。ただし、引数リスト内の識別子が typedef 名の場合は、次の例に示すように、型宣言子を省略することはできません。

```
typedef int INT;
int funct(int i, int INT) // int funct(int i, INT) is an error
{
  INT =90;
  /* ... */
}
```

次のような return ステートメントを使用すると、呼び出された関数から呼び出し元の関数に値を返すことができます。

```
return expression;
```

return の後にはどのような式も置くことができ、式を囲む括弧も省略可能です。必要に応じて、式は関数の戻り値の型に変換されます。ただし、組み込みのデータ型変換規則に準じて、関数の戻り値の型に式を暗黙的に変換できない場合は、構文エラーとなります。次に例を示します。

```
int funct()
{
  int *p;
  return p;       // ERROR: wrong return data type
```

[1]訳注：このようなコーディングは、プログラムの解析を困難にするため、実際にはおすすめできません。
[2]訳注：C/C++との互換性の観点から、おすすめできません。

```
  return (int)p; // OK: C type conversion
  return int(p); // OK: functional type conversion
}
```

戻り値の型が void でない場合は、関数の末尾に return ステートメントが必要です。return ステートメントがないと既定のゼロ値が戻り値に使用され、システムによって警告メッセージが表示されます。次に例を示します。

```
> int funct(){}
WARNING: missing return statement for function funct() and \
default zero is used
```

つまり、右側の閉じる括弧の前にゼロの戻り値を含む return ステートメントがあるかのように関数は扱われます。ここで、ゼロの値は一般的な意味で使用しています。たとえば、int 型のゼロは 0、float 型のゼロは 0.0、ポインタ型のゼロは NULL、複素数型のゼロは complexZero です。また、戻り値の型が void でない場合は return の後に式が必要です。式がないと既定のゼロ値が戻り値に使用され、警告メッセージが表示されます。次に例を示します。

```
int funct()
{
  return;  // WARNING: missing return expression and use default zero
}
```

ただし、呼び出し元の関数は自由に戻り値を無視できます。次に例を示します。

```
int funct(int i)
{
  return i+1;       // the same as 'return (i+1);'
}
funct(5);           // ignore the return value
```

何も返さないように関数を定義するなら、戻り値のデータ型は `void` となります。値を必要とするコンテキストで、`void` の戻り値の型を含む関数を呼び出すとエラーになります。次に例を示します。

```
void funct(int i){}
int k;
k = funct(3);          // ERROR: lvalue and rvalue are not compatible
```

戻り値の型が void の場合は、return ステートメントは省略可能です。ただし、return の後に式があると、構文エラーとなります。次に例を示します。

```
void funct (int i)
{
  if(i == 3)
  {
```

```
    printf("i is equal to 3 \n");
    return i;           // ERROR: return int
  }
  else if(i > 3)
  {
    printf("i is not equal to 3 \n");
    return;             // OK
  }
  i = -1;
}
funct(2);
```

呼び出し元の関数から呼び出された関数へ渡される引数の数が、関数定義内の引数の数より少ないと、構文エラーとなります。次に例を示します。

```
int funct(int i, j){return i}
funct(8)          // ERROR: fewer parameters are passed to funct(),
```

また、実引数の数が仮引数の数より多くなる場合も、構文エラーとなります。次に例を示します。

```
int funct(int i){return i}
funct(8, 9)       // ERROR: number of arguments is 2, need 1 argument
```

Ch では、システムに組み込みの関数とユーザー定義関数は、関数の引数として使用できます。システムに組み込みの関数はポリモーフィックに扱われます。さらに、関数の引数として関数自体を使用することができます。たとえば、プログラム 10.1に示す関数呼び出し `funct2(abs(-6), funct1(funct1(2)),funct2(1,2,3))` の各引数は、それぞれ、`abs()` はシステムに組み込みの関数、`funct1()` は自身を引数として使用するユーザー定義関数、`funct2()` は関数自体を示しています。

```
#include <stdio.h>

int funct1(int j) {
    return 2*j;
}
int funct2(int j1, j2, j3) {
    return j1+j2+j3;
}
int main () {
    int i;
    i = funct2(abs(-6), funct1(funct1(2)), funct2(1,2,3));
    printf("i = %d \n", i);  // output: i = 20
}
```

プログラム 10.1: 関数の引数として関数を使用する

10.3 関数プロトタイプ

Ch の関数の戻り値および引数をチェックする型は、C より厳密で一貫しているため、プログラム内に存在するバグの検出が容易です。C では、プログラムを多数のソースファイルに分割した場合、コンパイラには次のチェックが必要とされません。

- すべての関数呼び出しがプロトタイプによって統制されているか。
- 同じ関数のプロトタイプはすべて互換性があるか。
- または、すべてのプロトタイプは関数定義を満したているか。

Ch の場合は、多数のファイルに分割されたプログラムについても、これらをすべてチェックできます。 Ch では、互換性が保たれるのであれば、呼び出し元の関数の実引数のデータ型が、呼ばれた関数の仮引数のものと同じでなくても構いません。実引数の値は、組み込みのデータ変換規則に基づき、関数のインタフェース段階で暗黙的に仮の定義のデータ型に変換されます。ただし、同じ関数に対して異なる関数プロトタイプの同じ引数のデータ型は、ファイルが異なる場合でも同じであることが必要とされます。同様に、同じ関数に対して異なる関数プロトタイプの戻り値の型も同じでなければなりません。次に例を示します。

```
int funct1(int i);         // return and argument types are int
int funct1(float f);       // ERROR: change data type of argument
int funct1(void v);        // ERROR: change data type of argument
int funct1(int &i);        // ERROR: change data type of argument
float funct1(int i);       // ERROR: change return type of function
```

```
void funct1(int i);         // ERROR: change return type of function
funct1(5);                  // Ok
funct1(5.0);                // OK, 5.0 converted to 5

int funct2(int i)
{
  int funct3(int j);        // return and argument types are int
}
int funct3(int i)
{ }
int funct3(int *p);         // ERROR: change data type of argument
```

プログラムには、所定のレキシカルレベルに 1 つの関数定義しか含めることができません。次に例を示します。

```
int funct1(int i){};
int funct1(int i){};        // ERROR: redefine funct1()
```

同じレキシカルレベルにある変数名と関数名は、同じ名前にはできません。次に例を示します。

```
int funct2;
int funct2(int i){};        // ERROR: redefine funct2
int funct3;
int funct4(int i)
{
  int funct3(int i);       // ERROR: redefine funct3
}
```

ここでは、名前 `funct2` と名前 `funct3` は、関数として定義される前に単純変数として定義されています。

パラメータ名は関数定義に示される必要があります。ただし、関数プロトタイプの引数のパラメータ名は、関数を記録したり、プログラムを読み易くしたりするのに便利ですが、[3]記述しなくてもかまいません。たとえば、次の 2 つの関数プロトタイプは同じです。

```
int funct(int);
int funct(int i);
```

複数の引数を含む関数も同じ方法でプロトタイプ化できます。次に例を示します。

```
int funct1(int, int);
int funct2(int, float);
```

パラメータ名なしで、ポインタ型の引数をプロトタイプ化することもできます。次に例を示します。

[3]訳注：文法上、記述することをおすすめします。

```
void funct(int *, float *, complex **);
void funct(int *ip, float *fp, complex **zp){ }
```

後続の引数に同じ基本データ型が含まれている場合は、最初の引数以降は関数プロトタイプの型指定子を省略できます。次に例を示します。

```
void funct1(int *ip, *jp, **ip2);
void funct1(int *, *, **);
void funct2(int, , );
void funct2(int, int, int);
void funct2(int i, j, int k);
```

関数プロトタイプでは、名前付きのパラメータと名前のないパラメータを混在させることができます。次に例を示します。

```
void funct(int, *, float f)
void funct(int i, *p, float);
```

引数を必要としない関数は、単一の型指定子 void で、または void の後にダミーパラメータ名を付けてプロトタイプ化できます。次に例を示します。

```
void funct(void);
void funct(void){ }
```

配列の引数およびポインタの配列をパラメータ名なしでプロトタイプ化することもできます。次に例を示します。

```
void funct(int *[], *[3], [][3], char []);
int *ap[10], *bp[3], a[10][20], ca[3];
funct(ap, bp, a, ca);
void funct(int *ap[], *bp[3], a[][3], char c[]){  }
```

同様に、関数引数内での配列へのポインタと形状引継ぎ配列をパラメータ名なしでプロトタイプ化することができます。次に例を示します。

```
int a[10][10];
void funct(int (*)[3], [:][:]);
void funct(int (*a)[3], b[:][:]){ }
funct(a, a);
```

同じ方法で、基本データ型への参照およびポインタへの参照を扱うことができます。次に例を示します。

```
int i, *p1, **p2;
void funct1(int &, &, &*, &**);
funct1(i, i, p1, p2);
void funct1(int &i, &j, &*p1, &**p2){ };
```

関数のプロトタイプに引数がない場合、引数型の互換性は、Ch ではチェックされません。ただし、関数の戻り値の型はチェックされます。ISO C と K&R C の両方の関数プロトタイプをプログラムに混在させることはできますが、C スタイルのプロトタイプでは、戻り値の型と引数の両方がチェックされます。最初の C 関数プロトタイプまたは関数定義によって関数の引数の数とデータ型が決まります。たとえば、

```
int funct1();                // return type is int, ignore arguments
int funct1(int i);           // argument is an int
int funct1(int i, j);        // ERROR: change number of argument
int funct1(int *p);          // ERROR: change data type of argument
int funct1();                // OK to repeat the same function prototype
int funct1(int i){ }         // function definition

complex funct2(int i){ }     // return type is complex,
                             // argument is an int
int funct2();                // ERROR: change return type of function
int funct2(int i);           // ERROR: change return type of function
complex funct2(int i);       // OK:

int funct3(int i)
{
  int funct4();
  int funct4(int i);         // argument is an int
  int funct4(int i, j);      // ERROR: change number of argument
}
int funct4(int i)
{ }

void funct5();
void funct5(void);
void funct5(void v);
void funct5();               // OK
void funct(int);             // ERROR: change data type of argument
void funct5(){ }             // function definition
```

プログラム 10.2: K&R C および ISO C の関数プロトタイプ。

プログラム 10.2に示すように、`int funct1(int i)` の最初の C 関数プロトタイプの引数は、関数 `funct1()` を定義する前に、他の C 関数プロトタイプと関数呼び出しをチェックするために使用されます。関数 `funct5()` は引数を必要とせず、プログラム 10.2に示す関数定義は `void funct5(void){ }` と同じになっています。K&R C の関数プロトタイプはエラーが起き易いとのことです。新しいコードを書く場合、K&R C の関数プロトタイプは使用しないでください。

Ch プログラムでは、関数は任意の順番に配置できます。関数が定義される前に関数が呼び出されれば、int が関数の戻り値の型と見なされます。引数の数とデータ型は、最初の C 関数プロトタイプか関数定義によって決められます。つまり、関数 `funct()` の呼び出しが定義より前に行われた場合、この関数は `int funct()` でプロトタイプ化されているかのように動作します。これについての様々なプログラミング例をプログラム 10.3に示します。

```
funct1(3);                 // by default, funct1() return int;
void funct1(int i) { }     // ERROR: change return type of function

int i;
funct3(3);           // by default, funct3() return int;
int funct3(int i) { }// WARNING: missing return statement and default zero is returned

int i;
funct4(3);           // by default, funct4() return int;
int funct4(int i)
{ return; }          // WARNING: missing return expression, use default zero

funct5(8)            // WARNING: fewer parameters are passed to funct(),
                     //          default zeros are used for missing ones
int funct5(int i, j){return i}

funct6(8);
int funct6(int i){return 3}         // OK

funct7(8)
int funct7(int i){}  // WARNING: missing return statement, use default zero

funct8(8)
void funct8(int i){} // ERROR: change return type of function

funct9(8)
int funct9(float f){return (int)f}  // OK

funct10(8)                          // ERROR: non ptr value passed to ptr
int funct10(int *p){return 3}       // OK
```

プログラム 10.3: 既定のプロトタイプを使用したサンプルプログラム

関数定義の後では、関数プロトタイプ内ではなく、関数定義内の引数の数とデータ型が、後続の関数呼び出しの実際の引数に照らしてチェックされます。関数プロトタイプは、互換性がある限り、何度でも使用することができます。しかし、関数の戻り値が int でない場合は、プログラム 10.4に示すように、関数を呼び出す前に関数プロトタイプを使用する必要があります。

```
void funct1(int i) { }
int funct(int i)
{
  void funct1(int i);  // redundant prototype OK
  void funct2(int i);  // must have prototype
  int  funct3(int i);  // can be omitted by default
  funct1(i);
  funct2(i);
  funct3(i);
}
void funct2(int i) { }
int funct3(int i) { }
```

プログラム 10.4: プロトタイプが省略可能または必須である場合の例

10.4 再帰関数

関数を再帰的に使用できます。言葉を換えると、プログラム 10.5に示すように、関数は関数自身を直接呼び出すことができます。

```
int main() {        // level 1 for main
  funct(1);
}
int funct(int j)    // level 2 for funct and level 2 for j
{
  if(j <=  3)
  {
    printf ("recursively call funct() j = %d \n", j);
    j++;
    j = funct(j);
  }
  else
    printf ("exit funct() j = %d \n", j);
  return j;
}
```

プログラム 10.5: 直接再帰関数

プログラム 10.5の出力結果は次のとおりです。
recursively call funct() j = 1
recursively call funct() j = 2
recursively call funct() j = 3
exit funct() j = 4

関数が再帰的に関数自身を呼び出す場合、関数の各呼び出しには、新しいセットのローカル変数が含まれます。再帰関数内には、関数を終了して、呼び出し元の関数にプログラムの制御フローを返す

ために、通常、if-else などの条件ステートメントが必要になります。また、プログラム 10.6に示すように、関数自身を間接的に呼び出すこともできます。

```
int main() {
  funct1(1);
}
int funct1(int i)
{
  i = funct2(i);
  printf ("exit funct1() i = %d \n", i);
  return i;
}
int funct2(int j)
{
  if(j <=  3)
  {
    printf ("recursively call funct2() j = %d \n", j);
    j++;
    j = funct1(j);
  }
  else
  {
    printf ("exit funct2() j = %d \n", j);
    j++;
  }
  return j;
}
```

プログラム 10.6: 間接再帰関数

プログラム 10.6では、関数 funct1() と関数 funct2() は関数自身を間接的に呼び出し、関数 funct2() が関数 funct2() を呼び出す一方で、関数 funct2() は関数 funct1() を呼び出します。図 10.1にプログラム 10.6の出力結果を示します。

```
recursively call funct2() j = 1
recursively call funct2() j = 2
recursively call funct2() j = 3
exit funct2() j = 4
exit funct1() i = 5
exit funct1() i = 5
exit funct1() i = 5
exit funct1() i = 5
```

図 10.1: プログラム 10.6の出力結果

10.5 入れ子にされた関数

C の精神に従い、 Ch の入れ子にされた関数による関数定義は、次の形をとります。

```
return_type function_name(argument declaration)
```

```
{
  statements
  function_definitions
}
```

または

```
return_type function_name(argument declaration)
{
  function_definitions
  statements
}
```

ここで、ステートメントにはいずれかの有効な Ch ステートメントを指定でき、ローカル関数は他のローカル関数内で定義できます。Ch では、入れ子にする関数の数に制限はありません。このセクションでは、入れ子にされた関数の特徴について言語的な観点から説明します。

10.5.1 入れ子された関数のスコープとレキシカルレベル

Ch の変数名と関数名は、それぞれのスコープに関連付けられています。変数名または関数名の**スコープ**は、変数の使用が可能なプログラムの一部です。関数の引数のスコープは関数の本体です。関数内で定義された変数のスコープは、宣言の直後から関数定義の右側の括弧までの範囲です。異なる関数に含まれる同じ名前のローカル変数は、互いに関連はありません。これは関数のパラメータにも当てはまります。ローカル関数のスコープは、ローカル関数が定義される関数です。 Ch では、関数のスコープの最上位になるのは全体のプログラムですが、関数が定義の前に呼び出されれば、その関数はプロトタイプ化される必要はありません。C では、すべての関数のスコープは全体のプログラムですが、すべてにわたって関数をプロトタイプ化する必要があります。異なるファイルに含まれる同じ関数の関数プロトタイプが一致しない場合、C コンパイラではそれを検出できませんが、Ch では検出できます。Ch のプログラムでは、エラーの発生が少なくなっています。

変数名や関数名の**レキシカルレベル**は、それが宣言された位置を表します。プログラムの最上位のレベルを最初のレキシカルレベルにすると、最上位のレベルの関数の引数はレベル 2 になり、関数内で宣言されたローカル変数はレベル 3 になり、入れ子にされた関数の引数パラメータはレベル 4 になり、入れ子にされた関数で定義されたローカル変数はレベル 5 になるというように、レベルが下がってきます。レキシカルレベル i は、レキシカルレベル $i+1$ より高くなります。入れ子にされた関数内の変数のこれらのさまざまなレキシカルレベルをプログラム 10.7に示します。

```
int i;               // level 1
void funct1(int i)   // level 2 for i, level 1 for funct1
{
  int i;             // level 3
  void funct2(int i) // level 4 for i, level 3 for funct2
  {
     int k;
     k = i;          // use i at level 4
     int i=6;        // level 5
     i = 30;         // use i at level 5
     printf("k = %d \n", k);
  }
  i = 10+i;          // use i at level 3
  funct2(i);         // use i at level 3
}
i = 5;               // use i at level 1
funct1(i);           // output: 10
```

プログラム 10.7: Ch のレキシカルレベル

プログラム 10.7では、関数 `funct2()` は関数 `funct1()` の外からは見えません。[4]異なるレキシカルレベルであれば、変数と関数に同じ名前を使用できます。入れ子にされた関数群のレキシカルレベルの上位にある変数および関数がそれぞれのスコープ内にあれば、下位にあるプログラムの部分からそれらの変数および関数へアクセスが可能です。

ただし、レキシカルレベルの上位にあるプログラムの部分からは、レキシカルレベルの下位の変数と関数へはアクセスできません。たとえば、プログラム 10.8に示すように、レキシカルレベル 3 で定義された `funct3()` をレキシカルレベル 2 で呼び出すことはできません。

```
void funct1() {     // level 1
  void funct2() {   // level 2
    void funct3()   // level 3
    { }
  }
  funct3();         // Error: funct3() is at lower level
}
```

プログラム 10.8: レキシカルレベルの上位にあるプログラムの部分から下位にある関数を呼び出すことができない

レキシカルレベルが同じであっても、異なる関数であれば、同じ名前を持つ引数とローカル変数とが関連付けられることはありません。たとえば、

[4]訳注：しかし、推奨はできません。

```
void funct1()
{
  void funct2(int i)   // level 4
  {
    int k;             // level 5
  }
  void funct22(int i); // level 4
  {
    int k;             // level 5
  }
}
```

プログラム 10.9: 引数と異なるローカル関数のローカル変数とは関係付けられていない

.

プログラム 10.9に示すように、異なるローカル関数の funct2() と funct22() の引数 i とローカル変数 k は関連付けられていません。関数 funct2() 内の変数 k が変更されても、関数 funct22() 内の変数 k には影響しません。

プログラム 10.10では、

```
void funct1(complex z1) {       // definition of the function
  int i;
  complex funct2(complex z2) {  // define local funct before it is used
    complex z;
    complex funct3(complex z3) { // double nested function
      return z3;
    }
    z = funct3(z2);
    return z;
  }
  i = funct2(complex(1,2));
}
```

プログラム 10.10: Ch で 2 重に入れ子にされた関数

関数 funct3() をローカル関数 funct2() 内で定義しています。このように、関数を他のローカル関数内で定義することができます。Ch では、入れ子にする関数の数に制限はありません。レキシカルレベルが異なれば、変数の名前だけでなく、関数の名前も同じものを使用できます。異なるレキシカルレベルに同じ名前の変数が複数ある場合、すべての変数スコープ内で、最下位のレキシカルレベルの変数が使用されます。関数の場合も同様です。たとえば、プログラム 10.11では、3 つの異なるレキシカルレベルに同じ名前の関数が 3 つあります。

```
void funct() {       // level 1
  void funct() {     // level 2
    void funct()     // level 3
    { }
  }
  funct();           // invoke funct() at level 2
}
```

プログラム 10.11: 異なるレキシカルレベルで同じ名前を持つ異なる関数

ローカル変数の初期化、形状引継ぎ配列の渡し、引数の参照渡しなど、通常の関数のすべての構文規則を入れ子にされた関数にも適用できます。

```
void funct1(int i)
{
  void funct2(complex A[:][:], &z)
  {
    complex A1[][3]={
                     {ComplexInf,     ComplexNaN,      Inf},
                     {-Inf,           complex(-3,-1), complex(-7,2)},
                     {complex(-4,-3), complex(6,3),    complex(2,1)}
                    };
     z += A[1][2] + A1[1][2];
  }
  float F[4][4];
  complex z = complex(-3,2);
  F[1][2] = -i;
  funct2(F, z);
  printf("z = %f \n", z);
}
funct1(10);            // output: z = complex(-20.000000,4.000000)
```

プログラム 10.12: Ch での初期化、形状引継ぎ配列、入れ子にされた関数の参照

たとえばプログラム 10.12では、ローカル関数 `funct2()` の変数 `A1` は、3x3 の複素数配列として初期化されます。`A1` の最初のディメンションは、配列が宣言段階で初期化されるときに行の数で決定されます。float 配列 F は関数 `funct2()` の引数 `A` に渡されます。`F` からの引継ぎで、配列 `A` の形状は 4x4 となります。関数 `funct2()` の 2 番目の引数 `z` は、参照渡しされます。関数 `funct1()` の最後のステートメントによって出力される、プログラム 10.12の出力結果は次のようになります。

```
z = complex(-20.000000,4.000000)
```

10.5.2 入れ子にされた関数のプロトタイプ

これまでの説明で使用したプログラム例では、ローカル関数は、すべて入れ子にされた関数内で呼び出される前に定義されていました。しかし、ローカル関数の定義は、関数内の任意の場所で行うことができます。定義より先にローカル関数を呼び出す場合は、プログラム 10.13に示すようにローカル関数プロトタイプを使用する必要があります。

```
void funct1()            // level 1
{
  __declspec(local) float funct2(); // local function prototype
  funct2();
  float funct2()         // definition of the local function,
  {
    return 9;
  }
}
```

プログラム 10.13: `funct2()` をローカル関数として修飾する型修飾子`__declspec(local)`

プログラム 10.13では、定義より前に関数 `funct2()` が使用されているため、関数プロトタイプが必要です。ローカル関数であるため、型修飾子`__declspec(local)` を使用して、最上位の通常の C 関数とローカル関数とを区別します。また、関数内のローカル変数と関数定義の宣言を行うための型指定子の前に、ローカル関数の型修飾子を配置することもできますが、プログラム 10.14に示すように、その指定は任意です。

```
void funct1()            // level 1
{
  __declspec(local) float funct2(); // required 'local'
  __declspec(local) int i;          // optional 'local'
  funct2();
  __declspec(local) float funct2()  //optional 'local'
  {
    __declspec(local) int j;        // optional 'local'
    return 9;
  }
}
```

プログラム 10.14: Ch の省略可能な型修飾子

関数内の関数プロトタイプは、型修飾子 `local` でローカル関数として修飾されない場合は、最上位の関数と見なされます。これにより、入れ子にされた関数を言語に追加したときに、既存の C コードが完全に壊れることがないように防止します。たとえば、プログラム 10.15では、`funct2()` という関数が 2 つあります。

```
int main()                           // level 1
{
  void funct1(int i)             // level 2
  {
    void funct2(int i);
    funct2(i);                   // use funct2() at level 1
  }
  void funct2(int i)             // level 2
  {
    printf("i = %d \n", i+1);
  }

  funct1(100);
  funct2(100);                   // use funct2() at level 2
}
void funct2(int i)               // level 1, top level
{
  printf("i = %d \n", i+5);
}
```

プログラム 10.15: 最上位にある型修飾子のない関数プロトタイプ

1 つはメインルーチン `main()` の中で定義され、もう 1 つは最上位の関数です。関数 `funct1()` 内で、プロトタイプ `void funct2(int i)` は、`funct2` という関数名が、vold の戻り値データ型で、int の 1 つの引数を含んでいることをコンパイラに知らせます。関数プロトタイプより前に型修飾子がないため、既定で関数 `funct2()` が最上位の関数となります。したがって、以降の `funct2(i)` の関数呼び出しには、最上位の関数 `funct2()` が使用されます。ローカル関数 `funct2()` の定義の後で、メインルーチン `main()` 内の `funct2(100)` の関数呼び出しにはローカル関数が使用されます。

プログラム 10.15の出力結果は次のとおりです。

i = 105
i = 101

関数が定義より先に呼び出される場合、その関数は int の戻り値の型を含む最上位の関数であると見なされます。引数の数とデータ型は、C の関数プロトタイプと関数定義のどちらか最初に出現するものによって決められます。言い換えれば、関数 `funct()` が呼び出された時点で定義されていない場合は、`int funct()` によってプロトタイプ化されたかのように動作することで、C と互換性を保ちます。

たとえば、プログラム 10.16では、関数 `funct3()` は定義またはプロトタイプ化される前に呼び出されます。既定で、関数 `funct3()` は、int 型の値を返す最上位の関数になります。

```
void funct1(int i) {           // level 1
  void funct2(complex A[:][:]);// funct2(): default function at level 1
  complex A1[3][3];
  i = funct3()+4;   // funct3(): default function at level 1 return int
  funct2(A1);
}
void funct2(complex A[:][:]) { // level 1
  A[1][2] =70;
}
int funct3() {                  // level 1
  int i=90;
  return i;
}
```

プログラム 10.16: 既定で最上位に置かれる、関数定義とプロトタイプ化の前に呼び出された関数

入れ子の階層が深い関数の場合、最上位にも同レベルにも定義されていない関数を呼び出すときには、ローカル関数修飾子を使用して関数の始めの位置で関数をプロトタイプ化することができ、プロトタイプ化された関数は、その関数内で定義されます。このような目的で使用する関数プロトタイプを**補助関数プロトタイプ**と呼びます。

プログラム 10.17では、補助関数プロトタイプが`__declspec(local) void funct3()`であるため、ローカル関数`funct3()`を入れ子にされた関数`funct2()`と関数`funct4()`で使用できます。入れ子にされた関数のスコープ、レキシカルレベル、および関数プロトタイプについては、プログラム 10.18に示す 4 個のコード例で詳細に説明します。

```
void funct1() {        // level 1
  __declspec(local) void funct3(); // auxiliary function prototype
  void funct2() {      // level 2
    funct3();          // use funct3() at level 2
    void funct4() {
      funct3();        // use funct3() at level 2
    }
    funct3();          // use funct3() at level 2
  }
  void funct3()        // level 2
  { }
}
```

プログラム 10.17: 異なるレキシカルレベルで、関数を呼び出すために使用される修飾子`__declspec(local)`

```
/** EXAMPLE 1 **/
void funct1() {        // level 1
  void funct2() {      // level 2
    int i;
    i = funct3();      // use funct3() at level 1
  }
  void funct3()        // level 2
  { }
  funct3();            // use funct3() at level 2
}
int funct3()           // level 1
{ }

/** EXAMPLE 2 **/
void funct1() {        // level 1
  __declspec(local) void funct3();
  void funct2() {      // level 2
    funct3();          // use funct3() at level 2
    void funct3() {    // level 3
      __declspec(local) void funct3();
      funct3();        // use funct3() at level 4
      void funct3()    // level 4
      { }
    }
    funct3();          // use funct3() at level 3
  }
  void funct3()        // level 2
  { }
  funct3();            // use funct3() at level 2
}
```

プログラム 10.18: 入れ子にされた関数のスコープ、レキシカルレベル、プロトタイプを示すサンプルプログラム

```
/** EXAMPLE 3 **/
void funct2() {      // level 1
}
void funct1() {      // level 1
  funct2();          // invoke funct2() at level 1
  void funct2() {    // level 2
    void funct3()    // level 3
    { }
  }
}

/** EXAMPLE 4 **/
void funct2()        // level 1
{ }
void funct1() {      // level 1
  __declspec(local) void funct2();
  funct2();          // invoke funct2() at level 2
  void funct2() {    // level 2
    void funct3()    // level 3
    { }
  }
}
```

プログラム 10.19: 入れ子にされた関数のスコープ、レキシカルレベル、プロトタイプを示すサンプルプログラム (続き)

10.5.3 入れ子にされた再帰関数

Ch では、関数がローカル関数または最上位の関数のどちらに定義されようと、再帰的な状況でも、プログラムのメモリ容量や実行速度に大きな影響を与えません。入れ子にされた関数内では、関数呼び出しのスコープとレキシカル規則に違反しない限り、関数どうしで再帰呼び出しを行うことができます。プログラム 10.20では、関数 10.20は、ローカル関数 `funct2()` と関数自身を再帰的に呼び出します。しかし、関数 `funct2()` は関数自身を再帰的に呼び出すだけです。

```
void funct1(int &i)    // level 1
{
  int funct2(int j)    // level 2
  {
    if(j <=  3)
    {
      printf ("recursively call funct2() j = %d \n", j);
      j++;
      j = funct2(j);
    }
    else
    {
      printf ("exit funct2() j = %d \n", j);
      j++;
    }
    return j;
  }
  i = funct2(i);
  printf ("after call funct2() i = %d \n", i);

  if(i < 6)
    funct1(i);
}
funct1(1);
```

プログラム 10.20: 直接再帰関数

プログラム 10.20の出力結果は次のとおりです。
recursively call funct2() j = 1
recursively call funct2() j = 2
recursively call funct2() j = 3
exit funct2() j = 4
after call funct2() i = 5
exit funct2() j = 5
after call funct2() i = 6

プログラム 10.21で、関数 `funct1()` はローカル関数 `funct2()` を呼び出し、ローカル関数 `funct2()` は上位にある関数 `funct1()` を呼び出します。

```
int funct1(int i)      // level 1
{
  int funct2(int j)    // level 2
  {
    if(j <=  3)
    {
      printf ("recursively call funct2() j = %d \n", j);
      j++;
      j = funct1(j);
    }
    else
    {
      printf ("exit funct2() j = %d \n", j);
      j++;
    }
    return j;
  }
  i = funct2(i);
  printf ("after call funct2() i = %d \n", i);

  if(i < 6)
    i = funct2(i);
  return i;
}
funct1(1);
```

プログラム 10.21: 間接再帰関数

プログラム 10.21の出力結果は次のとおりです。

recursively call funct2() j = 1
recursively call funct2() j = 2
recursively call funct2() j = 3
exit funct2() j = 4
after call funct2() i = 5
exit funct2() j = 5
after call funct2() i = 6
after call funct2() i = 6
after call funct2() i = 6

プログラム 10.20と 10.21では、再帰関数呼び出しは入れ子にされた関数内だけに制限されます。Chでは、入れ子にされているどの関数からも最上位の関数を呼び出すことができます。間接再帰関数が最上位の関数を呼び出す例をプログラム 10.22に示します。ここでは関数 `funct1()` がローカル関数 `funct2()` を呼び出します。ローカル関数 `funct2()` は、関数 funct1() を呼び出す最上位の関数 `funct3()` を呼び出します。したがって、関数 `funct1()`、`funct2()`、`funct3()` はクローズドループを形成します。プログラム 10.22でのもう 1 つのプログラミング方法は、関数 `funct1()` と関数 `funct2()` を同じレキシカルレベルで扱う方法です。

```
int funct1(int i)      // level 1
{
  int funct2(int j)    // level 2
  {
    if(j <=  3)
    {
      printf ("recursively call funct2() j = %d \n", j);
      j++;
      j = funct3(j);
    }
    else
    {
      printf ("exit funct2() j = %d \n", j);
      j++;
    }
    return j;
  }
  i = funct2(i);
  printf ("after call funct2() i = %d \n", i);
  return i;
}
funct1(1);
int funct3(int i)
{
  i = funct1(i);
  return i;
}
```

プログラム 10.22: 最上位の関数による間接再帰関数

これにより、プログラム 10.23に示すように、関数 `funct3()` を省くことができます。

```
int main() {
  __declspec(local) int funct2();
  int funct1(int i)
  {
    i = funct2(i);
    printf ("after call funct2() i = %d \n", i);
    return i;
  }
  funct1(1);

  int funct2(int j)
  {
    if(j <=  3)
    {
      printf ("recursively call funct2() j = %d \n", j);
      j++;
      j = funct1(j);
    }
    else
    {
      printf ("exit funct2() j = %d \n", j);
      j++;
    }
    return j;
  }
}
```

プログラム 10.23: 同じレキシカルレベルの間接再帰関数

次に示すプログラム 10.23の出力結果は、プログラム 10.22の出力結果と同じです。
recursively call funct2() j = 1
recursively call funct2() j = 2
recursively call funct2() j = 3
exit funct2() j = 4
after call funct2() i = 5
after call funct2() i = 5
after call funct2() i = 5
after call funct2() i = 5

10.6 ポインタを使用した、関数の引数の参照渡し

Ch が引数を関数に渡す場合は、値で渡します。しかし、多くの場合、関数に渡された引数の値を変更したいときがあります。並べ替えルーチンを使用して、順番が誤った a と b の 2 つの要素を関数 `swap()` で入れ替える場合、次のコードでは動作しません。

```
swap(a, b);
```

ここで、スワップ関数を次のように定義します。

```
void swap(int x, int y) { // doesn't work as expected
    int temp;

     temp = x;
     x = y;
     y = temp;
}
```

値呼び出しされているため、swap() は呼び出し元の関数内の引数 a と b に影響できません。関数 swap() 内では、a と b のそれぞれのコピーである x と y とを交換できるだけです。
ポインタを使用すると、変数のアドレスを関数に渡し、それらのアドレスを使って間接的に変数へアクセスすることができます。明示的にポインタを使用すると、プログラム内の関数呼び出しは次のようになります。

```
swap(&a, &b)
```

上述したように、演算子'&' は変数のアドレスを示し、式&a は a へのポインタを表します。この場合、関数 swap() には値 a と値 b のコピーではなく、それらのアドレスを使用します。

```
void swap(int *pa, int *pb) {
    int temp;

    temp = *pa; // contents of pointer
    *pa = *pb;
    *pb = temp;
}
```

swap() の関数定義では、パラメータはポインタ pa とポインタ pb として宣言され、呼び出し元の関数の a と b へはポインタ pa とポインタ pb を介して間接的にアクセスします。

10.7 関数内の可変長の引数

Ch では、可変長の引数を関数に渡すことができます。一部のアプリケーションでは、関数に渡される引数の数は事前には分からず、ケースによって異なることがあります。この機能では、関数が扱う引数リストの長さが、ケースによってさまざまな長さになります。可変長の引数を取る典型的な関数は、次のように定義されます。

```
#include <stdarg.h>
type1 funcname (arg_list, type2 paramN, ...) {
  va_list ap;
  type3 v;                // first unnamed argument
  va_start(ap, paramN);   // initialize the list
  v = va_arg(ap, type3); // get 1st unnamed argument from the list
```

```
    ...                     // get the rest of the list
    va_end(ap);             // clean up the argument list
    ...
  }
```

ここで、arg_list は、指定した引数の引数リストです。paramN は最後に指定された引数です。v は型 type3 の最初の未指定の引数です。

データ型 **ChType_t** はヘッダファイル **stdarg.h** でも定義されます。標準のヘッダファイル **stdarg.h** には、引数リストの処理方法を定義するマクロ定義セットも含まれています。配列型と関数に使用するマクロのいくつかは表 10.1に挙げています。

表 10.1: ヘッダファイル **stdarg.h** で定義される、可変長の引数リストを扱うマクロ

マクロ	説明
VA_NOARG	**va_start**() の 2 番目の引数 (関数に渡す引数がない場合) 。
CH_UNDEFINETYPE	配列でない。
CH_CARRAYTYPE	C 配列。
CH_CARRAYPTRTYPE	C 配列のポインタ。
CH_CARRAYVLATYPE	C VLA 配列。
CH_CHARRAYTYPE	Ch 配列。
CH_CHARRAYPTRTYPE	Ch へのポインタ。
CH_CHARRAYVLATYPE	Ch VLA 配列。
va_arg	呼び出し元の関数内に指定された型と次の引数の値を含む式に拡張する。
va_arraytype	次の引数が配列であるかどうかを決定する。
va_arraydim	可変長の引数の配列次元を取得する。
va_arrayextent	可変長の引数の配列内の要素の数を取得する。
va_arraynum	可変長の引数の配列内の要素の数を取得する。
va_copy	**va_list** のコピーを作成する。
va_count	可変長の引数の数を取得する。
va_datatype	可変長の引数のデータ型を取得します。
va_end	関数から正常復帰させる。
va_start	以降の他のマクロによる使用のために *ap* を初期化する。
va_tagname	可変長の引数の struct/class/union 型のタグ名を取得する。

これらのマクロに加え、型 **va_list** はヘッダファイル **stdarg.h** でも定義されます。このマクロを使用して、引数リストの情報を保持でき、各引数を順番に参照することが可能なオブジェクトを宣言します。Ch の命名規則では、このオブジェクトは ap と呼ばれます。**VA_NOARG**、 **va_count** 、**va_datatype** 、**va_arraydim** 、**va_arrayextent** 、**va_arraynum** 、**va_arraytype** 、および **va_tagname** などのマクロは、多態性関数の実装に有用です。引数の配列型に応じて、関数 **va_arraytype**() は、**CH_UNDEFINETYPE** 、**CH_CARRAYTYPE** 、**CH_CARRAYPTRTYPE** 、**CH_CARRAYVLATYPE** 、**CH_CHARRAYTYPE** 、**CH_CHARRAYPTRTYPE** 、**CH_CHARRAYVLATYPE** などのいずれかのマクロの値を返しま

す。これらの関数の適用については、セクション 19.9で詳しく説明します。

マクロ **va_start** は ap を初期化し、最初の名前なし引数をポイントします。ap が使用される前に一度は呼び出されなければなりません。可変長の引数リストへアクセスする際に特殊な役割を果たす一番右の名前付きパラメータは、ここで paramN と指定されます。 **va_start** がこれを使用して、開始します。その後で、 **va_arg()** を呼び出すたびに名前なし引数を 1 つ返し、ap を次の名前なし引数に進めます。マクロ **va_arg** は引数として型名を取り、どの型を返すか、取得する次の名前なし引数がどこにあるかを判断します。データ型は、**int**、ポインタなどの単純なデータ型か、クラス、計算配列などの集積データ型のいずれかになります。最後に、引数をすべて読み込んだ後、関数から戻る前に、マクロ **va_end** を呼び出して、引数リストをクリーンアップする必要があります。たとえば、プログラム 10.24では、関数 f1() は可変長の引数を取ります。最後の名前付き引数 arg_num で指定される引数の長さは、1〜6 の範囲です。プログラム 10.24の出力結果をプログラム 10.25に示します。

```
#include<stdarg.h>
struct tag {int i; float j;};

void f1(int arg_num, ...) {
  va_list ap;
  int i;
  char *str;
  struct tag s;
  int *a;
  int a1;

  va_start(ap, arg_num);

  if (arg_num <= 1)
    return;
  if (arg_num >= 2) {
    i = va_arg(ap, int);
    printf("\nthe 2nd argument is %d\n", i);
  }
  if (arg_num >= 3) {
    str = va_arg(ap, char *);
    printf("the 3rd argument is %s\n", str);
  }
  if (arg_num >= 4) {
    s = va_arg(ap, struct tag);
    printf("the 4th argument s.i is %d, s.j is %f\n", s.i, s.j);
  }
  if (arg_num >= 5) {
    a = va_arg(ap, int *);
    printf("the 5th argument a is %d, %d, %d\n", a[0], a[1], a[2]);
  }
  if (arg_num >= 6) {
    a1 = va_arg(ap, int);
    printf("the 6th argument a1 is %d\n", a1);
  }

  va_end(ap);
  return;
}
int main(){
  struct tag s = {1, 2.0};
  int a[] = {1, 2, 3};
  int arg_num = 3;
  f1(arg_num, 3, "abc");
  arg_num = 6;
  f1(arg_num, 6, "def", s, a, a[1]);

  return 0;
}
```

プログラム 10.24: 可変長の引数リスト

```
the 2nd argument is 3
the 3rd argument is abc

the 2nd argument is 6
the 3rd argument is def
the 4th argument s.i is 1, s.j is 2.000000
the 5th argument a is 1, 2, 3
the 6th argument a1 is 2
```

プログラム 10.25: プログラム 10.24の出力結果

C では、可変長の引数リストを取る関数には、可変パラメータリストの前に名前付きパラメータが少なくとも１つ必要になります。Ch では、可変パラメータリストの前に名前付き引数がない場合、**va_start** がマクロ VA_NOARG を使用して開始します。たとえば、プログラム 10.26では、関数 f2() は名前付き引数を取りません。

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

void f2(...) {
  va_list ap;
  int vacount;
  int i, num = 0;

  va_start(ap, VA_NOARG);
  vacount = va_count(ap);
  printf("vacount = %d\n", vacount);

  while(num++, vacount--) {
    i = va_arg(ap, int);
    printf("argument %d = %d, ", num, i);
  }
  printf("\n\n");
  va_end(ap);
  return;
}

int main(){
  f2(1);
  f2(1, 2, 3);
  f2(1, 2, 3, 4, 5);

  return 0;
}
```

プログラム 10.26: 引数リスト内の名前なし引数

この場合は、 **va_start** が VA_NOARG を使用することができます。関数に渡される引数の数は、マクロ **va_count** で取得できます。プログラム 10.26の出力結果をプログラム 10.27に示します。他の機能と組み合わせることで、これは関数のポリモーフィズムにとって有用です。

```
vacount = 1
argument 1 = 1,

vacount = 3
argument 1 = 1, argument 2 = 2, argument 3 = 3,

vacount = 5
argument 1 = 1, argument 2 = 2, argument 3 = 3, argument 4 = 4, argument 5 = 5,
```

プログラム 10.27: プログラム 10.26の出力結果

va_list のオブジェクトとして、ap はマクロ **va_copy** でコピーしたり、引数として関数へ渡したりすることができます。プログラム 10.28では、 **va_list** ap2 のオブジェクトは ap のコピーです。

```
#include <stdarg.h>

int  funct2(int num, va_list ap) {
  int args;
  while(num--) {
    args = va_arg(ap, int);
    printf("args in funct2() is %d\n", args);
  }
}

void funct1(int arg_num, ...) {
  va_list ap, ap2;
  int args;
  int num;

  va_start(ap, arg_num);
  printf("print with ap\n");
  args= va_arg(ap, int); // ap points to the next
  printf("args in funct1 is %d\n", args);
  va_copy(ap2, ap);      // ap2 starts from the second argument

  num = arg_num - 1;
  while(num--) {
    args= va_arg(ap, int);
    printf("args in funct1 is %d\n", args);
  }
  va_end(ap);

  printf("\nprint with ap2\n");
  num = arg_num - 1;
  while(num--) {
    args= va_arg(ap2, int);
    printf("args in funct1 is %d\n", args);
  }
  va_end(ap2); // for va_copy()

  /* pass ap as argument to functions */
  printf("\npass ap to another function\n");
  va_start(ap, arg_num);  // restart
  funct2(arg_num, ap);
  va_end(ap);
}

int main(){
  int arg_num = 3;
  funct1(arg_num, 1, 2, 3);
}
```

プログラム 10.28: コピーされ、引数として渡される `ap`

コピーされたオブジェクト `ap2` は、`ap` と同じ状態になります。つまり、コピーされた時に `ap` がポイントしていた同じ引数を `ap2` もポイントします。この例では、`ap2` は可変長の引数リストの 2 番目の引数から開始します。**va_copy** マクロの各呼び出しは、対応する **va_end** マクロの呼び出しと一致します。関数 `funct2()` は型 **va_list** の引数を取ります。プログラム 10.28では、`ap` は引数とし

て funct2() に渡されます。プログラム 10.28の出力結果をプログラム 10.29に示します。

```
print with ap
args in funct1 is 1
args in funct1 is 2
args in funct1 is 3

print with ap2
args in funct1 is 2
args in funct1 is 3

pass ap to another function
args in funct2() is 1
args in funct2() is 2
args in funct2() is 3
```

プログラム 10.29: プログラム 10.28の出力結果

可変長の引数を使用して、異なるデータ型の配列を関数の同じ引数に渡すことができます。例として、ヘッダファイル **numeric.h** で定義された関数プロトタイプを含む関数 **lindata()**

```
int lindata(double first, double last, ... /* type a[:]...[:] */);
```

のソースコードをプログラム 10.30に示します。

```
/* File: lindata.chf */
#include <stdarg.h>
#include <stdio.h>
int lindata(double first, double last, ...){
    va_list ap;
    int i, n;
    ChType_t dtype;
    double step;
    void *vptr;

    va_start(ap, last);
    if(!va_arraytype(ap)) {
      fprintf(stderr, "Error: 3rd argument of %s() is not array\n", __func__);
      return -1;
    }

    n = va_arraynum(ap);
    double a[n];
    step = (last - first)/(n-1);
    for(i=0; i<n; i++) {
       a[i]=first+i*step;
    }

    dtype = va_datatype(ap);
    vptr = va_arg(ap, void*);
    arraycopy(vptr, dtype,  a, CH_DOUBLETYPE, n);
    // or arraycopy(vptr, dtype,  a, elementtype(double), n);
    return n;
}
```

プログラム 10.30: 関数 **lindata()** のソースコード

この関数は、引数 `first` と引数 `last` の入力でそれぞれ指定した初期値と最終値の間をスペースで区切った連続的なデータを生成します。関数 **lindata()** は関数 **va_arraynum()** を呼び出して、渡された配列 `a` の要素の数を特定します。その後、この情報を使用して、スペース区切りされた連続的なデータセットを生成します。結果は、関数 **arraycopy()** を使用して、最終的に、3 番目の渡された引数内の配列 `a` にコピーされます。生成されたデータポイントの総数は、戻り値として渡されます。

ヘッダファイル **stdarg.h** で定義された関数 **arraycopy()** には、次のプロトタイプが含まれます。

```
int arraycpoy(void *des, ChType_t destype,
                 void *src, ChType_t srctype, int n);
```

このプロトタイプを使用すると、可変長の引数リストを使用して、異なるデータ型の配列の結果を、呼び出された関数から呼び出し元の関数へ渡すことができます。プログラム 10.31では、int 型の配列 `a` と main() 関数の double 型の計算配列 `b` は、関数 `lindata()` を使用してスペース区切りされた連続する値で割り当てられ、3 番目の引数として呼び出し元の関数に返されます。

```
#include <numeric.h>

int main () {
    int i, a[6], *p;
    array double b[6];

    lindata(2, 12, a);
    printf("a = ");
    for(i=0; i<6; i++) {
      printf("%d ", a[i]);
    }
    p = &a[0];
    lindata(20, 120, p, 6);
    printf("\na = ");
    for(i=0; i<6; i++) {
      printf("%d ", a[i]);
    }
    printf("\nb = ");
    lindata(2, 12, b);
    printf("%g", b);
}
```

プログラム 10.31: 関数 **arraycopy()** を使用した、関数 **lindata()** の引数として渡された配列のコピー

プログラム 10.31の出力結果を図 10.2に示します。

```
a = 2 4 6 8 10 12
a = 20 40 60 80 100 120
b = 2 4 6 8 10 12
```

図 10.2: プログラム 10.31の出力結果

10.8　関数へのポインタ

Ch では、関数へのポインタを定義できます。各関数には、メモリに配置されたプログラミングステートメントが含まれています。関数ポインタとは、関数のアドレスを含む変数です。関数のアドレスは関数のエントリポイントです。このため、関数を呼び出すのに関数ポインタを使用できます。また、関数ポインタを割り当てたり、配列に配置したり、関数へ渡したり、関数で返したりすることなどもできます。関数ポインタは次のように宣言します。

```
void (*f1) (void);
int  (*f2) ();
int  (*f3) (float f);
typedef int (*PF)(int i);
PF f4;
```

ここで、`f1` は、戻り値も引数もない関数へのポインタとして宣言されます。`f2` は、引数を含むまたは含まない整数値を返す関数として宣言されます。`f3` は、整数を返し、**float** 型の引数を含む関数

へのポインタとして宣言されます。他のデータ型と同じく、関数へのポインタをユーザー定義のデータ型として定義できます。データ型 PF は、整数値を返し、**int** 型の引数を取る関数へのポインタとして型定義されます。したがって、f4 はタイプ PF(関数へのポインタ) の変数です。プログラム 10.32に関数へのポインタの使用方法を示します。

```
int fun (float f) {
  printf("f = %f\n", f);
  return 0;
}

int main() {
  int (*pf)(float f);

  fun(10);

  pf = fun; // no & before fun
  pf(20);   // call function fun by calling pf

  return 0;
}

/* execution and output
f = 10.000000
f = 20.000000
*/
```

プログラム 10.32: 関数へのポインタの使用方法

fun() は、pf でポイントされた関数と同じプロトタイプを含む通常の関数です。

pf の宣言と割り当ての後で、pf を使用して関数 fun を呼び出すことができます。プログラム 10.32の実行と出力結果は、プログラムの最後に追加されます。

配列名と同じく、関数名は関数のアドレスを表します。アドレス演算子'&' は、C でも Ch でも無視されます。たとえば、次のステートメント

```
pf = &fun;
```

は、次のように扱われます。

```
pf = fun;
```

他のポインタと同じく、関数を参照する 2 個のポインタどうしを比較できます。たとえば、次のような場合、

```
int fun (float f) {
    printf("f = %f\n", f);
    return 0;
}
int (*pf1)(float f);
```

```
int (*pf2)(float f);

pf1 = fun;
pf2 = fun;
```

等式 pf1 == pf2 が保持されます。

関数へのポインタは配列または構造体に含めることができます。関数へのポインタの配列は、メニューを実装する効果的な方法です。プログラム 10.33では、配列 options に関数へのポインタの 3 つの要素が含まれています。

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

int opt0() {
    printf("to handle option 0\n");
    return 0;
}

int opt1() {
    printf("to handle option 1\n");
    return 0;
}

int opt2() {
    printf("to exit\n");
    exit(0);
}

int getChoice() {
    int i;
    printf("input the choice (0,1,2): ");
    scanf("%d", &i);
    if (i > 2 || i < 0) i = 2;
    return i;
}

typedef int (*PF)();
int main() {
   // or int (*options[])() = {
   PF options[] = {
      opt0,
      opt1,
      opt2,
   };

   do {
       options[getChoice()]();
   }
   while(1);

   return 0;
}

/***** execution and output
input the choice (0,1,2): 0
to handle option 0
input the choice (0,1,2): 1
to handle option 1
input the choice (0,1,2): 2
to exit
******/
```

プログラム 10.33: 関数へのポインタを使用するメニューの実装

配列 options は、`int (*options[])()` (int の値を返す関数へのポインタの配列) として定義できます。プログラム 10.33では、配列 options の宣言は、`typedef int (*PF)()` (int の値を返す関数へ

のポインタ) として宣言される新しいデータ型 PF によって簡素化されています。関数 getChoice() は、対応する関数を呼び出すために、配列 options の添字として使用される整数値を返します。プログラム 10.33の対話的な実行と出力結果は、プログラムの最後に追加されます。
関数へのポインタを引数として関数に渡すことができます。コールバック関数を設定するのによく使用されます。プログラム 10.34に、関数へのポインタを関数の引数として使用する例を示します。

```
#include<stdio.h>

int f1(int i) {
    printf("i = %d\n", i);
    return 0;
}

int f2(int (*pf)(), int i) {
 pf(i);
 return 0;
}

int main() {
   f2(f1, 5);
   return 0;
}

/* execution and output
i = 5
*/
```

プログラム 10.34: 関数ポインタを関数への引数として渡す例

関数 f2 は 2 つの引数を取ります。1 つは関数ポインタ pf で、もう１つは整数値です。関数 f2 内では、関数ポインタの引数は、int の引数を取る関数を呼び出すために使用されます。メイン関数では、関数 f1() の名前が関数ポインタとして f2() に渡されます。プログラム 10.34の実行と出力結果は、プログラムの最後に追加されています。
通常のポインタと同じく、関数ポインタは、関数の引数になるだけでなく、関数の戻り値になることもできます。プログラム 10.35は、プログラム 10.33を書き換えたものです。

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

typedef int (*PF)();

int opt0() {
    printf("to handle option 0\n");
    return 0;
}

int opt1() {
    printf("to handle option 1\n");
    return 0;
}

int opt2() {
    printf("to exit\n");
    exit(0);
}

int getChoice() {
    int i;
    printf("input the choice (0,1,2): ");
    scanf("%d", &i);
    if(i > 2 || i < 0)
       i = 2;
    return i;
}

// or int (*processChoice(int i))() {
PF processChoice(int i) {
    switch(i) {
      case 0:
        return opt0;
      case 1:
        return opt1;
      default:
        return opt2;
    }
}

int main() {
   do {
     // call function returned from processChoice()
     processChoice(getChoice())();
   }
   while(1);
   return 0;
}
```

プログラム 10.35: 関数ポインタを返す例

関数へのポインタの配列の代わりに、関数 `processChoice()` を使用して、異なるオプションを処理できます。関数 `processChoice()` の戻り値のデータ型 `PF` は、関数ポインタ型として定義されます。関数 `processChoice()` は、`PF processChoice(int i);` または

`int (*processChoice(int))();` のいずれかでプロトタイプ化できます。
関数を使って関数ポインタを取得するもう１つの方法は、関数ポインタのアドレス (関数へのポインタのポインタ) を関数に渡す方法です。プログラム 10.36では、関数 `processChoice2()` は 2 つの引数を取ります。

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

int opt0() {
    printf("to handle option 0\n");
    return 0;
}

int opt1() {
    printf("to handle option 1\n");
    return 0;
}

int opt2() {
    printf("to exit\n");
    exit(0);
}

int getChoice() {
    int i;
    printf("input the choice (0,1,2): ");
    scanf("%d", &i);
    if(i > 2 || i < 0)
       i = 2;
    return i;
}

void processChoice2(int(**pf)(), int i) {
    switch(i) {
      case 0:
        *pf = opt0;
        break;
      case 1:
        *pf = opt1;
        break;
      default:
        *pf = opt2;
    }
    return;
}

int main() {
   int(*pf)();
   do {
     processChoice2(&pf, getChoice());
     pf();
   }
   while(1);
   return 0;
}
```

プログラム 10.36: 関数ポインタのアドレスを関数への引数として渡す例

1 つは関数へのポインタのポインタです。もう 1 つは関数 `getChoice()` によって返されたオプションである整数値です。関数 `main()` では、関数ポインタ `pf` のアドレスが関数 `processChoice2()`

に渡された後、オプション i に従って適切な関数ポインタが pf のアドレスに割り当てられます。関数 processChoice2() を呼び出した後、関数 pf へのポインタを使用して opt1()、opt2()、opt3() のいずれかの関数が呼び出されます。

入れ子にされた関数へのポインタは、プログラム 10.37に示すように、通常の関数へのポインタと同じように扱われます。

```
int main() {
  int func(int i) {
    printf("i in func1() = %d\n", i);
    return 2*i;
  }
  int j;
  int (*fp)(int);

  fp =  func;
  j = fp(10);
  printf("j in main() = %d\n", j);
}
/* output
i in func1() = 10
j in main() = 20
*/
```

プログラム 10.37: 入れ子にされた関数へのポインタ。

ここでは、func() は入れ子にされた関数 func() へのポインタです。プログラム 10.37の実行結果は、プログラムの最後に追加されています。

コールバック関数の登録に関数へのポインタを使用できます。Ch では、ローカル関数をコールバック関数として登録すると、ローカル変数、ローカル関数の引数、またはグローバル変数しか使用できなくなりますが、入れ子にする関数の囲みブロックでは、中間変数は許可されません。

10.9 関数間の通信

入れ子にされた関数のおかげで、 Ch での関数間の通信は、C よりも多くのオプションが用意されています。Ch での関数間の通信方法の概要は次のとおりです。

Ch の関数は、戻り値、引数、および変数を使って、上位のレキシカルレベルで通信できます。関数への入力は、引数から取得するか、または、上位のレキシカルレベルの変数を使用できます。関数の出力には、上位のレキシカルレベルの戻り値、引数、および変数が含まれます。呼び出された関数から呼び出し元の関数へ結果を返すには、ポインタを使用して値渡しするか、参照を使用して参照渡しします。関数を式のオペランドとして使用する場合、関数の結果は戻り値として実装される必要があります。異なる関数間で多数の変数を共有する必要がある場合は、長い引数リストを使用するよりも上位のレキシカルレベルの変数のほうが便利です。入れ子にされた関数を使用して作成した Ch プログラムは、モジュール化される傾向があります。読みやすさの面から言えば、関数は複数のファイルにまたがって定義するべきではありません。そのため、関数内のローカル変数は、関数が定義されているファイルの外からは見えないようになっています。ローカル関数間で通信を行うには上位のレ

キシカルレベルの変数が有用です。ローカル関数があるデータを共有する必要があるのに、互いに関数を呼び出すことがない場合は、特に役立ちます。関数間のデータのやり取りが多くなりすぎないように、関数自身を含む関数は、引数と戻り値を使って他の関数と通信する必要があります。

10.10 main() 関数とコマンドライン引数

メインルーチン main() は特別な関数です。コマンドライン引数またはパラメータを、次の 2 つの書式で、関数 main() の引数を使ってプログラムに渡すことができます。

```
int main(int argc, char *argv[], char **environ) {
    ...
}
int main(int argc, char *argv[]) {
    ...
}
```

関数 main() には最大 3 つの引数を含めることができます。引数の数を示す argc という 最初の引数はコマンドラインの引数の数を表し、引数のベクタを示す argv という 2 番目の引数は、可変長の文字列の配列を参照するポインタです。各文字列にはコマンドラインの引数が 1 つ含まれています。したがって、引数 argv は char へのポインタのポインタであると見なすこともできます。そうであれば、代わりに関数 main() を次のように書くことができます。

```
int main(int argc, char **argv) {
    ...
}
```

省略可能な 3 番目の引数は環境変数のテーブルを参照するポインタです。プログラムが呼び出されると、関数 main() の引数 argc と argv の値は Ch のプログラミング環境によってプログラムに渡されます。標準 C では、argv[0] がプログラムの名前になるため、argc の値は 1 以上になります。argc が 1 の場合、プログラム名の後にはコマンドライン引数がありません。さらに、argv[argc] の値は null ポインタとなります。たとえば、プログラム10.38には、スペースで区切られた連続的なコマンドライン引数が反映されます。

```
int main(int argc, char *argv[])
// or int main(int argc, char **argv)
{
  int i;
  for(i = 0; i < argc; i++)
    printf("%s ", argv[i]);
  /* or */
  // do{
  //   printf("%s ", argv[i]);
  // }while(argv[++i] != NULL);
  printf("\n");
}
```

プログラム 10.38: main() ルーチンのコマンドライン関数

プログラム10.38のファイル名を echo とした場合、Ch コマンドラインモードでプログラム10.38を次のように実行できます。

```
> echo testing example -a
echo testing example -a
>
```

ここで、4 つの引数からなるコマンドライン echo testing example -a は、プログラムの出力にもなります。

Unix システムのプログラムでよく行われる方法の 1 つに、引数の先頭にマイナス記号 '–' を付けてオプションを示す方法があります。たとえば、 Ch の **which.ch** プログラムでは、有効なオプションとして-a と-v を取ることができます。which -a コマンドは、環境変数とヘッダファイルを含め、すべてのコマンドを検索します。名前が見つからない場合は、コマンド which -v によって検索メッセージが送信されます。2 つのオプションを、たとえば which -a -v または which -va という形式で、同時に指定できます。

プログラム10.39は、プログラム **which.ch** から抽出された、コマンドライン引数を扱うコードです。

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

int main(int argc, char **argv) {
  char *s;
  int a_option = false; // default, no -a option
  int v_option = false; // default, no -v option

  if(argc == 1){         // no argument
    fprintf(stderr, "Usage: which [-av] names \n");
    exit(1);
  }

  argc--; argv++;        // for every argument beginning with -
  while(argc > 0 && **argv == '-')
  {
      /* empty space is not valid option */
      for(s = argv[0]+1; *s&&*s!=' '; s++) { // for -av
        switch(*s)
        {
          case 'a':
            a_option = true;      // get all possible matches
            break;
          case 'v':
            v_option = true;      // print message
            break;
          default:
            fprintf(stderr,"Warning: invalid option %c\n", *s);
            fprintf(stderr, "Usage: which [-av] names \n");
            break;
        }
      }
      argc--; argv++;
  }

  if(a_option)
    printf("option -a is on\n");
  if(v_option)
    printf("option -v is on\n");
  while(argc > 0) { // print out the remaining arguments
      printf("%s\n",  *argv);
      argc--; argv++;
  }
  return 0;
}
```

プログラム 10.39: コマンドライン引数を扱うプログラム commandline.ch

ここで、変数 a_option と変数 v_option は、オプションの-a と-v がオンまたはオフであることを示します。これらの値は既定では false となっています。コマンドライン引数がない場合、プログラムはエラーメッセージを出力します。これはプログラム **which.ch** に少なくとも 1 つの引数 (検索対象の名前) があるためです。このプログラム内の while ループは、マイナス記号 (–) で始まる引数をすべて処理します。ポインタ argv でポイントされる引数の先頭にマイナス記号がある場合、

```
**argv == '-'
```

次の等号が保持されます。

```
s = argv[0]+1
```

次のステートメントの s はこの引数の 2 番目の文字を参照します。ポインタのポインタの詳細については、セクション9.4を参照してください。これらの引数に文字 'a' と 'v' があれば、変数 a_option と変数 v_option がそれぞれ true に設定されます。他の文字が検出されると、エラーメッセージが出力されます。

プログラム10.39の最後に、オプションおよび残りのコマンドライン引数が出力されます。プログラム10.39のファイル名を commandline.ch にして、異なるオプションでプログラム10.39を実行した場合、結果は以下のようになります。

```
> commandline.ch -a -v arg1
option -a is on
option -v is on
arg1
> commandline.ch -av arg1
option -a is on
option -v is on
arg1
> commandline.ch -v arg1 arg2
option -v is on
arg1
arg2
```

関数 **main()** は、3 つの引数付きで使用することも可能です。3 番目のオプションの引数は、環境変数のテーブルへのポインターです。以下のプログラムを使用すると、すべての環境変数とそれらに対応する値を出力できます。

```
#include <stdio.h>
int main(int argc, char *argcv[], char **environ) {
    int i;
    for(i=0; environ[i] != NULL; i++) {
        printf("environ[%d] = %s\n", i, environ[i]);
    }
}
```

他の方法として、ヘッダファイル stdlib.h で定義された変数 environ を使用し、次のプログラムですべての環境変数とそれらの対応する値を出力できます。

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

```
int main() {
    int i;
    for(i=0; environ[i] != NULL; i++) {
        printf("environ[%d] = %s\n", i, environ[i]);
    }
}
```

10.11 関数ファイル

Ch プログラムは多数の個別ファイルに分割できます。各ファイルは、プログラムのあらゆる部分からアクセス可能な、最上位にある多数の関連した関数で構成されます。最上位にある各関数は、その後で、前のセクションで説明したとおり、入れ子にされた形式で多くのローカル関数を含むことができます。複数の関数が記述されているファイルには、通常、 Ch プログラムの一部として認識されるためのサフィックスである`.ch`が付加されます。Ch プログラミング環境では、関数ファイルを作成できます。Ch の**関数ファイル**は、1 つの関数定義だけを含むファイルです。関数ファイルの名前は、`qsort.chf`の例のように`.chf`で終わります。関数ファイルの名前と、関数ファイル内の関数定義の名前は、同じにする必要があります。関数ファイルを使用して定義した関数は、Ch プログラミング環境で、システムに組み込みの関数と同様に扱われます。たとえば、プログラム 10.40に示すように、`qsort.chf`というファイルにプログラムが含まれていると、関数`qsort()`はシステムに組み込みの関数として扱われ、1 次元配列の要素を昇順に並べ替えるために呼び出すことができます。

```
/* qsort: sort v[left] .. v[right] into increasing order */
void qsort(int v[], int left, int right) {
  int i, last;
  /* interchange v[i] and v[j] */
  void swap(int v[], int i, int j)  // local function
  {
    int temp;
    temp = v[i];  v[i] = v[j];  v[j] = temp;
  }

  if(left >= right)
    return;
  swap(v, left, (left + right)/2);
  last = left;

  for(i = left+1; i <= right; i++)
    if(v[i] < v[left])
      swap(v, ++last, i);

  swap(v, left, last);
  qsort(v, left, last-1);
  qsort(v, last+1, right);
}
```

プログラム 10.40: 関数`qsort()`に対する関数ファイル`qsort.chf`

プログラム 10.40では、関数`qsort()`は、1 次元配列の要素を昇順に並べ替えるために再帰的に呼

び出されます。関数 swap() は関数 qsort() によってのみ使用され、ローカル関数として定義されます。したがって、プログラム 10.41の例に示すように、関数 qsort() をスタンドアロンのシステム関数として使用できます。

```
int main() {
  int i, a[] = {2, 6, 5 , 3, 4, 1};

  qsort(a, 0, 5);
  for(i=0; i<=5; i++) {
    printf("a[%d] = %d ", i, a[i]);
  }
  printf("\n");
}
```

プログラム 10.41: 関数ファイル qsort.chf. を使用するプログラム

プログラム 10.41では、関数 qsort() は、関数ファイル qsort.chf 中に定義されている関数プロトタイプを呼び出すため、main() 関数に関数プロトタイプを指定せずに呼び出されます。関数 qsort() の戻り値の型は void です。関数ファイルがない場合、プロトタイプ化または定義される前に呼び出される、関数の既定の戻り値の型は int です。プログラム 10.41の出力結果は次のとおりです。
a[0] = 1 a[1] = 2 a[2] = 3 a[3] = 4 a[4] = 5 a[5] = 6

Ch では、プログラム 10.40に示すように、再帰的に呼び出すことが可能な関数内でローカル関数を定義できます。関数ファイルの関数は、他の関数ファイルを呼び出すことができるだけでなく、間接的に再帰的に関数自身をも呼び出すことができます。キーワードを変えることで置き換えが可能なシステムに組み込みの関数と同様に、Ch プログラムでは関数ファイルで定義された関数を非表示にできます。関数が呼び出される前にプログラムで定義されると、プログラムでユーザー定義関数が使用されます。同様に、呼び出される前に関数がプロトタイプ化されるなら、その関数はユーザー定義関数です。関数がプロトタイプ化される場合、その関数が関数ファイルで定義されているかどうかにかかわらず、ユーザーはどこかで関数を定義する必要があります。関数ファイルで多くの関数を定義できますが、関数ファイルに *1* **つ**の関数と多くのローカル関数だけを含めるほうが良い方法です。たとえば、関数 funct() を最上位のシステム機能として扱いたい場合、プログラム 10.42に示すように、関数ファイル funct.chf に他の関数を含めるのはよいデザインではありません。

```
int funct()
{
   void localfunct1() // OK
   {  }
   void localfunct2() // OK
   {  }
}
int anotherfunct()   // bad
{  }
```

プログラム 10.42: 関数ファイル funct.chf の最上位にある複数の関数

セクション 6.4の説明にあるように、関数ファイルで定義された関数を、関数またはブロックスコー

プで静的変数の識別子の初期化子として使用することはできません。

10.12 汎用関数

汎用関数は、システムに組み込みの関数です。Ch の汎用関数のリストについては、セクション 2.2を参照してください。ほとんどの汎用関数はポリモーフィックです。**sin()** などの汎用関数を明示的に呼び出すと、ユーザーが関数を再定義した場合でも、システムに組み込みの関数が使用されます。この場合、ユーザー定義の関数は無視されます。たとえば、sin(x) の関数呼び出しでは、組み込みのシステム関数を使用するため、引数 x を関数 sin() で有効な任意のデータ型とすることができます。

ただし、汎用関数 **alias()**、**dlrunfun()**、**elementtype()**、**polar()**、**max()**、**min()**、および **transpose()** に対応する標準の C 関数はありません。これらの汎用関数は、ユーザーによる再定義はできません。関数 polar() を除き、汎用関数を再定義すると、警告メッセージが表示されます。

汎用関数の名前が関数へのポインタとして割り当てられる場合は、標準の C 関数が使用されます。たとえば、次のコード例ではシンボル `sin` が使用されています。

```
#include <math.h>
double func1( double (*fp)(double), double x) {
  return fp(x);
}
int main() {
  double (*fp)(double) = sin;
  fp = sin;
  double val;
  val = fp(10.0);  // same as val = sin(10.0);
  func1(fp, 10);
  func1(sin, 10);
}
```

次のプロトタイプを含む標準の C 関数が使用されます。

```
double sin(double);
```

[5]ユーザーは、非関数型の識別子として汎用関数の名前を使用できます。たとえば、汎用関数 max、min、および exp の名前は、次のようにスカラ変数として宣言されます。

```
double max;
void func2() {
    int min, exp;
    ...
}
```

汎用関数は、ユーザーのホームディレクトリにあるシステム起動ファイル (Unix でｇは.chrc、Windows では_chrc) で使用できます。

[5]訳注：混乱を避けるためにおすすめできません。

第11章 参照型

本章では、 Ch に現在実装されている参照の言語的な特徴について説明します。一般に、手続き型のコンピュータプログラミング言語で記述されたプログラムは、一連の関数と、それに続く多数のプログラミングステートメントから成り立っています。関数を使用すると、大規模なコンピューティングタスクを小さいタスクに細分化できます。そのため、ゼロから始めるのではなく、他のユーザーが実行済みの作業を基にしてアプリケーションプログラムを開発できます。プログラミング言語にとって、関数の性能とユーザーが使いやすいインタフェースは重要な要素です。他のユーザーが開発した関数内部の詳細を知る必要はないかもしれません。しかし、関数を有効に利用するには、関数の引数や戻り値を介した関数とのインタフェース方法を理解する必要があります。一般に、**値呼び出し**または**参照呼び出し**の 2 つの方法のいずれかで、引数を関数に渡すことができます。値呼び出しのモデルでは、関数を呼び出すと、呼び出された関数のローカル仮パラメータに実パラメータ値がコピーされます。仮パラメータを lvalue (割り当てステートメントの左側に指定できるオブジェクト) として使用している場合は、パラメータのローカルコピーだけが変更されます。呼び出された関数で呼び出し元関数の実パラメータを変更する場合は、呼び出された関数にパラメータのアドレスを明示的に渡す必要があります。一方、参照呼び出しの方法では、引数のアドレスが関数の仮パラメータにコピーされます。関数内部では、そのアドレスを使用して、呼び出し元関数で使用されている実引数にアクセスします。このため、仮パラメータを lvalue として使用している場合、そのパラメータは関数呼び出しに使用する変数に影響を与えることになります。

FORTRAN では参照呼び出しモデルを使用するのに対して、C は値呼び出しを使用します。FORTRAN は最も古いコンピュータプログラミング言語の 1 つであり、現在でも科学スーパーコンピューティング用の主要な言語です。優れた FORTRAN プログラムが数多く存在します。FORTRAN のサブルーチンまたは関数を C の関数に移植する場合、通常、サブルーチンの仮引数は C の関数ではポインタ型の引数として扱われます。そのため、サブルーチンの内部にある引数のすべての変数を修正することが必要になり、元のアルゴリズムの明瞭性とコードの可読性が低下する可能性があります。これは、以前に FORTRAN を経験して C を初めて使用するユーザーが混乱する点でもあります。Ch は、C のスーパーセットとして設計されていますが、FORTRAN 77 のすべてのプログラミング機能を網羅しています。FORTRAN と C の差異を補い、FORTRAN コードの Ch への移植を容易にするために、Ch では、複素数型や形状引継ぎ配列などの多くのプログラミング機能が設計および実装されています。参照は、FORTRAN のサブルーチンと関数を Ch の関数に移植する作業をさらに簡素化するために、Ch に追加されたものです。

C への参照の追加は、これが初めてではありません。C++には参照型があります。 C++ の参照は、主にユーザー定義型に対する処理を指定する際に使用します。Ch の参照は、FORTRAN コードの Ch への移植を容易にし、科学プログラミングや未経験のユーザーに対する Ch の適合性を高めるだけでなく、ポインタが制限されているセーフ Ch プログラムで関数に引数を渡すためにも不可欠です。Ch の参照は、C、 C++ 、および FORTRAN の思想で設計され、実装されています。C++ および FORTRAN

での参照の言語的な特徴を科学プログラミング用に拡張しました。Ch では、基本データ型の変数とポインタ型の変数の両方を参照として使用できます。また、参照によって、データ型の異なる変数を関数の引数に渡すことができます。さらに、入れ子にされた関数や再帰的に入れ子にされた関数の引数およびローカル変数として、参照を使用することも可能です。

11.1 ステートメント内の参照

Ch の参照は、C++の場合と同様にオブジェクトの別名です。次の宣言ステートメント

```
int i, &j = i;
```

は、変数 j が int データ型の i への参照であることを示します。つまり、j は i の別名です。宣言された変数と参照される変数が同じデータ型である場合は、それらを相互参照と見なすことができます。したがって、上記の例では、i は j への参照であると言うこともできます。変数 i と j の両方が、システム内部の同じメモリ領域を共有します。同じ型の 2 つの変数に対してリンケージを確立すると、2 つの変数を同じ変数として使用できます。次に例を示します。

```
int i, &j = i;
i++;              // the same as 'j++'
```

C++ では、基本データ型の単純変数のみを参照として扱うことができます。Ch では、基本データ型の単純変数だけでなく、ポインタ型の変数も参照として宣言できます。次に例を示します。

```
int i, *p1 = &i, **p2 = &p1;
int &*pp1 = p1, &**pp2 = p2;
```

ここで、pp1 は int へのポインタである p1 への参照で、pp2 は int へのポインタであるポインタ p2 への参照です。

参照は宣言の段階で初期化する必要があります。参照関係はいったん確立すると変更できません。たとえば、次のコードは、参照の変数 j と p が初期化されていないため構文エラーとなります。

```
int &j;                 // ERROR: reference not initialized
int &*p;                // ERROR: reference not initialized
```

複数の変数がメモリ上の同じ場所を参照できます。次に例を示します。

```
int i, &j = i, &k = i, &l = k;
int &m = i;
```

ここでは、変数 i、j、k、l、および m が互いを参照しています。1 つの変数を変更すると、他のすべての変数に影響します。

別名の使用を避けて実装を簡素化するために、現在の Ch の実装では相互参照が可能なのは単純変数だけです。初期化されて参照となる rvalue が単純変数でない場合、参照は単純変数として扱われ、初期化は単純変数に対する初期化として処理されます。たとえば、次の宣言の参照はすべて、システムでは実質的に単純変数として処理されます。

```
int a[10];
float f, *fp = &f;
complex z;
int &i = 6;                      // int i = 6;
int &j =6+a[1];                  // int j = 6+a[1];
int &*pp = &i+6;                 // int *pp = &i+6;
float &f1 = real(complex(1,2)); // float f1 = real(complex(1,2));
float &f2 = real(z);             // float f2 = real(z);
float &f3 = a[1];                // float f3 = a[1];
float &f4 = *fp;                 // float f4 = *fp;
float &*p = &f;                  // float *p = &f;
f = 5;                           // the same as *fp = 5 or *p = 5;
```

上記の例では、real(z)、a[1] および*ptr は lvalue であり単純変数ではありません。したがって参照にはできません。C に準拠した処理方法では、ポインタ p は変数 f のアドレスを指す点に注意してください。

また、データ型に互換性がある限り、データ型の異なる変数を参照と見なすことができます。次にコード例を示します。

```
int i = 30;
double &d = i;
printf("d = %lf \n", d);   // output: d = 30.000000
```

double データ型の変数 d は、i という int への参照です。変数 i と d は両方とも、4 バイトを占有する int の同じメモリ領域を参照します。変数 d のデータ型が double であるため、**abs**(d) および d+3 の結果は double です。d を式で使用した場合、処理の実行前に、int の値が暗黙的に double に変換されます。同様に、割り当てステートメントで d を lvalue として使用すると、rvalue の結果は int にキャストされた後、4 バイトのみを保持するメモリに割り当てられます。したがって、値が整数値 [INT_MIN, INT_MAX] の範囲内にない場合、暗黙的なデータ変換によって情報が失われる場合があります。一方、int 型の変数が double 型の変数への参照である場合は、double 型変数の小数部を除くすべての情報が保持されます。次に例を示します。

```
double d = 3.6;
int &i = d;
printf("i = %i \n", i);   // output: i = 3
i = 7;                    // i = 7; d = 7.000000
d = 5.2;                  // i = 5; d = 5.2
```

このコードでは、変数 i と d の両方が 8 バイトの double データの同じメモリ領域を共有します。互換性のないデータ型の変数は参照にできません。次に例を示します。

```
int i, *p = &i;
int &*ptr = i;            // data types of ptr and i are incompatible
int &j = p;               // data types of j and p are incompatible
```

参照のリンケージは、異なるレキシカルレベルの変数に適用することもできます。以下のサンプルコードに示すように、下位のレキシカルレベルの変数を宣言して初期化し、より上位のレキシカルレベルで定義された変数を参照することができます。

```
int i = 8;
void funct()
{
  int &j = i, &k =i;          // j = 8; k = 8;
  printf("j = %d, ", j);      // output: j = 8,
  j =90;
}
funct();                      // get output: j = 8
printf("i = %d \n", i);       // output: i = 90
```

このコードでは、変数 j と k の両方が、変数 i と同じメモリ領域を共有します。上記のプログラムの出力は次のとおりです。
j = 8, i = 90

11.2　関数の引数の参照渡し

C では、関数が呼び出されると、呼び出し元関数の実引数が呼び出された関数の引数に値で渡されます。実パラメータの値は、呼び出された関数のローカル仮パラメータにコピーされます。仮パラメータが lvalue として使用されている場合、パラメータのローカルコピーだけが変更されます。したがって、次の関数 `swap()` は、x と y が値渡しされるため正しく動作しません。

```
int a = 5, b = 6;
void swap(int x, y)
{
  int temp;
  temp = x; x = y; y = temp;
}
swap(a, b);                     // fails to swap a and b
```

呼び出された関数で呼び出し元関数の実パラメータを変更する場合、C では、呼び出された関数にパラメータのアドレスを明示的に渡す必要があります。正しい関数 `swap()` の一例としては、次のコードに示すように、ポインタを使用して呼び出し元関数の変数のアドレスを呼び出された関数に渡します。

```
int a = 5, b = 6;
void swap(int *x, *y)
{
  int temp;
```

```
  temp = *x; *x = *y; *y = temp;
}
swap(&a, &b);                     // a = 6; b = 5;
```

ここでは、間接化処理を使用して、呼び出し元関数の変数の値を変更しています。一方、FORTRAN のような参照呼び出しの方法では、引数のアドレスが関数の仮パラメータにコピーされます。関数の内部では、このアドレスを使用して、呼び出し元関数で使用されている実引数にアクセスします。このため、仮パラメータが lvalue として使用されている場合、パラメータは関数呼び出しに使用する変数に影響を与えることになります。Ch で参照を関数の引数として使用する場合、関数は参照によって呼び出されます。次のように Ch の参照を使用して関数 `swap()` を実装できます。

```
int a = 5, b = 6;
void swap(int &x, &y)
{
  int temp;
  temp = x; x = y; y = temp;
}
swap(a, b);                     // a = 6; b = 5;
```

このコードではポインタの間接化を使用していません。

C では、関数の引数を使用してポインタ型の変数の値を変更する必要がある場合は、ポインタへのポインタ (すなわち、二重ポインタ) を関数に渡す必要があります。Ch では、プログラム 11.1に示すように、単純変数だけでなくポインタも参照渡しができます。

```
int i =5, *p1 = &i, **p2;
int &*pp1 = p1, &**pp2 = p2;            // pp1 = &i

p2 = malloc(5*sizeof(int));
void funct1(int& *p)
{
  p = malloc(9);
  printf("In funct2() p = %p \n", p);
}
printf("Before funct1() pp1 = %p \n", pp1);
funct1(pp1);
printf("After funct1() pp1 = %p \n", pp1);

void funct2(int& **pp)
{
  pp++;
  printf("In funct2() pp = %p \n", pp);

}
printf("Before funct2() pp2 = %p \n", pp2);
funct2(pp2);
printf("After funct2() pp2 = %p \n", pp2);
```

プログラム 11.1: Ch でのポインタへの参照

プログラム 11.1では、関数 `funct1(pp1)` が呼び出される前の時点で、ポインタ変数 `pp1` が変数 i のメモリ上の場所を指しています。ポインタ `pp1` は、`funct1(pp1)` の関数呼び出しによって新たに割り当てられた 9 バイトのメモリ領域を指しています。これは関数の仮引数 `p` によって実現されます。同様に、二重ポインタ `pp2` の変数は、関数 `funct2()` の仮引数 `pp` に渡されます。ポインタは関数内部のアドレス演算によって int 用の領域である 4 バイト単位で増分されます。プログラム 11.1の出力は次のとおりです。

Before funct1() pp1 = 11b578
In funct2() p = 11ea38
After funct1() pp1 = 11ea38
Before funct2() pp2 = 11e930
In funct2() pp = 11e934
After funct2() pp2 = 11e934

Ch では、関数 `free(ptr)` はポインタ `ptr` が指すメモリの割り当てを解除し、ポインタ `ptr` を NULL にリセットします。C では、`ptr` が指すメモリの割り当てを解除するとき、`ptr` は NULL に設定されません。この宙に浮いたぶら下がりメモリは、C プログラムのデバッグを非常に困難にします。その理由は、割り当てを解除されたメモリがプログラムの他の部分によって再度要求されるまで、問題が表面化しないためです。C では関数 **free**() が外部関数として実装されているため、`free(ptr)` の関数呼び出しによってポインタ `ptr` を解放するときに `ptr` を NULL に設定する方法はありません。しかし、C に参照が追加されると、Ch のプログラム 11.2に示すように、ポインタ `ptr` が指すメモリを解放する関数 deallocate(ptr) を実装して、`ptr` を NULL にリセットすることが可能になります。

```
void deallocate(void &* ptr)
{
  free(ptr);
  ptr = NULL;
}
void *p;
p = malloc(10);
deallocate(p);  // free memory and reset p to NULL
```

プログラム 11.2: 関数 `deallocate()` でメモリを解放してポインタを NULL にリセットする処理

プログラム 11.2では、関数 **free**() は、関数呼び出しの完了時に引数に NULL を設定しない C の関数であると想定しています。

Ch では、1 つの変数の同じメモリ領域を、関数の引数に含まれる異なる参照に渡すことができます。たとえば、プログラム 11.3では、関数 `funct()` の引数である `r1` と `r2` の両方が変数 `i` の同じメモリ領域を使用するのに対し、`r3` は、関数が `funct(i, i, i)` によって呼び出される際に独自のローカルメモリを所有します。

```
int i;
void funct(int &r1, &r2, r3)
{
  r1 = 3; r2++; r3++;
  printf("r1 = %d\n", r1);  // output: r1 = 4
}
funct(i, i, i);
printf("i = %d\n", i);      // output: i = 4
```

プログラム 11.3: 同じ変数を別々の参照に渡す処理

FORTRAN では、関数の引数がサブルーチン内で lvalue として使用されるとき、呼び出し元関数の実引数は変数でなければなりません。FORTRAN とは異なり、 Ch の参照変数は、実引数が lvalue ではない場合でも関数内部の lvalue として使用することができます。仮定義の参照に対応する関数の実引数が単純変数でない場合、引数は値で渡されます。

プログラム 11.4では、参照 `r1` および `r2` を関数 `funct()` の lvalue として使用しています。

```
int i =50, j=0, &k = j;
int funct(int &r1, &r2)
{
  r1 += 100;
  r2 += r1+2;
  printf("r1 = %d, r2 = %d\n", r1, r2);
  return r1+r2;
}
funct(i+1,3);                // output: r1 = 151, r2 = 156
funct(i,k);                  // output: r1 = 150, r2 = 152
funct(abs(-3), funct(1,2)); // output: r1 = 101, r2 = 105
                             // output: r1 = 103, r2 = 311
printf(i, " ", j, "\n");    // output: 150 152
```

プログラム 11.4: 実引数が式である場合に参照を lvalue として使用する例

`funct(i+1,3)` の関数呼び出しは、式 `i+1` および `3` をそれぞれ参照 `r1` および `r2` に渡します。`funct(i, k)` の関数呼び出しでは、変数 j の代わりに参照 k が参照 r2 に渡されることに注意してください。`funct(abs(-3), funct(1,2))` の関数呼び出しでは、ユーザー定義関数によってシステム組み込み関数 `abs()` とユーザー定義関数自身の実行結果への参照が関数の引数として使用されています。

関数の引数への参照を関数内部のローカル変数にすることができます。たとえば、プログラム 11.5では、ローカル変数 `r` は関数の整数型引数 `j` への参照です。

```
int i=5;
void funct(int j)
{
  int &r = j;
  printf("r = %d ", r);
  r++;
  printf("j = %d ", j);
}
funct(i);
printf("i = %d\n", i);
```

プログラム 11.5: ローカル変数が関数の引数への参照である例

プログラム 11.5の出力結果は次のとおりです。

r = 5 j = 6 i = 5

11.3 データ型の異なる変数を同じ参照に渡す方法

宣言ステートメントでデータ型の異なる変数を使用して参照を初期化するのと同様に、データ型の異なる変数を関数の引数内の参照に渡すこともできます。この場合のインタフェース規則は、セクション 11.1に示した規則とほぼ同じです。

たとえば、プログラム 11.6では、関数 `funct()` 内部の変数 `r1` および `r2` はそれぞれ、`funct(f, i)` の関数呼び出しにおける変数 `f` および `i` と同じメモリ領域を共有します。

```
float f = 90;
int i = -4;
void funct(int &r1, complex & r2)
{
   printf("r1 = %d ", r1++);
   printf("sqrt(r2) = %.3f \n", sqrt(r2--));
}
funct(f, i);  // output: r1 = 90 sqrt(r2) = complex(0.000,2.000)
printf(f," ", i,"\n");  // output: 91.000 -5
```

プログラム 11.6: データ型の異なる参照に実引数を渡す処理

引数のインタフェースは、最初に変数 `f` および `i` の値が int 型および複素数型に変換された場合と同様に扱われ、次に変数 `r1` および `r2` にそれぞれコピーされます。プログラム実行のフローが関数から出た時点で、`r1` と `r2` の結果はそれぞれ変数 `f` と `i` の値に変換されます。C にはデータ型が定義と異なる変数を渡して正しい結果を得る方法はないので、関数の引数で異なるデータ型のインタフェースを提供できることは重要な機能拡張です。

Ch では int 型データの平方根は float 型を返すため、**sqrt**(-4) は NaN (非数) である点に注意してください。プログラム 11.6とプログラム 11.7は異なります。

```
float f = 90;
int i = -4;
void funct(int *r1, complex * r2)
{
   printf("r1 = %d ", (*r1)++);
   printf("sqrt(r2) = %.3f \n", sqrt((*r2)--));
}
funct(&f, &i); // output: r1 = 1119092736 sqrt(r2) = complex(NaN,NaN)
printf(f," ", i,"\n");  // output: 90.000 -4
```

プログラム 11.7: 関数のポインタ型引数に異なる型のポインタを渡す処理

プログラム 11.7では、変数 `f` のアドレスを `funct(&f,&i)` の関数呼び出しで引数 `r1` に渡すとき、アドレスだけが渡されます。関数内部では、float のメモリマップが int 用のメモリマップとして使用されます。これはユーザーの意図に反する場合もあります。同様に、関数呼び出しで int 型変数 `i` のメモリ上の場所が複素数型の変数 `r2` へのポインタに渡されます。

関数呼び出しの参照の実引数が単純変数でない場合、関数内部の参照が単純変数として扱われます。関数呼び出しの参照の実引数が単純変数でなく、引数のデータ型が定義と異なる場合、結果は定義のデータ型に変換されてから参照の変数に割り当てられます。次に例を示します。

```
void funct(float &r)
{
   printf("r = %.3f \n", r);
}
funct(90.0);                  // output: r = 90.000
funct(90);                    // output: r = 90.000
funct(complex(90,0));         // output: r = 90.000
funct(complex(1,2));          // output: r = NaN
```

虚部がゼロに等しくない複素数から変換した実数は NaN (非数) である点に注意してください。

関数の参照引数の仮定義とは異なるデータ型を持つ単純変数を関数の引数に渡すと、Ch プログラムでは値を式の lvalue またはオペランドとして使用するときに値が調整されます。ただし、関数の引数がポインタデータ型への参照である場合、システムは関数に渡されたオブジェクトを参照用に宣言されたポインタ型として扱います。つまり、オブジェクト用のメモリだけが使用され、呼び出し元関数内の元のポインタ型は関数内部では無視されます。

たとえば、プログラム 11.8では、変数 `p1` および `p2` はそれぞれ int 型および float 型へのポインタです。

```
void deallocate(void &* ptr)
{
  free(ptr);
  ptr = NULL;
}
void getmem(void &* ptr, int i)
{
  ptr = malloc(i);
}
void funct(int &*iptr, float &*fptr)
{
  printf("before: *iptr = %d *fptr = %f \n", *iptr, *fptr);
  *iptr = 90;
  *fptr = 90;
  printf("after: *iptr = %d *fptr = %f \n", *iptr, *fptr);
}
int *p1;
float *p2;
getmem(p1, sizeof(int)); // p1 = malloc(sizeof(int))
p2 = malloc(sizeof(float));
*p1 = 4; *p2 = 5;
funct(p1, p2);
printf("*p1 = %d *p2 = %f \n", *p1, *p2);
*p1 = 4; *p2 = 5;
funct(p2, p1);
printf("*p1 = %d *p2 = %f \n", *p1, *p2);
deallocate(p1);  // free memory and reset p1 to NULL
deallocate(p2);  // free memory and reset p2 to NULL
```

プログラム 11.8: 異なる型のポインタを関数内のポインタへの参照の引数に渡す処理

これらの変数を、`funct(p1, p2)` および `funct(p2, p1)` の関数呼び出しによって

関数 `funct(int &*, float &*)` に渡しています。異なるデータ型を指定してポインタ型の参照が渡された場合、`funct(p2, p1)` の関数呼び出しで正しい結果を導くための間接化処理は調整されません。プログラム 11.8の出力結果は次のとおりです。

*before: *iptr = 4 *fptr = 5.000000*
*after: *iptr = 90 *fptr = 90.000000*
**p1 = 90 *p2 = 90.000000*
*before: *iptr = 1084227584 *fptr = 0.000000*
*after: *iptr = 90 *fptr = 90.000000*
**p1 = 1119092736 *p2 = 0.000000*

ただし、間接化処理を使用しない場合、ポインタへの参照は関数内部の通常のポインタとして使用できます。たとえば、ポインタ `p1` は関数 `getmem(p1, sizeof(int))` によってメモリに割り当てられ、`deallocate(p1)` の関数呼び出しによってメモリから解放されて NULL にリセットされます。変数 `p1` は int へのポインタですが、関数 `getmem()` および `deallocate()` の対応する引数のデータ型は void へのポインタである点に注意してください。

第12章　汎用数学関数を使用する科学計算

Ch は、科学技術とシステムの両方のプログラミングのために設計された言語です。本章では、 Ch 言語を科学計算の観点から説明します。2 進数の浮動小数点演算用に策定された ANSI/IEEE 754 規格 [11] は、実数に対する一貫した浮動小数点演算を確立する上での重要な指標です。プログラマが簡単に IEEE 754 規格の有用性を活かせるように、Ch では、Inf、−Inf、および NaN などの**メタ数値**と呼ばれる浮動小数点数を導入しています。

これらのメタ数値は、プログラマにとってわかりやすいものです。Ch では、符号付きゼロの +0.0 および −0.0 は、符号付き無限小の数量である 0_+ および 0_- と同じように正しく動作します。また、シンボル Inf と −Inf は、それぞれ数学的に無限大を表す ∞ と $-\infty$ に相当します。Inf や NaN などのシンボルを採用したアプリケーションは、いくつかのソフトウェアパッケージにも見られますが、これらの特殊な数字の扱い方には欠陥があります。たとえば、ソフトウェアパッケージの Mathematica には ComplexIninity があり、MATLAB には Inf と NaN があります。Mathematica では、複素数の無限大と実数の無限大は区別されず、−0.0 と 0.0 の違いも区別されていません。したがって、この章で定義されている多くの演算は Mathematica で行うことはできません。また、MATLAB には複素数の無限大がありません。

数学関数のないコンピュータ言語は、科学計算や他の多くのアプリケーションには適していません。C 言語は小規模な言語であり、内部に数学関数を備えていません。数学関数は、数学関数の標準ライブラリで提供されます。C には内部に数学関数がないため、K&R C の算術演算と同様に、標準の数学関数からの戻り値は、入力された引数のデータ型にかかわらず、double 型の浮動小数点数となります。C の実装方法によっては、double 型以外の引数を入力すると、数学関数から警告なく誤った結果が返されることがあります。数値計算を重要視するプログラマは、コンピュータ言語の算術演算において、データ型が float から double へ暗黙的に変換されることに寛容ではありません。

ただし、プログラマは通常、厳密に型指定された数学関数の実装は容認します。数学関数からの戻り値のデータ型を変えたい場合は、名前の異なる新しい関数が必要になります。たとえば、**sin**(1) の関数呼び出しは、C では正しく見えます。実際、C プログラムの多くは通常どおりこの演算を実行しますが、入力された整数のデータ型が **sin**() 関数の期待するものではないため、おそらくは誤った結果を返します。別の例として、C では関数 **abs**() は int 型の絶対値を返し、**fabs**() は double 型の数値を返します。したがって、float 型の絶対値が返されるようにするには、新しい関数の作成が必要になります。その結果、さまざまな関数のために、プログラマは仲間内でしか理解できない名前をたくさん覚えていなければなりません。このような問題を解決するために、Ch では汎用関数を使用します。

Ch の外部関数は、C と同じように作成できます。Ch の内部には、使用頻度の高い数学関数が組み込まれています。Ch の数学関数は、データ型の異なる引数をうまく扱うことができます。関数の戻り値のデータ型は、入力された引数のデータ型によって決まります。これは**ポリモーフィズム**と呼ばれます。算術演算子と同じく、 Ch で使用頻度の高い汎用数学関数はポリモーフィックです。たとえば、ポリモーフィックな関数 **abs**() の場合、入力された引数のデータ型が int であれば、int 型の絶対値を返します。**abs**() に float 型または double 型の引数が入力された場合は、それに応じて、float また

は double のデータ型が出力結果として返されます。複素数が入力された場合、関数 **abs**() から返される結果は、入力された複素数の絶対値の値を持つ float 型です。

同様に、引数のデータ型が float 型よりも下位か、または float 型に等しい場合、**sin**() は正しく float 型の結果を返します。関数 **sin**() は、入力された引数が double 型または complex 型の場合、それに応じて、double 型または complex 型の結果を返すことができます。また、 Ch 自体に入出力関数が組み込まれているため、異なるデータ型は Ch の内部で調整されます。

たとえば、C の `printf("%f", x)` は、x が float 型であれば x を出力できます。しかし、x をプログラム内で int 型に変更した場合、それに従って出力ステートメントも `printf("%d", x)` に変える必要があります。このため、変数のデータ型の宣言を変更すると、プログラムの他の多くの部分を変更する必要が出てきます。Ch の **printf**(x) および **printf**(sin(x)) コマンドは柔軟性があり、x はさまざまなデータ型 (char、int、float、double、または complex) を扱うことができます。

移植性を持たせるため、C のヘッダーファイル `math.h` で定義されているすべての数学関数は、ポリモーフィックな関数として Ch に実装されています。関数の戻り値のデータ型は、入力された引数のデータ型によって決まります。これにより、科学技術の数値計算は大幅に簡素化されます。本章に記載されている Ch のこれらの汎用数学関数の名前は、C のヘッダーファイル `math.h` に基づきます。Ch の数学関数は、C と互換性があります。これらの関数の引数のデータ型が C の数学関数のデータ型に対応していれば、ユーザーの観点からは、C と Ch の関数の間には違いがまったくありません。

12.1 汎用数学関数の概要

このセクションでは、 Ch の汎用数学関数について説明します。特に、メタ数値を扱う関数の入力と出力について重点的に説明します。表 12.1〜12.4に、メタ数値を扱う数学関数の結果を示します。表 12.1〜12.4では、特に説明がない限り、x、x_1、x_2 は $0 < x, x_1, x_2 < \infty$ の範囲内の実数です。k は整数値を表します。`pi` の値は、浮動小数点数での無理数 π の有限表現です。

関数の戻り値のデータ型は、入力された引数のデータ型に応じて float または double になります。表 12.1では、`x` のデータ型の順位が float 型と同じか、それより下位の場合、戻り値のデータ型は float になります。x が double 型の場合は、戻り値のデータ型は double です。表 12.1の関数の引数 `x` が NaN の場合、関数は NaN を返します。

表 12.2〜12.4では、入力された 2 つの引数のどちらかが float 型または double 型の場合、戻り値のデータ型は、入力された 2 つの引数のうち順位が上位のデータ型と同じになります。それ以外の場合は、既定により、float が戻り値のデータ型になります。

このセクションで定義されている関数は、関数 **abs**() と関数 **pow**() を除き、float 型または double 型の値を返します。関数 **abs**() の引数が整数値の場合、戻り値のデータ型は int になります。関数 **fabs**() の引数が int と float を含む単純なデータ型の場合、戻り値のデータ型は double になります。関数 **pow**() の引数が整数値の場合、戻り値のデータ型は double です。たとえば、**pow**(2,16) は double 型の値 65536 を返します。

絶対値関数 **abs**(x) は、整数または浮動小数点数の絶対値を計算します。負の無限大 $-\infty$ の絶対値は、正の無限大 ∞ です。

関数 **sqrt**(x) は、非負数 `x` の平方根を計算します。`x` が負数の場合、結果は NaN となります。ただし、**sqrt**(-0.0) の場合は、IEEE 754 規格に準じて -0.0 です。無限大平方根 **sqrt**(∞) は、無限大になります。

表 12.1: 実数関数による ±0.0、±∞、および NaN の結果

関数	x の値と結果						
	−Inf	−x1	−0.0	0.0	x2	Inf	NaN
abs(x)	Inf	x_1	0.0	0.0	x_2	Inf	NaN
fabs(x)	Inf	x_1	0.0	0.0	x_2	Inf	NaN
sqrt(x)	NaN	NaN	−0.0	0.0	sqrt(x)	Inf	NaN
exp(x)	0.0	e^{-x_1}	1.0	1.0	e^{x_2}	Inf	NaN
log(x)	NaN	NaN	−Inf	−Inf	log(x_2)	Inf	NaN
log10(x)	NaN	NaN	−Inf	−Inf	$\log_{10}(x_2)$	Inf	NaN
sin(x)	NaN	−sin(x_1)	−0.0	0.0	sin(x_2)	NaN	NaN
cos(x)	NaN	cos(x_1)	1.0	1.0	cos(x_2)	NaN	NaN
tan(x)	NaN	−tan(x_1)	−0.0	0.0	tan(x_2)	NaN	NaN
	注意: tan($\pm\pi/2 + 2 * k * \pi$) = ±Inf						
asin(x)	NaN	−asin(x_1)	−0.0	0.0	asin(x_2)	NaN	NaN
	注意: asin(x) = NaN, ここで $\|x\| > 1.0$						
acos(x)	NaN	acos(x_1)	pi/2	pi/2	acos(x_2)	NaN	NaN
	注意: acos(x) = NaN, ここで $\|x\| > 1.0$						
atan(x)	−pi/2	−atan(x_1)	−0.0	0.0	atan(x_2)	pi/2	NaN
sinh(x)	−Inf	−sinh(x_1)	−0.0	0.0	sinh(x_2)	Inf	NaN
cosh(x)	Inf	cosh(x_1)	1.0	1.0	cosh(x_2)	Inf	NaN
tanh(x)	−1.0	−tanh(x_1)	−0.0	0.0	tanh(x_2)	1.0	NaN
asinh(x)	−Inf	−asinh(x_1)	−0.0	0.0	asinh(x_2)	Inf	NaN
acosh(x)	NaN	NaN	NaN	NaN	acosh(x_2)	Inf	NaN
	注意: acosh(x) = NaN, ここで $x < 1.0$; acosh(1.0) = 0.0						
atanh(x)	NaN	−atanh(x_1)	−0.0	0.0	atanh(x_2)	NaN	NaN
	注意: atanh(x) = NaN, ここで $\|x\| > 1.0$; atanh(±1.0) = ±Inf						
ceil(x)	−Inf	ceil($-x_1$)	−0.0	0.0	ceil(x_2)	Inf	NaN
floor(x)	−Inf	floor($-x_1$)	−0.0	0.0	floor(x_2)	Inf	NaN
ldexp(x, k)	−Inf	ldexp($-x_1, k$)	−0.0	0.0	ldexp(x_2, k)	Inf	NaN
modf(x, &y)	−0.0	modf($-x_1, \&y$)	−0.0	0.0	modf($x_2, \&y$)	0.0	NaN
y	−Inf	y	−0.0	0.0	y	Inf	NaN
frexp(x, &k)	−Inf	frexp($-x_1, \&k$)	−0.0	0.0	frexp($x_2, \&k$)	Inf	NaN
k	0	k	0	0	k	0	0

表 12.2: 関数 pow(y, x) による ±0.0、±∞、および NaN の結果

pow(y, x)											
y の値	x の値										
	−Inf	−x1	$-2k-1$	−2k	−0.0	0.0	2k	$2k+1$	x2	Inf	NaN
Inf	0.0	0.0	0.0	0.0	1.0	1.0	Inf	Inf	Inf	Inf	NaN
y2 > 1	0.0	${y_2}^{-x_1}$	${y_2}^{-2k-1}$	${y_2}^{-2k}$	1.0	1.0	y_2^{2k}	y_2^{2k+1}	$y_2^{x_2}$	Inf	NaN
1.0	NaN	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	NaN	NaN
0 < y2 < 1	Inf	${y_2}^{-x_1}$	${y_2}^{-2k-1}$	${y_2}^{-2k}$	1.0	1.0	y_2^{2k}	y_2^{2k+1}	$y_2^{x_2}$	0.0	NaN
0.0	Inf	Inf	Inf	Inf	1.0	1.0	0.0	0.0	0.0	0.0	NaN
−0.0	Inf	Inf	−Inf	Inf	1.0	1.0	0.0	−0.0	0.0	0.0	NaN
−y1	NaN	NaN	$-{y_1}^{-2k-1}$	${y_1}^{-2k}$	1.0	1.0	y_1^{2k}	$-y_1^{2k+1}$	NaN	NaN	NaN
−Inf	NaN	NaN	−0.0	0.0	1.0	1.0	Inf	−Inf	NaN	NaN	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

表 12.3: 関数 atan2(y, x) による ±0.0、±∞、および NaN の結果

atan2(y, x)							
y の値	x の値						
	−Inf	−x1	−0.0	0.0	x2	Inf	NaN
Inf	3∗pi/4	pi/2	pi/2	pi/2	pi/2	pi/4	NaN
y2	pi	atan2($y_2, -x_1$)	pi/2	pi/2	atan2(y_2, x_2)	0.0	NaN
0.0	pi	pi	pi	0.0	0.0	0.0	NaN
−0.0	−pi	−pi	−3∗pi/4	−pi/2	−0.0	−0.0	NaN
−y1	−pi	atan2($-y_1, -x_1$)	−pi/2	−pi/2	atan2($-y_1, x_2$)	−0.0	NaN
−Inf	−3∗pi/4	−pi/2	−pi/2	−pi/2	−pi/2	−pi/4	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

表 12.4: 関数 fmod(y, x) による ±0.0、±∞、および NaN の結果

fmod(y, x)							
y の値	x の値						
	−Inf	−x1	−0.0	0.0	x2	Inf	NaN
Inf	NaN	NaN	NaN	NaN	NaN	NaN	NaN
y2	y_2	fmod($y_2, -x_1$)	NaN	NaN	fmod(y_2, x_2)	y_2	NaN
0.0	0.0	0.0	NaN	NaN	0.0	0.0	NaN
−0.0	−0.0	−0.0	NaN	NaN	−0.0	−0.0	NaN
−y1	y_1	fmod($-y_1, -x_1$)	NaN	NaN	fmod($-y_1, x_2$)	$-y_1$	NaN
−Inf	NaN	NaN	NaN	NaN	NaN	NaN	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

関数 **exp**(x) は x の指数関数を計算します。$e^{-\infty} = 0.0$、$e^{\infty} = \infty$、および $e^{\pm 0.0} = 1.0$. という結果が成立します。

関数 **log**(x) は x の自然対数を計算します。x が負数の場合、結果は NaN です。この場合、-0.0 の値は 0.0 と同じと見なされます。$\mathbf{log}(\pm 0.0) = -\infty$ および $\mathbf{log}(\infty) = \infty$ という結果が成立します。`log10`(x) 関数は x の常用対数（10 を底とする対数）を計算します。x が負数の場合、結果は NaN です。関数 log() と同様、-0.0 の値は 0.0 と同じと見なされます。$\mathbf{log10}(\pm 0.0) = -\infty; \mathbf{log10}(\infty) = \infty.$ という結果が成立します。

三角関数 **sin**(x)、**cos**(x)、および **tan**(x) は、それぞれラジアンで測定された x のサイン、コサイン、およびタンジェントを計算します。サインとタンジェントは奇関数であるため、それぞれ、$\mathbf{sin}(\pm 0.0) = \pm 0.0$ および $\mathbf{tan}(\pm 0.0) = \pm 0.0$ となります。コサインは偶関数であるため、$\mathbf{cos}(\pm 0.0) = 1.0$ となります。引数の値が正の無限大または負の無限大の場合、これらの関数はすべて NaN を返します。理論的には、$\mathbf{tan}(\pm\pi/2 + 2 * k * \pi) = \pm\infty$ が正しいですが、実際には、無理数 π を float 型または double 型のデータで正確に表すことができないため、**tan**(x) 関数は $\pm\infty$ の無限大を返すことはありません。関数 **tan**() は、$\pi/2$、$\mathbf{tan}(\pi/2 - \varepsilon) = \infty$ および $\mathbf{tan}(\pi/2 + \varepsilon) = -\infty$ の場合 (ε は非常に小さい値を示す)、連続しません。浮動小数点数の有限精度と丸めエラーのため、誤った結果として $\pi/2$ の近似値を得る場合があります。

サインとタンジェントの奇関数の特性は、それらの逆関数である **asin**(x) と **atan**(x) にも反映されます。関数 **asin**(x) は x のアークサイン (逆正弦) の主値を計算します。x の値が $[-1.0, 1.0]$ の範囲内にある場合、関数 asin(x) は $[-\pi/2, \pi/2]$ ラジアンの範囲内の値を返します。x が $[-1.0, 1.0]$ の範囲外にある場合、アークサインは定義されず、**asin**(x) は NaN を返します。アークコサイン (逆余弦) の偶関数 **acos**(x) の入力値の範囲は **asin**(x) の場合と同じです。関数 **acos**(x) は x のアークコサインの主値を計算します。アークコサインの主値の範囲は $[0.0, \pi]$ ラジアンです。関数 **atan**(x) は x のアークタンジェント (逆正接) の主値を計算します。atan(x) 関数は、$[-\pi/2, \pi/2]$ ラジアンの範囲内の値を返します。$\mathbf{atan}(\pm\infty) = \pm\pi/2$ という結果が成立します。

三角関数 **sin**(x) および **tan**(x) と同様に、双曲線関数の **sinh**(x) と **tanh**(x) は奇関数です。関数 **sinh**(x) と関数 **tanh**(x) は、それぞれ、x の双曲線正弦と双曲線正接を計算します。偶関数 **cosh**(x) は x の双曲線余弦を計算します。$\mathbf{sinh}(\pm 0.0) = \pm 0.0$、$\mathbf{cosh}(\pm 0.0) = 1.0$、$\mathbf{tanh}(\pm 0.0) = \pm 0.0$、$\mathbf{sinh}(\pm\infty) = \pm\infty$、$\mathbf{cosh}(\pm\infty) = \infty$、$\mathbf{tanh}(\pm\infty) = \pm 1.0$、という結果が成立します。

逆双曲線関数は、C 規格では定義されていません。Ch では、逆双曲線正弦、逆双曲線余弦、および逆双曲線正接は、それぞれ **asinh**(x)、**acosh**(x)、および **atanh**(x) として定義されています。関数 **acosh**(x) は、引数が 1.0 未満の場合は定義されず、NaN が返されます。**acosh**(1.0) の場合は、正のゼロが返されます。関数 atanh(x) の有効範囲は $[-1.0, 1.0]$ です。$\mathbf{asinh}(\pm 0.0) = \pm 0.0$、$\mathbf{asinh}(\pm\infty) = \pm\infty$、$\mathbf{acosh}(\infty) = \infty$、$\mathbf{atanh}(\pm 0.0) = \pm 0.0$、$\mathbf{atanh}(\pm 1.0) = \pm\infty$、という結果が成立します。

関数 **ceil**(x) は、x の値より小さくない最小の整数値を計算します。**ceil**(x) と対になる関数は、x の値より大きくない最大の整数値を計算する **floor**(x) です。$\mathbf{ceil}(\pm 0.0) = \pm 0.0$、$\mathbf{floor}(\pm 0.0) = \pm 0.0$、$\mathbf{ceil}(\pm\infty) = \pm\infty$、$\mathbf{floor}(\pm\infty) = \pm\infty$、という結果が成立します。

関数 **ldexp**(x, k) は、浮動小数点数 x の値と 2 を k で累乗した値を掛け合わせます。$x * 2^k$ の戻り値は x の符号を保持します。

関数 **modf**(x, xptr) と関数 **frexp**(x, iptr) は、それぞれ 2 つの引数を受け取ります。最初の引数は入力データで、2 番目の引数は、関数呼び出しの結果の整数部分を格納するポインタです。関数 **modf**(x, xptr) は、引数 x を、それぞれ引数と同じ符号を持つ整数部と小数部に分けます。関数 **modf**() は小数部を返し、整数部は 2 番目の引数が参照するメモリに格納されます。2 つの引数の基本データ型は同

じでなければなりません。たとえば、最初の引数 x が float 型であるなら、2 番目の引数 `xptr` は float 型へのポインタでなければなりません。

最初の引数がメタ数値の場合、整数部はメタ数値と等しくなり、小数部は NaN の場合を除き、最初の引数の符号を持つゼロになります。**frexp**(x, iptr) 関数は、浮動小数点数を $x * 2^k$ という形式に、つまり正規化された小数部と 2 を累乗した整数部に分けます。**frexp**(x, iptr) 関数は正規化された小数部を返します。整数部は、int へのポインタである 2 番目の引数が指すメモリに格納されます。最初の引数がメタ数値の場合は、小数部はメタ数値と等しくなり、整数部はゼロになります。

数学関数 **pow**(y, x)、**atan2**(y,x)、および **fmod**(y,x) は、2 つの入力引数を受け取ります。表 12.2〜12.4にこれら 3 つの関数の結果を示します。関数 **pow**(y, x) は y の x 乗を計算し、y^x または $e^{x\log(y)}$ となります。x が負の値の場合は、表 7.6に示す定義された除算演算によって y^x は $1/y^{|x|}$ となります。y がゼロ未満で、x が整数値 (ゼロも含む) でない場合、関数は定義されません。-0.0 の値は、x の値が整数でない場合は、$\log(-0.0)$ の評価で 0.0 に等しいと見なされます。x が奇整数で y が負の値の場合、結果は負の値になります。y が正の値である場合、x が無限大ならば、結果は y の値に依存します。y が 1 未満の場合、y^∞ は 0.0 で、1.0^∞ は不定となり、y が 1 以上の場合、y^∞ は無限大になります。y が無限大で、x がゼロの場合、$(\pm\infty)^{\pm 0.0}$ は 1.0 です。

atan2(y, x) 関数は、両方の引数の符号を使用して、y/x のアークタンジェントの主値を計算し、$[-\pi, \pi]$ ラジアンの範囲内で戻り値を求めます。X-Y 平面上の点の座標 (x, y) を引数とした場合、関数 **atan2**(y, x) は、原点からその点までの半径の角度を計算します。正の数のオーバーフローは Inf で表します。負の数のオーバーフローは –Inf で表します。$\textbf{atan2}(\pm\text{Inf}, -\text{Inf}) = \pm 3\pi/4$、$\textbf{atan2}(\pm\text{Inf}, \text{Inf}) = \pm\pi/4$、$\textbf{atan2}(\pm\text{Inf}, \text{x}) = \pm\pi/2$、$\textbf{atan2}(\pm y, \text{Inf}) = \pm 0.0$、および $\textbf{atan2}(\pm y, -\text{Inf}) = \pm\pi$、という結果が成立します。

y と x の両方の値がゼロである場合、関数 **atan2**(y, x) は、これまでに説明したメタ数値の操作と一貫した結果を返します。–0.0 の値は、ゼロ未満の負の数と見なされます。したがって、これらの特殊な計算では、$\textbf{atan2}(0.0, -0.0) = \pi$、$\textbf{atan2}(0.0, 0.0) = 0.0$、$\textbf{atan2}(-0.0, -0.0) = -3\pi/4$、および $\textbf{atan2}(-0.0, 0) = -\pi/2$、という結果が成立します。

これは、**atan2**($-$Inf, $-$Inf) = $-3pi/4$ における ±Inf のメタ数値の処理と一貫性があります。Ch では、**atan2**(0.0, 0.0) の値が特別に定義されています。これらの結果は、4.3 Berkeley Software Delivery (SUN, 1990a) に準拠する SUN の C コンパイラによる結果とは異なります。4.3BSD に準拠した場合、これらの特殊なケースでは、$\textbf{atan2}(\pm 0.0, -0.0) = \pm 0.0$ および $\textbf{atan2}(\pm 0.0, 0.0) = \pm\pi$ という結果になり、これは、x 軸の ±0.0 値と y 軸の値が異なることを示します。

関数 **fmod**(y,x) は、y/x の浮動小数点数の剰余を計算します。関数 **fmod**(y,x) は、ある整数 i に対して $y - i * x$ の値を返します。x と同じ符号を持つ戻り値の絶対値は、x の絶対値より小さくなります。x がゼロの場合、関数は不定であり、NaN を返します。y が無限大である場合も、結果は不定です。x が無限大であり、かつ y が有限数である場合、結果は y と同じになります。

12.2 プログラミング例

12.2.1 浮動小数点数の極値の計算

マシンアーキテクチャごとに浮動小数点数の表現が異なるため、表現可能な最大の浮動小数点数値などの極値もさまざまになります。浮動小数点数値について同じ表現形式を持つ 2 台のマシンを使用

して各マシンで同じ演算 (2 つの値を加算するなど) を行っても、マシンの仕組みによっては値の丸めが正確に表現できずに、結果が異なる場合があります。

プログラム開発で数値計算を重要視するプログラマを支援するため、C 規格では、マシンに依存する整数値のみを扱う既存のヘッダーファイル limits.h と共に、ヘッダーファイル float.h が追加されています。このセクションでは、コンピュータの複雑なアーキテクチャを考慮せずに、C の標準ライブラリファイル float.h で定義されているパラメータを Ch で計算する方法を説明します。プログラミング言語が Inf および NaN のメタ数値をサポートできる場合には、プログラムがこれらのパラメータに依存する度合いは低くなります。 Ch プログラミングでは、これらのパラメータの代わりに、Inf や NaN などのメタ数値を使用することが推奨されます。

最小浮動小数点数_MIN および FLT_MINIMUM

パラメータ FLT_MIN は、標準 C ライブラリのヘッダーファイル float.h で、正規化された float 型の最小の正の浮動小数点数として定義されています。数値が FLT_MIN 未満の場合は、**アンダーフロー**と呼ばれます。IEEE 754 規格で**段階的アンダーフロー**が定義されているため、Ch では、非正規化された float 型の最小の正の浮動小数点数を FLT_MINIMUM と定義しています。段階的アンダーフローにより、 Ch の式 x - y == 0 は、x = y のとき true になります。段階的アンダーフローのないシステムでは true にはなりません。このパラメータはプログラミングの観点からは非常に有用です。たとえば、FLT_MINIMUM と FLT_MIN の値が、それぞれ、1.401298e-45 と 1.175494e-38 であるとします。次の Ch コードは、この 2 つのパラメータの微妙な違いを示しています。

```
float f, *flt_minimum;
int minimum, i;
minimum = 1;                        // memory location becomes 00000001
flt_minimum = &minimum;             // *flt_minimum becomes FLT_MINIMUM
i = *flt_minimum > 0.0;             // i becomes 1
i = FLT_MIN > *flt_minimum;         // i becomes 1
i = fabs(*flt_minimum) > 0.0;       // i becomes 1
f = (*flt_minimum)/(*flt_minimum); // f becomes 1.0; note 0.0/0.0 = NaN
f = f/1.e-46                      // f becomes Inf: 1.e-46 < FLT_MINIMUM
```

多価複素関数のブランチカット処理でのこれら 2 つの数値の適用については、第 13章で説明します。

マシンイプシロン FLT_EPSILON

マシンイプシロン FLT_EPSILON は、浮動小数点数で表現可能な 1 の次に大きい数と 1 との間の差です。C ではヘッダーファイル float.h で定義されているこのパラメータは、Ch ではシステム定数です。このパラメータは科学計算にとって非常に有用です。たとえば、極度に小さい数値と大きい数値を加算した場合、浮動小数点表現の有限精度および加算演算の調整によって、小さい数値は加算結果に影響しないことがあります。FLT_EPSILON を使用して、大きい正の数値 y に小さい正の数値 x を加えた場合は、10 進 3 桁以上の y の有効数字を得ることができます。これは次のコードでテストできます。

```
if(x < y * FLT_EPSILON * 1000)
```

次の Ch コードは、マシンイプシロンを計算し、計算結果を画面に出力します。

```
float epsilon;
epsilon = 1.0;
while(epsilon+1 > 1)
  epsilon /= 2;
epsilon *= 2;
printf("The machine epsilon FLT_EPSILON is %e", epsilon);
```

SUN SPARCStations で上記のコードを実行すると、次の結果が出力されます。

The machine epsilon FLT_EPSILON is 1.192093e-07

これは、C のヘッダーファイル float.h で定義されているパラメータ FLT_EPSILON の値と同じです。上記のパラメータ FLT_EPSILON の計算は、既定の丸めモード (最近似値への丸め) を使用する Ch では簡単ですが、他の丸めモードでは適切な結果が得られないことがあります。このパラメータを得る堅実な方法 (Plauger、1992) は、セクション 12.2.1で説明されているように、float 型変数のメモリのビットパターンを操作することです。

最大浮動小数点数 FLT_MAX

C のヘッダーファイル float.h で定義されているパラメータ FLT_MAX は、表現可能な最大の有限の浮動小数点数です。FLT_MAX より大きい値は Inf と表され、−FLT_MAX より小さい値は −Inf と表されます。FLT_MAX の値が $fltmax * 10^e$ として表される場合は、次の 2 つの方程式が満たされます。

$$(fltmax + FLT_EPSILON) * 10^e = Inf$$

$$(fltmax + FLT_EPSILON/2) * 10^e = FLT_MAX$$

ここで、マシンイプシロン FLT_EPSILON はセクション 12.2.1で定義されているとおりです。指数値 e が計算されます。次の Ch プログラムは、マシンの FLT_MAX だけでなく、FLT_MAX_10_EXP および FLT_MAX_EXP を計算して、画面に結果を出力します。FLT_MAX_10_EXP の値は、10 を累乗すると、表現可能な有限浮動小数点数の範囲の最大の整数となります。FLT_MAX_EXP の値は、2 を累乗して 1 を引くと、表現可能な有限浮動小数点数になる最大の整数となります。次の例では、説明のために、while ループ制御構造のみを使用しています。

```
float b, f, flt_max;
int e, i, flt_max_exp, flt_max_10_exp;
b = 10; e = 0; f = b;
/* calculate exponential number e, 38 in the example */
while(f != Inf)
{
```

```
  e++; f*=b;
}
flt_max_10_exp = e;
/* calculate leading non-zero number, 3 in the example */
i = 0; f = 0.0;
while(f != Inf)
  f = ++i * pow(b, e);
/* calculate numbers after decimal point, 40282347... in the example */
flt_max = i;
while(e != 0)
{
  flt_max = --flt_max * b;
  e--; i = 0; f = 0.0;
  while( f != Inf && i < 10)
  {
    f = ++flt_max * pow(b, e);
    i++;
  }
}
f = frexp(flt_max, &flt_max_exp);       // calculate FLT_MAX_EXP
printf("FLT_MAX = %.8e \n", flt_max);
printf("FLT_MAX (in binary format) = %b \n", flt_max);
printf("FLT_MAX_10_EXP = %d \n", flt_max_10_exp);
printf("FLT_MAX_EXP = %d \n", flt_max_exp);
```

SUN SPARCStations では、上記のコードの出力結果は次のとおりです。

FLT_MAX = 3.40282347e+38
FLT_MAX (in binary format) = 01111111011111111111111111111111
FLT_MAX_10_EXP = 38
FLT_MAX_EXP = 128

上記の FLT_MAX、FLT_MAX_10_EXP、および FLT_MAX_EXP の値は、C のヘッダーファイル float.h で定義されているパラメータと同じです。最初のステートメントの宣言を float 型から double 型へ変えるだけで、対応する DBL_MAX、DBL_MAX10EXP、および DBL_MAX EXP の各極値を double 型で取得できます。この場合、ポリモーフィックな算術演算子と数学関数 **pow**() および **frexp**() によって、double 型のデータが返されます。

上記の浮動小数点の限界値の計算では、ユーザーはマシンの浮動小数点数の複雑な表現を考慮する必要はありません。マシンの浮動小数点数の表現を知っていれば、限界値の計算はもっと簡単になります。たとえば、表 6.1に示すとおり、FLT_MAX の値は 16 進形式で $(7F7FFFFF)_{16}$ と表されます。表現可能な最大の有限浮動小数点数である FLT_MAX を取得するには、次の Ch プログラムを使用できます。

```
int i; float *flt_max;
flt_max = &i;        // flt_max points to the memory location of i
i = 0X7F7FFFFF;      // *flt_max becomes FLT_MAX
```

最大浮動小数点数 FLT_MAX は、入出力関数 **scanf**() に 2 進数の入力形式"%32b"を使用しても簡単に得ることができます。興味のある読者は、これらの方法以外に、マシンアーキテクチャを考慮せずに、C や Fortran のプログラムで表現可能な最大の有限浮動小数点数 FLT_MAX を計算する方法を考えてみてはいかがでしょうか。主要な困難は、浮動小数点数の計算では内部調整が行われるため、大きい数値に対して極度に小さい数値を足したり引いたりしても、それらは無視されるということです。たとえば、f = FLT_MAX + 3.0e30 のコマンドを実行すると、変数 f に FLT_MAX の値が得られます。$3.0 * 10^{30}$ という値は小さな数ではありませんが、FLT_MAX に比べると非常に小さいため、この加算演算では無視されてしまいます。Ch で書かれた次の 2 つの式は、FLT_MAX と Inf の違いを明確に示しています。

$$1/\mathrm{Inf} * \mathrm{FLT_MAX} = 0.0$$

,

$$1/\mathrm{FLT_MAX} * \mathrm{FLT_MAX} = 1.0$$

.

12.2.2 メタ数値を使用したプログラミング

Ch 言語では、実数の −0.0 と 0.0 は区別されます。0.0、−0.0、Inf、−Inf、および NaN のメタ数値は、科学計算では非常に有用です。たとえば、関数 $f(x) = e^{\frac{1}{x}}$ は、図 12.1 (23章のプログラム 23.12 (518ページ) から生成されたもの) に示すように、座標原点で途切れています。

Ch では、この不連続をうまく扱うことができます。Ch の式 exp(1/0.0) による評価の結果、Inf が返され、exp(1/(-0.0)) が 0.0 となります。これは数学式 $e^{\frac{1}{0+}}$ および $e^{\frac{1}{0-}}$ または $\lim_{x\to 0_+} e^{\frac{1}{x}}$ および $\lim_{x\to 0_-} e^{\frac{1}{x}}$ にそれぞれ対応します。

さらに、**exp**(1.0/Inf) と **exp**(1.0/(−Inf)) の式の評価によって、1.0 の値が得られます。別の例として、IEEE 754 規格で推奨されている関数 finite(x) は、Ch の式-Inf < x && x < Inf に相当します。ここで、x は float 型または double 型の変数または式です。x が float 型の場合、-Inf < x && x < Inf は-FLT_MAX <= x && x <= FLT_MAX と同じです。x が double 型の場合は、-Inf < x && x < Inf は、-DBL_MAX <= x && x <= DBL_MAX と同じです。"$if\ -\infty < value <= \infty, then\ y\ becomes\ \infty$"という数学的ステートメントも、Ch では、次のように簡単に記述できます。

```
if(-Inf < value && value <= Inf) y = Inf;
```

ただし、コンピュータは式を段階的にしか評価できません。メタ数値は浮動小数点数の限界値ですが、数理解析の代替とすることはできません。たとえば、2.718281828... に相当する自然数 e は、次の式の制限値として定義されます。

$$\lim_{x\to\infty}\left(1+\frac{1}{x}\right)^x = e$$

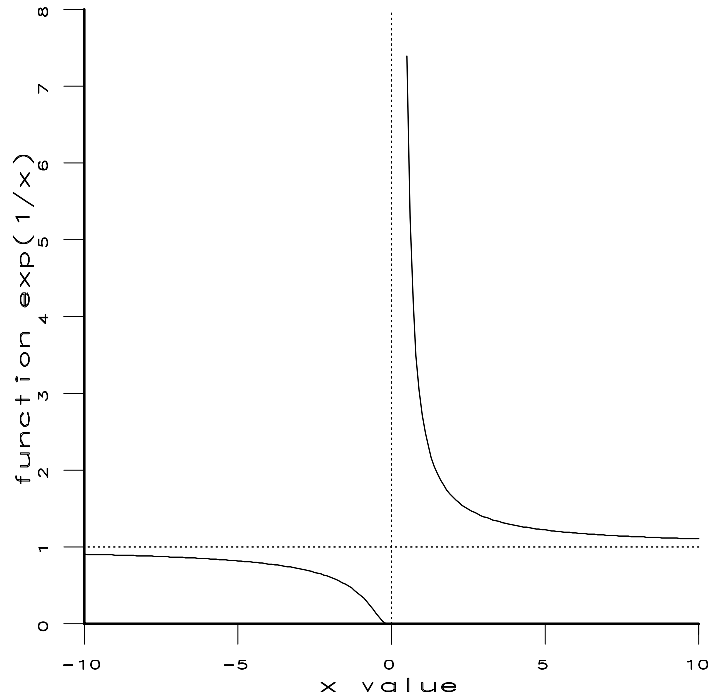


図 12.1: 関数 $f(x) = e^{\frac{1}{x}}$.

しかし、 Ch の式 **pow**(1.0 + 1.0/Inf, Inf) の結果値は、NaN になります。この式の評価は次のように行われます。

$$\left(1.0 + \frac{1.0}{Inf}\right)^{Inf} = (1.0 + 0.0)^{Inf} = 1.0^{Inf} = NaN$$

上記の式で、Inf の代わりに値 FLT_MAX を使用すると、次の結果が得られます。

$$\left(1.0 + \frac{1.0}{FLT_MAX}\right)^{FLT_MAX} = (1.0 + 0.0)^{FLT_MAX} = 1.0^{FLT_MAX} = 1.0$$

メタ数値 NaN は順序付けされないため、関係演算を行うプログラムの扱いには注意が必要です。たとえば、式 `x > y` は、x と y のどちらかが NaN の場合は、`!(x <= y)` と同じ結果になりません。別の例として、次の Ch コード例

```
if(x > 0.0)  function1();
else function2();
```

は、次のコード例とは異なります。

```
if(x <= 0.0) function2();
else function1();
```

これら 2 つのコード例で同じ結果を得るには、後者の `if` ステートメントを `if(x <= 0.0 || isnan(x))` と書き換える必要があります。

第13章　複素数を使用したプログラミング

実数の拡張である複素数には、科学および工学の分野において幅広い用途があります。複素数は科学プログラミングにおいて重要であるため、通常、数値計算を重要視するプログラミング言語とソフトウェアパッケージはなんらかの方法で複素数をサポートしています。

たとえば、主に科学計算に使用される Fortran 言語では、complex データ型が当初から用意されています。数値の処理を重要視する科学計算は当初の設計目標ではなかったので、C の初期バージョンには基本データ型として complex は用意されていません。complex データ型は、C99 で追加されています。Ch は、C99 で要求されているすべての機能と拡張機能をサポートします。一般的な数学関数は、さまざまなブランチカット用の省略可能な引数を使用して複素数を処理するためにオーバーロードされます。Ch には、実数のメタ数値である Inf、−Inf、および NaN と、符号付きのゼロである 0.0 と −0.0 が用意されています。これにより、プログラマは簡単に、2 進数の浮動小数点演算で IEEE 754 規格の有用性を活かすことができます。

Ch では、IEEE 754 規格の精神に従い、算術演算だけではなく使用頻度の高い数学関数でも、メタ数値の考え方を複素数にまで広げています。 Ch では、符号付きゼロを持つ浮動小数点実数、符号なしゼロを持つ複素数、および NaN (非数) と Inf (無限大) を統一性のある一貫した方法で処理します。

13.1　複素数

13.1.1　複素定数と複素変数

複素数 $z \in \mathcal{C} = \{(x, y) \mid x, y \in \mathcal{R}\}$ は、次のように

$$z = (x, y) \tag{13.1}$$

特定の加算規則と乗算規則を持つ順序対として定義できます [10][17]。実数 x と実数 y は、z の**実部**と**虚部**と呼ばれます。$(x, 0.0)$ の対を実数と識別した場合、実数 $\mathcal{R}$ は $\mathcal{C}$ のサブセットになります。つまり、$\mathcal{R} = \{(x, y) \mid x \in \mathcal{R}, y = 0.0\}$ であり、$\mathcal{R} \subset \mathcal{C}$ です。実数が x または $(x, 0.0)$ のいずれかと見なされ、$i * i = -1$ として i で**純虚数** $(0, 1)$ を示す場合、複素数は数学的に次のように表すことができます。

$$z = x + iy \tag{13.2}$$

(13.1) および (13.2) の両方の式で、コンピュータ言語で複素数を実装できます。Fortran、Ada、Common Lisp などの汎用コンピュータプログラミング言語では式 (13.1) を使用するのに対して、一部の数学ソフトウェアパッケージでは式 (13.2) を使用する傾向があります。

科学プログラミングでは Fortran が優位なので、Ch では、複素数コンストラクタ **complex**(x, y) によって複素数を作成できます。このとき、$x, y \in \mathcal{R}$ です。たとえば、実部が 3.0 で虚部が 4.0 である

複素数は、**complex**(3.0, 4.0) で作成できます。新しい型修飾子 **complex** は、Ch のキーワードです。現在の実装では、複素数は、内部的に 2 つの float 型で構成されます。そのため、複素コンストラクタの引数が float 型ではない場合は、内部的に float 型にキャストされます。

すべての浮動小数点の定数は、 Ch では既定により double 型です。float 定数は、浮動小数点の定数にサフィックスとして F または f を付加することで得ることができます。複素コンストラクタは、入力引数のデータ型に応じて、complex 型または double complex 型の複素数をポリモーフィックに返します。

たとえば、**complex**(3, 4.0)、**complex**(3.0f, 4.0)、および **complex**(3.0, 4.0F) は、double complex 型の数値 **complex**(3.0, 4.0) を返します。

単純な複素変数だけではなく、*complex* **へのポインタ**、*complex* **の配列**、*complex* **へのポインタ配列**などを宣言できます。これらの複素変数の宣言は、C における他のデータ型の宣言に類似しています。Ch における complex の配列とポインタは、浮動小数の float 型および double 型と同じ方法で操作されます。次のコード例は、Ch での complex の宣言方法と操作方法を示しています。

```
double complex z;       // declare z as double complex variable
float complex z1;       // declare z1 as float complex variable
complex *zptr1;         // declare zptr1 as pointer to complex variable
complex z2[2], z3[2,3];// declare z2 and z3 as arrays of complex
complex *zptr2[2][4];  // declare zptr2 as array of pointer to complex
zpt1r = &z1;            // zptr1 point to the address of z1
*zptr1 = complex(1,2); // z1 becomes 1+i2
```

複素数は、C99 と C++でサポートされています。C99 と C++の両方との互換性を得るために、Ch では、1 つのマクロ、2 つの型、およびいくつかの関数プロトタイプが **complex.h** と **complex** の両方のヘッダーファイルで定義されています。マクロ **I** は、虚数と単位長を表す `complex(0.0, 1.0)` として定義されています。

13.2　複素平面と複素メタ数値

数学的に、複素数は、図 13.1に示す拡張された複素平面で表すことができます [10][17]。

図 13.1では、リーマン球面 Γ 上の点と拡張された複素平面 $\mathcal{C}$上の点の間には、1 対 1 の対応があります。球面の表面上の点 p は、点 z と球面の北極 N を結ぶ直線の交点によって決定されます。拡張された複素平面には、複素無限大が *1* つだけあります。

北極 N は、無限遠点に対応します。浮動小数点数の有限表現のために、この章では、図 13.2に示すように**有限の拡張された複素平面**を取り上げています。

$|x| <$ FLT_MAX および $|y| <$ FLT_MAX の範囲内のすべての複素値は、有限浮動小数点数で表すことができます。変数 x は複素数の実部を、y は虚部を表すために使用されます。事前定義されたシステム定数 FLT_MAX は、float データ型で表現できる最大の有限浮動小数点数です。この長方形の領域の外では、複素数は、Ch では ComplexInf または complex(Inf,Inf) として表される複素無限大として処理されます。リーマン球面 Γ 上の点と拡張された複素平面上の点の間の 1 対 1 の対応は、単位球面 Λ と有限の拡張された複素平面では有効ではありません。単位球面の上部 Λ_1 の表面上の点はすべて、複素無限大に対応します。

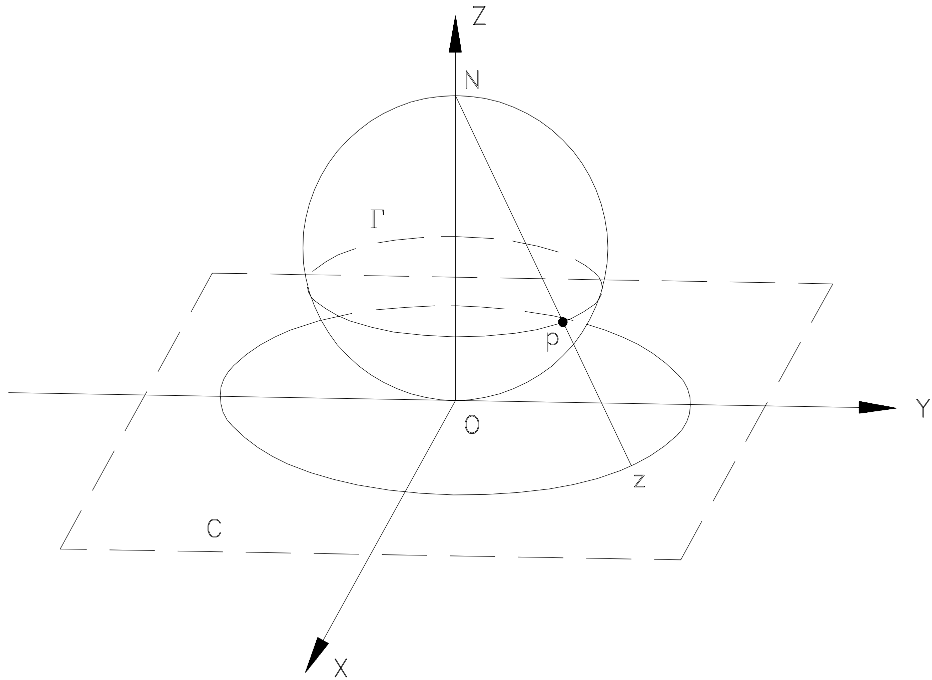


図 13.1: リーマン球面 Γ と拡張された複素平面

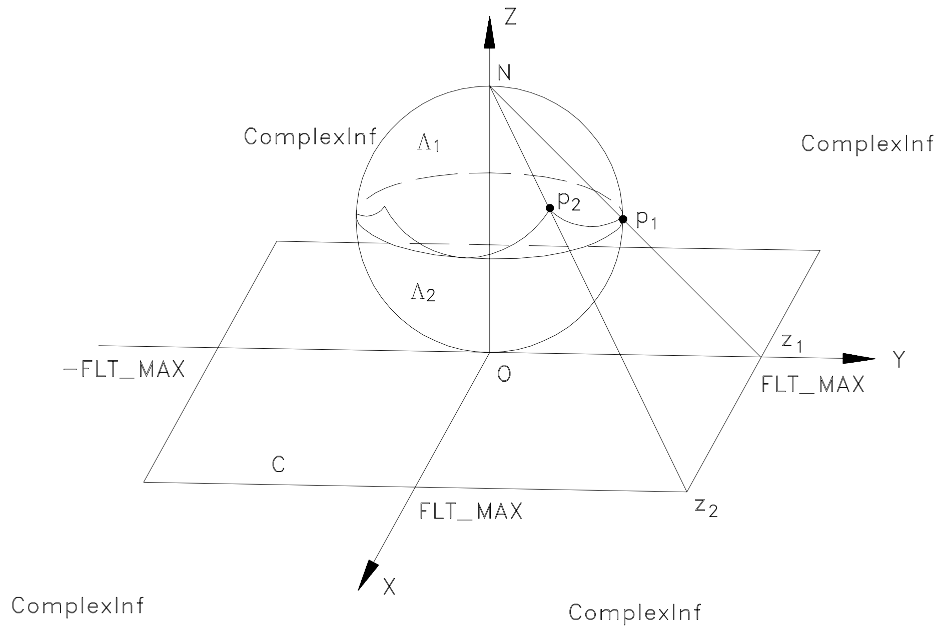


図 13.2: 単位球面 Λ と有限の拡張された複素平面

この半球の下部 Λ_2 上の点と有限の拡張された複素平面上の点は、1 対 1 で対応します。表面 Λ_1 と Λ_2 の間の境界は、オーバーフローのしきい値に対応します。たとえば、単位半球 Λ 上の点 p_1 と p_2 は、図 13.2に示すように、それぞれ有限の拡張された複素平面上の点 $z_1 =$ complex(FLT_MAX, 0.0) と点 $z_2 =$ complex(FLT_MAX, FLT_MAX) に対応します。有限の拡張された複素平面の原点は、complex 型のゼロを表す complex(0.0, 0.0) です。Ch では、未定義の複素数または数学的に不定の複素数は、複素非数 (Complex-Not-a-Number) を表す complex(NaN, NaN) または ComplexNaN で示されます。特殊な複素数である ComplexInf と ComplexNaN は、複素メタ数値と呼ばれます。

数学的な無限大 $\pm\infty$ により、実数では正のゼロ 0.0 と負のゼロ −0.0 を区別する必要が生じています。実数が正数または負数を通じて原点に接近できる数直線とは異なり、複素平面の原点には、限界値 $\lim_{r\to 0} re^{i\theta}$ の観点から任意の方向で到達できます。ここで、r は係数、θ は複素数の位相です。

したがって、Ch における複素演算と複素関数では、複素数の実部と虚部の 0.0 と −0.0 は区別されません。これらの違いのために、実数と複素数、特に実メタ数値と複素メタ数値の一部の演算と関数は、異なる方法で処理する必要があります。

たとえば、IEEE 754 規格に従って、2 つの実数の正の無限大の加算は、Ch では無限大の値です。2 つの複素無限大の加算は、複素解析によると不定ですが、ComplexInf の値は、内部的には 2 つの正の無限大 Inf で表されます。

別の例として、C 標準に従って、Ch の数学関数 **atan2**(y, x) は $[-\pi, \pi]$ の範囲の値を返します。式 **atan2**($-0.0, -1$) の値は $-\pi$ です。

この結果を複素数 $-1.0 - i0.0$ の位相角度として使用して、Ch では **sqrt**(complex($-1.0, -0.0$)) と表現される $-1.0 - i0.0$ の平方根は complex($0.0, -1.0$) になり、それは $\cos(-\pi/2) + i\sin(-\pi/2) = 0.0 - i$ によって得られます。

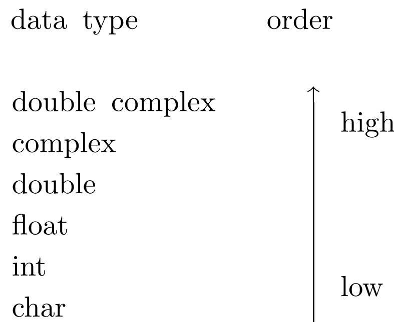
定義では、これは、式 **sqrt**(complex($-1.0, -0.0$), 1) によって得られる複素数 complex($-1.0, -0.0$) の平方根関数の 2 番目の分岐であり、関数 **sqrt**() の 2 番目の引数が分岐数を示します。既定値は 0 です。

この例に示すように、Ch における数学関数は引数の変数の数によってポリモーフィックであり、関数 **sqrt**() は、実数の平方根を計算するために使用できるだけでなく、複素数の平方根の異なる分岐を計算するために使用することもできます。

数学関数のポリモーフィズムと引数の変数の数により、Ch における複素数の科学計算は、Fortran やその他の言語と比較して、はるかに単純です。

13.2.1 データ変換規則

Ch は、型指定の緩やかな言語です。呼び出し元関数のすべての引数は、呼び出し先のデータ型と互換性があるかどうかチェックされます。演算のオペランドのデータ型に対する互換性もチェックされます。データ型が一致しない場合、システムはエラーを通知し、プログラムをデバッグしやすくするための通知メッセージを出力します。ただし、型の自動変換を禁止する Pascal などの言語と異なり、Ch にはいくつかのデータ型変換規則が組み込まれ、必要に応じて呼び出すことができます。これにより、プログラムでの多数の明示的な型変換コマンドが不要になります。Ch におけるデータ型 (data type) は、以下のように順序 (order) で並べられます。



char が最下位のデータ型であり、double complex が最上位のデータ型です。このセクションでは、既定の変換規則について簡潔に説明します。

1. char、int、float、および double は、ISO C データ変換規則に従って変換できます。char データ型の変換では、文字の ASCII 値が使用されます。データ型を下位に下げると、情報が失われる場合があります。

2. char、int、float、および double は、虚部がゼロの complex 型に変換できます。実数を複素数にキャストすると、Inf と −Inf の値は ComplexInf になり、NaN の値は ComplexNaN になります。double から complex 型に変換すると、情報が失われる場合があります。実数は、複素を構成する関数 **complex**(x,y) により明示的に complex 型にキャストできます。この関数の詳細については、セクション 13.5を参照してください。

3. complex が char、int、float、および double に変換されるとき、虚部がゼロの場合はその実部のみが使用されて、虚部は破棄されます。虚部がゼロに等しくない場合、変換後の実数は NaN になります。複素数の実部と虚部は、関数 **real**(z) と関数 **imag**(z) により明示的に得ることができます。 これらの関数の詳細については、セクション 13.5を参照してください。

   複素数が、f = z などの代入ステートメントにより暗黙的に、または **real**(z)、**imag**(z)、**float**(z)、**double**(z)、(**float**)z、および (**double**)z により明示的に実数に変換されるとき、ゼロの符号は引き継がれません。複素数を char や int などの整数値に変換することは、虚部がゼロと同一でない場合は、**real**(z) を整数値に変換することに相当します。

   たとえば、`i = ComplexInf` は、i を INT_MAX と等しくします。ただし、**real**() または **imag**() が lvalue として使用される場合、rvalue のゼロの符号は保持され、それにより、複素数の計算において、符号付きのゼロを使用できます。

   lvalue は、代入ステートメントの左側に出現する任意のオブジェクトです。lvalue は、関数または定数ではなく、変数やポインタなどのメモリを参照します。一方、*rvalue* は、代入ステートメントの右側で式の値を参照します。lvalue の詳細については、セクション 13.6を参照してください。

4. データ型が混在する加算、減算、乗算、除算などの二項演算では、演算の結果は、二つのオペランドの優先順位が上位のデータ型が使用されます。たとえば、int と double を加算すると、結果は double になります。二項演算の 2 つのオペランドの片方が complex 型であり、他方のオペランドのデータ型が実数の場合、演算の実行前に実数が complex にキャストされます。この変換規則は、lvalue と rvalue のデータ型が異なる代入ステートメントでも有効です。

5. ポインタ代入ステートメントでは、lvalue と rvalue のポインタ型は違っていてもかまいません。それらは内部的に調整されます。ISO C 規格に準拠するため、rvalue のデータ型を、 Ch の代入で lvalue のデータ型に明示的にキャストすることもできます。たとえば、ステートメント $fp = (float*)intptr$ は、そのアドレスが float ポインタ fp に割り当てられる前に、整数ポインタ $intptr$ を float 型のポインタにキャストします。

   ただし、$intptr$ によってポイントされる内容が、このデータ型キャスト演算によって変更されることはありません。たとえば、$*intptr$ が 90 の場合、$*fp$ の値は 90 に等しくなりません。これは、int と float の内部表現が異なるためです。複素変数のメモリは、ポインタによってアクセスできます。複素変数の実部または虚部が float ポインタによって得られる場合、ゼロの符号は引き継がれます。詳細については、セクション 13.6を参照してください。

異なるデータ型が Ch でどのように自動的に変換されるかを以下のコードセグメントに示します。

```
char c;
int i;
float f;
double d;
complex z, *zptr;
c = 'a';              // c is  'a'
i = c;                // i is 97, ASCII number of  'a'
f = i;                // f is 97.0
d = i;                // d is 97.0
z = complex(c+1, f);  // z is 98.0 + i 97.0
z = complex(Inf, Inf);// z is ComplexInf
z = Inf;              // z is ComplexInf
z = -Inf;             // z is ComplexInf
f = z;                // f is NaN, since real(ComplexInf) is NaN
d = z;                // d is NaN, since real(ComplexInf) is NaN
i = Inf;              // i is 2147483647 = INT_MAX,
i = z;                /* i is 2147483647, int of NaN is 2147483647
                         plus warning message */
z = complex(d+1, 3);  // z is 98.0 + i 3.0
c = z;                // c is the delete character, ASCII number is 127
i = z;                // i is 2147483647, int of NaN
f = z;                // f is NaN
d = z;                // d is NaN
z = NaN;              // z is ComplexNaN
zptr = &z;            // zptr point to address of z
zptr++;               // zptr point to memory z plus 8 bytes
```

13.3　複素数に対する入出力

複素数は Ch における基本のデータ型なので、このデータ型に対する入出力も、実数と同じ方法で処理されることが望まれます。Fortran と同じように、複素数の実部と虚部は、セクション 13.4および13.5で説明する関数 **real**(z) と関数 **imag**(z) により、2 つの個別の float として処理できます。その際、実数用の **printf**() や **scanf**() などの標準入出力関数はすべてそのまま使用できます。

このセクションでは、標準入出力関数によって複素数を単一のオブジェクトとして処理する方法について説明します。スペースに限りがあるため、**printf**() 関数に関連する拡張部分についてのみ説明します。ただし、根底にある原則は、他の入出力関数にも適用できます。Ch における **printf**() 関数の書式を以下に示します。

```
int printf(char *format, arg1, arg2, ...)
```

printf() 関数は、`format` によってポイントされる文字列の制御下で標準出力デバイスに出力し、出力された文字数を返します。format 文字列に、通常の文字と、文字%で始まり変換文字で終わる変換仕様という 2 種類のオブジェクトが含まれる場合は、**printf**() の ISO C 規則が使用されます。**printf**() の format 文字列に通常の文字だけが含まれる場合は、以降の数値定数または変数が、事前設定された既定の書式に従って出力されます。

printf() 関数では、float に対する単一の変換仕様が、複素数の実部と虚部の両方で使用されます。complex の既定の書式は%.2f であり、これが複素数の実部と虚数の両方に適用されます。メタ数値 ComplexInf と ComplexNaN は、入出力関数では通常の複素数として処理されます。

デバッグを容易にするため、ComplexInf と ComplexNaN の既定の出力は、それぞれ complex(Inf, Inf) と complex(NaN, NaN) になります。complex 型のゼロの既定の出力は、complex(0.00,0.00) です。実部と虚部の書式は、書式指定子で制御できます。入出力関数 **printf**() と **scanf**() による複素数の処理方法を以下の Ch プログラムに示します。

```
complex z1;
double complex z2, *zptr;
zptr = &z2;          /* zptr points to z2's memory location  */
printf("Please type in real and imaginary of two complex numbers \n");
scanf(&z1, zptr);
printf("The first complex is ", z1, "\n");
printf("The second complex is ", z2, "\n");
printf("The second complex is  %f \n", z2);
```

上記のプログラムの対話的な実行結果を以下に示します。

```
Please type in real and imaginary of two complex numbers
```

1 2.0 3.0 4

```
The first complex is complex(1.0000,2.0000)
The second complex is complex(3.0000,4.0000)
The second complex is complex(3.000000,4.000000)
```

斜体になっている 2 行目の部分が入力であり、残りの部分がプログラムの出力です。

13.4 複素演算

Ch における複素数に対する算術演算と関係演算は、実数に対する演算と同じ方法で処理されます。このセクションでは、これらの演算が Ch ではどのように定義され、処理されるかについて説明します。

13.4.1 通常の複素数による複素演算

複素数の符号反転と、2 つの複素数の算術演算と比較演算を表 13.1に定義します。

表 13.1: 複素演算

定義	Ch 構文	Ch における意味
符号反転	−z	$-x - iy$
加算	z1 + z2	$(x_1 + x_2) + i(y_1 + y_2)$
減算	z1 − z2	$(x_1 - x_2) + i(y_1 - y_2)$
乗算	z1 ∗ z2	$(x_1 * x_2 - y_1 * y_2) + i(y_1 * x_2 + x_1 * y_2)$
除算	z1 / z2	$\frac{x_1 * x_2 + y_1 * y_2}{x_2^2 + y_2^2} + i\frac{y_1 * x_2 - x_1 * y_2}{x_2^2 + y_2^2}$
等しい	z1 == z2	$x_1 == x_2$ および $y_1 == y_2$
等しくない	z1 != z2	x_1!=x_2 または y_1 != y_2

複素数 z、$z1$、および $z2$ は、それぞれ、$x + iy$、$x1 + iy_1$、および $x2 + iy_2$ と定義されます。
複素数の符号反転は、複素数の実部と虚部の両方の符号を変更します。2 つの複素数の加算は、2 つの複素数の実部と虚部を別々に加算します。2 つの複素数の減算は、2 つ目の複素数の実部と虚部を、1 つ目の複素数の実部と虚部からそれぞれ減算します。表 13.1には、虚数 i を複素数 complex(0, 1) として処理する、2 つの複素数の乗算と除算が定義されています。
実数のオペランドと複素数のオペランドによる二項演算では、通常の実数のオペランドが演算前に complex にキャストされます。複素数には順序がありません。ある複素数が別の複素数よりも大きいまたは小さいかどうかを判断するための比較はできません。ただし、2 つの複素数が等しいか等しくないかは検査できます。2 つの複素数は、2 つの複素数の実部と虚部両方がそれぞれ互いに等しい場合のみ、等しいと見なされます。2 つの複素数は、実部または虚部が等しくない場合は、等しくないと見なされます。

13.4.2 複素メタ数値による複素演算

前述の複素演算の定義では、すべてのオペランドは通常の複素数であることを前提としています。複素数の実部と虚部は、2 つの通常の float 型の浮動小数点数として処理されます。オペランドの値に複素メタ数値が関係する場合、表 13.1で定義された定義は有効でないことがあります。たとえば、ComplexInf は、内部的には complex(Inf, Inf) と表現されます。
表 13.1に定義された complex の加算の定義と、Ch における実数の加算規則に従うと、2 つの ComplexInf の加算結果は complex(Inf, Inf) になります。しかし、2 つの複素無限大の加算は、数学的には不

定です。このため、通常の複素数と複素メタ数値の両方による算術演算と関係演算の結果を、表 13.2～13.7に定義します。

表 13.2: complex の符号反転の結果

符号反転 −				
オペランド	complex(0.0, 0.0)	z	ComplexInf	ComplexNaN
結果	complex(0.0, 0.0)	−z	ComplexInf	ComplexNaN

表 13.3: Complex の加算と減算の結果

加算および減算 ±				
左オペランド	右オペランド			
	complex(0.0, 0.0)	z2	ComplexInf	ComplexNaN
complex(0.0, 0.0)	complex(0.0, 0.0)	±z2	ComplexInf	ComplexNaN
z1	z1	z1 ± z2	ComplexInf	ComplexNaN
ComplexInf	ComplexInf	ComplexInf	ComplexNaN	ComplexNaN
ComplexNaN	ComplexNaN	ComplexNaN	ComplexNaN	ComplexNaN

プログラマの観点からは、Ch でオペランドまたは引数として使用された場合、complex(±0.0, ±0.0) の値は complex(0.0, 0.0) と同じです。この後の説明では、複素数の実部と虚部の正のゼロ 0.0 と負のゼロ −0.0 は、同じであると見なされます。したがって、complex(0.0, 0.0) の符号反転では complex(−0.0, −0.0) が返りますが、表 13.2の結果には complex(0.0, 0.0) と記載されます。複素無限大の符号反転は、依然として複素無限大です。同様に、複素非数 (complex not-a-number) の符号反転は ComplexNaN です。

表 13.3～13.5の二項演算で、オペランドの片方が ComplexNaN であれば、結果は ComplexNaN です。2 つのオペランドの片方が ComplexInf であり、他方が有限複素数の場合、加算と減算の結果は ComplexInf です。実数とは異なり、2 つの ComplexInf の加算と減算は ComplexNaN です。

ComplexInf と complex(0.0, 0.0) の乗算は ComplexNaN です。ComplexInf と有限の非ゼロ値の乗算は ComplexInf です。2 つの ComplexInf の乗算は ComplexInf です。実数と同じように、complex(0.0, 0.0) の complex(0.0, 0.0) による除算と ComplexInf の ComplexInf による除算は ComplexNaN です。complex(0.0, 0.0) で除算された有限数または無限数は ComplexInf になります。有限数による ComplexInf の除算は ComplexInf です。理論的には、2 つの複素無限大は、互いに等しいかどうかにかかわらず、比較することはできません。

表 13.4: Complex の乗算の結果

乗算 *				
左オペランド	右オペランド			
	complex(0.0, 0.0)	z2	ComplexInf	ComplexNaN
complex(0.0, 0.0)	complex(0.0, 0.0)	complex(0.0, 0.0)	ComplexNaN	ComplexNaN
z1	complex(0.0, 0.0)	z1*z2	ComplexInf	ComplexNaN
ComplexInf	ComplexNaN	ComplexInf	ComplexInf	ComplexNaN
ComplexNaN	ComplexNaN	ComplexNaN	ComplexNaN	ComplexNaN

表 13.5: Complex の除算の結果

除算 /				
左オペランド	右オペランド			
	complex(0.0, 0.0)	z2	ComplexInf	ComplexNaN
complex(0.0, 0.0)	ComplexNaN	complex(0.0, 0.0)	complex(0.0, 0.0)	ComplexNaN
z1	ComplexInf	z1/z2	complex(0.0, 0.0)	ComplexNaN
ComplexInf	ComplexInf	ComplexInf	ComplexNaN	ComplexNaN
ComplexNaN	ComplexNaN	ComplexNaN	ComplexNaN	ComplexNaN

表 13.6: Complex の同等性比較の結果

等しい比較 ==				
左オペランド	右オペランド			
	complex(0.0, 0.0)	z2	ComplexInf	ComplexNaN
complex(0.0, 0.0)	1	0	0	0
z1	0	z1 == z2	0	0
ComplexInf	0	0	1	0
ComplexNaN	0	0	0	0

表 13.7: Complex の非同等性比較の結果

等しくない比較 !=				
左オペランド	右オペランド			
	complex(0.0, 0.0)	z2	ComplexInf	ComplexNaN
complex(0.0, 0.0)	0	1	1	0
z1	1	z1 != z2	1	0
ComplexInf	1	1	0	0
ComplexNaN	0	0	0	0

ただし、 Ch では、2 つの ComplexInf は、表 13.6に示すように、プログラミング上の観点からは同一であると見なされます。実数における NaN のように、2 つの ComplexNaN を比較すると、論理 false が得られます。この設計上の考慮は、表 13.7に示す `not equal` 関係演算にも反映されます。

13.5 複素関数

Ch で実装される数学関数には、ポリモーフィズムに加え、変数の引数を使用できます。これは、複数の分岐がある複素数学関数を計算する際に非常に便利です。数学関数が実関数として実数の引数を 1 つだけ持つ場合、特に説明がない限り、2 番目の引数は、関数を複素関数にします。2 番目の引数の整数値は、複素関数の分岐を示します。

この 2 番目の引数が存在するとき、1 番目の引数のデータ型の順序が complex よりも下位の場合、1 番目の引数は、前述したデータ型変換規則に従って複素数にキャストされます。実関数として 2 つの引数をとる数学関数では、2 つの入力引数のいずれかが complex であれば、その数学関数は複素関数になります。分岐指示子として 3 番目の引数が追加されている場合は、最初の 2 つの引数のデータ型が complex 以下であれば、関数は複素関数になります。それらのデータ型が complex より下位の場合は、複素数にキャストされます。

13.5.1 通常の複素数を使用する複素関数の結果

複素数に関連する組み込み関数を、表 13.8に定義と一緒に示します。

これらの関数の入力引数は、複素数、変数、または式のどれでもかまいません。表現上、複素数 z、$z1$、および z_2 は、それぞれ $x + iy$、$x_1 + iy_1$、および $x_2 + iy_2$ と定義されます。整数値 k、k_1、および k_2 は、複素関数の分岐数です。呼び出し元関数のこれらの分岐数の引数が整数でない場合、それらは内部的に整数にキャストされます。表 13.8の第 2 列に示す数式では、数学関数の引数が通常の実数であれば、その数学関数は実数学関数です。

複素メタ数値が関係する複素関数の結果については、次のセクションで説明します。表 13.8では、複素数の引数の主値 Θ は、$-\pi < \Theta \le \pi$ の範囲にあります。さまざまな複素数の主値 Θ の定義を表 13.9に示します。

三角関数 atan2(y,x) は、$-\pi \le \mathrm{atan2}(y, x) \le \pi$ の範囲にあることに注意してください。通常、複素算術関数と複素関数では、実部または虚部のいずれかが値 –Inf、Inf,、または NaN であり、他方が通

表 13.8: 組み込み複素関数の構文と意味

Ch 構文	Ch における意味
sizeof(z)	8
abs(z)	sqrt(x^2+y^2)
fabs(z)	sqrt(x^2+y^2)
real(z)	x
imag(z)	y
complex(x,y)	$x+iy$
conj(z)	$x-iy$
carg(z)	$\Theta;\ \Theta = \text{atan2}(y,x)$
polar(z)	sqrt$(x^2+y^2)+i\Theta;\ \Theta = \text{atan2}(y,x)$
polar$(r, theta)$	$r\cos(theta)+ir\sin(theta)$
sqrt(z)	sqrt(sqrt(x^2+y^2))$(\cos\frac{\Theta}{2}+i\sin\frac{\Theta}{2});\ \Theta = \text{atan2}(y,x)$
sqrt(z,k)	sqrt(sqrt(x^2+y^2))$(\cos\frac{\Theta+2k\pi}{2}+i\sin\frac{\Theta+2k\pi}{2});\ \Theta = \text{atan2}(y,x)$
exp(z)	$e^x(\cos y+i\sin y)$
log(z)	$\log(\sqrt{x^2+y^2})+i\Theta;\ \Theta = \text{atan2}(y,x)$
log(z,k)	$\log(\sqrt{x^2+y^2})+i(\Theta+2k\pi);\ \Theta = \text{atan2}(y,x)$
log10(z)	$\frac{\log(z)}{\log(10)}$
log10(z,k)	$\frac{\log(z,k)}{\log(10)}$
pow(z_1,z_2)	${z_1}^{z_2} = e^{z_2 \ln z_1} = \exp(z_2 * \log(z_1))$
pow(z_1,z_2,k)	${z_1}^{z_2} = e^{z_2 \ln z_1} = \exp(z_2 * \log(z_1,k))$
ceil(z)	ceil$(x)+i$ ceil(y)
floor(z)	floor$(x)+i$ floor(y)
fmod$(z1,z2)$	$z;\quad \frac{z_1}{z_2} = k+\frac{z}{z_2},\ k \geq 0$
modf$(z1,\&z2)$	modf$(x_1,\&x_2)+i$ modf$(y_1,\&y_2)$
frexp$(z1,\&z2)$	frexp$(x_1,\&x_2)+i$ frexp$(y_1,\&y_2)$
ldexp$(z1,z2)$	ldexp$(x_1,x_2)+i$ ldexp(y_1,y_2)
sin(z)	$\sin x\cosh y+i\cos x\sinh y$
cos(z)	$\cos x\cosh y-i\sin x\sinh y$
tan(z)	$\frac{\sin z}{\cos z}$
asin(z)	$-i\log(iz+\text{sqrt}(1-z^2))$
asin(z,k)	$-i\log(iz+\text{sqrt}(1-z^2,k))$
asin$(z,k1,k2)$	$-i\log(iz+\text{sqrt}(1-z^2,k_1),k_2)$
acos(z)	$-i\log(z+i\text{sqrt}(1-z^2))$
acos(z,k)	$-i\log(z+i\text{sqrt}(1-z^2,k))$
acos$(z,k1,k2)$	$-i\log(z+i\text{sqrt}(1-z^2,k_1),k_2)$

次ページに続く

表 13.8: 続き

Ch 構文	Ch における意味
$\mathrm{atan}(z)$	$\frac{1}{2i}\log(\frac{1+iz}{1-iz})$
$\mathrm{atan}(z,k)$	$\frac{1}{2i}\log(\frac{1+iz}{1-iz},k)$
$\mathrm{atan2}(z1,z2)$	$\frac{1}{2i}\log(\frac{1+iz_1/z_2}{1-iz1_{/z_2}})$
$\mathrm{atan2}(z1,z2,k)$	$\frac{1}{2i}\log(\frac{1+iz_1/z_2}{1-iz1_{/z_2}},k)$
$\mathrm{sinh}(z)$	$\sinh x\cos y+i\cosh x\sin y$
$\mathrm{cosh}(z)$	$\cosh x\cos y+i\sinh x\sin y$
$\mathrm{tanh}(z)$	$\frac{\sinh x\cos y+i\cosh x\sin y}{\cosh x\cos y+i\sinh x\sin y}$
$\mathrm{asinh}(z)$	$\log(z+\mathrm{sqrt}(z^2+1))$
$\mathrm{asinh}(z,k)$	$\log(z+\mathrm{sqrt}(z^2+1,k))$
$\mathrm{asinh}(z,k1,k2)$	$\log(z+\mathrm{sqrt}(z^2+1,k_1),k_2)$
$\mathrm{acosh}(z)$	$\log(z+\mathrm{sqrt}(z+1)\mathrm{sqrt}(z-1))$
$\mathrm{acosh}(z,k)$	$\log(z+\mathrm{sqrt}(z+1,k)\mathrm{sqrt}(z-1,k))$
$\mathrm{acosh}(z,k1,k2)$	$\log(z+\mathrm{sqrt}(z+1,k_1)\mathrm{sqrt}(z-1,k_1),k_2)$
$\mathrm{atanh}(z)$	$\frac{1}{2}\log(\frac{1+z}{1-z})$
$\mathrm{atanh}(z,k)$	$\frac{1}{2}\log(\frac{1+z}{1-z},k)$

表 13.9: complex(x,y) の引数の主値 Θ $(-\pi<\Theta\leq\pi)$

Θ						
y 値	x 値					
	−x1	−0.0	0.0	x2	Inf	NaN
y2	atan2($y_2,-x_1$)	pi/2	pi/2	atan2(y_2,x_2)		
0.0	pi	0.0	0.0	0.0		
−0.0	pi	0.0	0.0	0.0		
−y1	atan2($-y_1,-x_1$)	−pi/2	−pi/2	atan2($-y_1,x_2$)		
Inf					Inf	
NaN						NaN

常の実数である複素数は得られません。この種の結果は、関数 **real**(z)、関数 **imag**(z)、および lvalue 経由の float 型のポインタ変数によってのみ明示的に得ることができます。詳細については、セクション 13.6を参照してください。

表 13.8の最初の 4 つの関数は実数を返します。

sizeof() 関数は、変数の整数、型指定子、または先行する式をバイト単位で返します。返されるデータ型は、符号なしの int 型です。引数が complex の場合は値 8 を返します。これは、complex の実部と虚部の 2 つの float を格納するのに必要なバイト数です。**abs**(z) 関数は、複素数の係数を計算します。入力値が float complex の場合、返されるデータ型は float です。入力値が double complex の場合、返されるデータ型は double です。入力型が complex 型のとき、関数 **fabs**(z) は関数 **abs**(z) と同じ動作をします。

関数 **real**(z) と関数 **imag**(z) は、それぞれ複素数の実部と虚部を返します。**real**(z) と **imag**(z) の結果は、常に float です。**real**() の引数のデータ型が double 以下の場合、入力データは float にキャストされます。**imag**() の引数のデータ型が double 以下の場合は、値ゼロが返されます。関数 real(z) と関数 imag(z) では、ゼロの符号は無視されます。たとえば、**real**(complex(−0.0,0.0)) は 0.0 を返します。

複素数は、2 つの実数から複素構築関数 **complex**(x,y) により作成できます。入力引数が float でない場合は、内部データ変換規則に従って、float にキャストされます。x または y のゼロの符号は、複素数に引き継がれます。

conj(z) 関数は z の複素共役 $\overline{z}$ を返します。点 $(x, -y)$ で表される複素数 $\overline{z}$ は、z を表す点 (x, y) の実軸における鏡像です。

polar() 関数は、主に複素数のデカルト表現と極座標表現間の変換の便宜のために実装されます。入力引数が 1 つだけある場合、それぞれが入力複素数の係数と引数である実部と虚部を有する複素数が返されます。入力引数が 2 つある場合、極座標形式の複素数 z が返されます。1 番目と 2 番目の入力引数は、それぞれ z の係数と引数です。極関数の定義 $re^{i\theta}$ に従って、r の負の値は有効です。

平方根関数 **sqrt**() では、2 つの引数がある場合、常に 1 番目の引数が複素数として処理されます。それが複素数でなく、複素数にキャストできない場合は、システムによって構文エラーメッセージが報告されます。2 番目の引数が整数でない場合は、内部データ変換規則に従って、整数値にキャストされます。複素平方根では、正弦関数と余弦関数の周期的性質のため、明確な分岐は 2 つのみです。一般に、n 番目の根を取ると、n 本の明確な分岐が存在します。**sqrt**() 関数を単一の複素引数で呼び出した場合は、既定の分岐値 0 が使用されます。

exp(z) 関数は、複素数 z の指数関数を計算します。

平方根関数のように、自然対数関数 **log**() には、複数の分岐があります。分岐数は、関数の 2 番目の引数によって与えられます。便宜上、**log10**() 関数は、複素値の 10 を底とする対数関数を計算します。

複素数の底を有する指数関数は **pow**() 関数で計算できます。これは、表 13.8に示す指数関数と対数関数により実現されます。対数関数の分岐が、**pow**() 関数の分岐を決定します。対応する実関数とは異なり、複素関数 **pow**() は常に明確に定義されます。**pow**(z1, z2) の 2 つの引数の片方が complex の場合、結果は complex です。これは、式 **exp**(z2∗**log**(z1)) の主分岐によって得られます。式 $y\,x$ の結果は、虚部がゼロである式 **pow**(complex(y,0.0), complex(x,0.0)) の実部に等しくなります。関数 **pow**(z1, z2, k) では、z1 と z2 は complex 以下の任意のデータ型が可能であり、k は整数です。関数 **pow**() に 3 つの引数がある場合、1 番目と 2 番目の引数は複素数として処理されます。z2 が整数の場合、すべての分岐で結果は同じになり、従って解は一意になります。

関数 **ceil**(z)、関数 **floor**(z)、および関数 **ldexp**(z1, z2) では、実部と虚部は、2 つの別々の実関数であるかのように処理されます。関数 **modf**(z1, &z2) と関数 **frexp**(z1, &z2) は、同じ方法で処理されます。

この 2 つの関数では、1 番目の引数のデータ型が complex のとき、2 番目の引数のデータ型は complex へのポインタでなければなりません。**fmod**(z1,z2) 関数は、z1/z2 の complex の剰余を計算します。

複素三角関数 **sin**(z)、**cos**(z)、**tan**(z)、複素双曲線関数 **sinh**(z)、**cosh**(z)、**tanh**(z) の値は一意です。ただし、複素逆三角関数 **asin**(z)、**acos**(z)、**atan**(z)、複素逆双曲線関数 **asinh**(z)、**acosh**(z)、**atanh**(z) では、特定の入力複素値で複数の分岐があります。これらの逆関数の 2 番目の引数は、分岐数を示します。関数 **asin**()、**acos**()、**asinh**()、および **acosh**() では、

2 番目と 3 番目の引数は、それぞれ関連する平方根関数と対数関数の分岐を指定します。関数 **atan2**() は、関数 **atan**() と同じように実装されます。

13.5.2　複素メタ数値を使用する複素関数の結果

複素算術演算と同じように、通常の複素関数の定義は、入力引数が複素メタ数値のときは有効ではない場合があります。入力引数として複素メタ数値を使用する組み込み複素関数の結果を、表 13.10に示します。

表 13.10: complex(0.0, 0.0)、ComplexInf、および ComplexNaN の複素関数の結果

function	z value and results		
	complex(0.0, 0.0)	ComplexInf	ComplexNaN
sizeof(z)	8	8	8
abs(z)	0.0	Inf	NaN
real(z)	0.0	NaN	NaN
imag(z)	0.0	NaN	NaN
conj(z)	complex(0.0, 0.0)	ComplexInf	ComplexNaN
polar(z)	complex(0.0, 0.0)	ComplexInf	ComplexNaN
sqrt(z)	complex(0.0, 0.0)	ComplexInf	ComplexNaN
exp(z)	complex(1.0, 0.0)	ComplexNaN	ComplexNaN
log(z)	ComplexInf	ComplexInf	ComplexNaN
log10(z)	ComplexInf	ComplexInf	ComplexNaN
ceil(z)	complex(0.0, 0.0)	ComplexInf	ComplexNaN
floor(z)	complex(0.0, 0.0)	ComplexInf	ComplexNaN
modf(z, &z2)	complex(0.0, 0.0)	complex(0.0, 0.0)	ComplexNaN
z2	complex(0.0, 0.0)	ComplexInf	ComplexNaN
frexp(z, &z2)	complex(0.0, 0.0)	ComplexInf	ComplexNaN
z2	complex(0.0, 0.0)	complex(0.0, 0.0)	ComplexNaN
ldexp(z, z2)	complex(0.0, 0.0)	ComplexInf	ComplexNaN
sin(z)	complex(0.0, 0.0)	ComplexNaN	ComplexNaN
cos(z)	complex(1.0, 0.0)	ComplexNaN	ComplexNaN
tan(z)	complex(0.0, 0.0)	ComplexNaN	ComplexNaN
	Note: tan(complex($\pi/2 + k*\pi$, 0.0)) = ComplexInf		
asin(z)	complex(0.0, 0.0)	ComplexInf	ComplexNaN
acos(z)	complex(pi/2, 0.0)	ComplexInf	ComplexNaN
atan(z)	complex(0.0, 0.0)	complex(pi/2, 0.0)	ComplexNaN
	Note: atan(complex(0.0, $\pm$1.0)) = ComplexInf; atan(ComplexInf, k) = complex(pi/2 + k*pi, 0.0)		
sinh(z)	complex(0.0, 0.0)	ComplexNaN	ComplexNaN
cosh(z)	complex(1.0, 0.0)	ComplexNaN	ComplexNaN
tanh(z)	complex(0.0, 0.0)	ComplexNaN	ComplexNaN
	Note: tanh(complex(0.0, $\pi/2 + k*\pi$)) = ComplexInf		
asinh(z)	complex(0.0, 0.0)	ComplexInf	ComplexNaN
acosh(z)	complex(0.0, pi/2)	ComplexInf	ComplexNaN
atanh(z)	complex(0.0, 0.0)	complex(0.0, pi/2)	ComplexNaN
	Note: atanh(complex($\pm$1.0, 0.0)) = ComplexInf; atanh(ComplexInf, k) = complex(0.0, pi/2 + k*pi)		

表 13.10では、 Ch における complex($\pm$0.0, $\pm$0.0) は、complex(0.0, 0.0) として処理されます。関

数の入力引数が ComplexNaN の場合、返される結果は、**sizeof**() 関数以外は常に ComplexNaN です。図 13.2に示すように、複素無限大は、実無限大 $\pm\infty$ とは異なります。

複素値の実部または虚部のいずれかが、表現可能な浮動小数点数の範囲外にある場合、それは ComplexInf になります。したがって、ComplexInf の絶対値は、実数 Inf です。ComplexInf の実部と虚部は NaN です。ただし、ComplexInf の共役は、依然として複素無限大です。**polar**(complex(0.0,0.0)) の結果は complex(0.0,0.0) と定義されます。これは、表 13.9に定義されているように、complex(0.0, 0.0) の主値 Θ は 0.0 に等しいからです。**polar**(ComplexInf) の結果は complex(Inf, Inf) と定義されています。したがって、z が complex(0.0,0.0) または ComplexInf と等しい場合、z = **polar**(real(polar(z)), imag(polar(z))) の同等性は依然として満たされます。実関数と同じように、ComplexInf の平方根は ComplexInf です。

実関数のように、**exp**(Inf) = Inf であり、**exp**(−Inf) = 0.0 です。ただし、値 ±Inf は、複素数にキャストされた場合は、どちらも ComplexInf になります。したがって、複素指数関数 **exp**(z) は、入力引数が ComplexInf のときは ComplexNaN です。入力引数が complex(0.0,0.0) または ComplexInf である複素対数関数 **log**(z) は、ComplexInf を返します。複素メタ数値を入力引数とする関数 **ceil**(z)、**floor**(z)、および **ldexp**(z1, z2) の実部と虚部は、2 つの個別の実関数と同じように処理されます。実関数と同じように、複素三角関数 **sin**(z)、**cos**(z)、および **tan**(z) は、入力引数が ComplexInf のときは未定義です。無理数 π は、コンピュータプログラムでは表現できません。

値 π があった場合、式 $\tan(k\pi + \pi/2)$ は ComplexInf を返します。実関数とは異なり、複素逆三角関数 **asin**(z) および **acos**(z) は、入力引数が ComplexInf のとき、ComplexInf を返します。逆関数 **tan**(z) のように、1 番目の入力値が ComplexInf のとき、関数 **atan**(z,k) には複数の分岐が存在します。定義に従って、**atan**($\pm i$) は ComplexInf と等しくなります。複素双曲線関数 **sinh**(z)、**cosh**(z)、**tanh**(z)、複素逆双曲線関数 **asinh**(z)、**acosh**(z)、**atanh**(z) の結果は、複素三角関数と複素逆三角関数と同じように実装されます。

表 13.11に、複素構築関数 complex(x,y) の結果を示します。

表 13.11: 0.0、$\pm\infty$、および NaN に対する関数 complex(x, y)

complex(x, y)						
x 値	y 値					
	−Inf	−y1	0.0	y2	Inf	NaN
Inf	ComplexInf	ComplexInf	ComplexInf	ComplexInf	ComplexInf	ComplexNaN
x2	ComplexInf	complex(x2, −y1)	complex(x2, 0.0)	complex(x2, y2)	ComplexInf	ComplexNaN
0.0	ComplexInf	complex(0.0, −y1)	complex(0.0, 0.0)	complex(0.0, y2)	ComplexInf	ComplexNaN
−x1	ComplexInf	complex(−x1, −y1)	complex(−x1, 0.0)	complex(−x1, y2)	ComplexInf	ComplexNaN
−Inf	ComplexInf	ComplexInf	ComplexInf	ComplexInf	ComplexInf	ComplexNaN
NaN	ComplexNaN	ComplexNaN	ComplexNaN	ComplexNaN	ComplexNaN	ComplexNaN

複素数の構築では、実部または虚部のいずれかが NaN の場合、その結果は、複素非数 (complex Not-a- Number) になります。同様に、どちらかが値 $\pm\infty$ の場合、結果は ComplexInf です。表 13.12に示す関数 polar(r, theta) では、係数が無限大の場合、与えられた複素数の引数が無限大の場合でも、結果の複素数は ComplexInf になります。

表 13.12: 0.0、±∞、および NaN に対する関数 polar(r, theta) の結果

polar(r, theta)						
r 値	theta 値					
	−Inf	−theta1	0.0	theta2	Inf	NaN
Inf	ComplexInf	ComplexInf	ComplexInf	ComplexInf	ComplexInf	ComplexNaN
r2	ComplexNaN	polar(r2, −theta1)	complex(r2, 0.0)	polar(r2, theta2)	ComplexNaN	ComplexNaN
0.0	ComplexNaN	complex(0.0, 0.0)	complex(0.0, 0.0)	complex(0.0, 0.0)	ComplexNaN	ComplexNaN
−r1	ComplexNaN	polar(−r1, −theta1)	complex(−r1, 0.0)	polar(−r1, theta2)	ComplexNaN	ComplexNaN
−Inf	ComplexInf	ComplexInf	ComplexInf	ComplexInf	ComplexInf	ComplexNaN
NaN	ComplexNaN	ComplexNaN	ComplexNaN	ComplexNaN	ComplexNaN	ComplexNaN

これは、**polar**(ComplexInf) = complex(Inf, Inf) の結果に対応します。これは、複素算術関数と複素関数では、実部または虚部のいずれかが値 –Inf、Inf,、または NaN であり、他方が通常の実数である複素数を得ることはないという規則にも従っています。指数関数 **exp**(z) 同様、表 13.13に示すように、関数 **pow**(z1,z2) は、2 番目の引数が ComplexInf のときは未定義です。

表 13.13: complex(0.0, 0.0)、ComplexInf、および ComplexNaN に対する関数 ComplexNaN

pow(z1, z2)						
z1 値	z2 値					
	complex(0.0, 0.0)	z2; ($\|y2\| < \infty$)			ComplexInf	ComplexNaN
		$-\infty < x2 < 0.0$	x2 = 0.0	$0 < x2 < \infty$		
complex(0.0, 0.0)	ComplexNaN	ComplexInf	ComplexNaN	complex(0.0, 0.0)	ComplexNaN	ComplexNaN
z1	complex(1.0, 0.0)	$z_1^{z_2}$	$z_1^{z_2}$	$z_1^{z_2}$	ComplexNaN	ComplexNaN
ComplexInf	ComplexNaN	complex(0.0, 0.0)	ComplexNaN	ComplexInf	ComplexNaN	ComplexNaN
ComplexNaN	ComplexNaN	ComplexNaN	ComplexNaN	ComplexNaN	ComplexNaN	ComplexNaN

z2 の虚部 y2 が有限値のとき、値 z1 が complex(0.0, 0.0) または ComplexInf の場合、関数の結果は、その実部 x2 の値に依存します。実関数と同じように、式 **pow**(complex(0.0,0.0)、complex(0.0,0.0))、**pow**(complex(0.0,0.0)、complex(0.0,y2))、**pow**(ComplexInf, complex(0.0,0.0))、および **pow**(ComplexInf, complex(0.0,y2)) は、ComplexNaN です。**pow**(0.0, Inf) = 0.0 であり、**pow**(0.0, –Inf) = Inf であり、Inf と –Inf はどちらも ComplexInf と見なされるので、**pow**(complex(0.0,0.0),ComplexInf) は ComplexNaN と定義されます。

表 13.14に、複素メタ数値に対する関数 **fmod**(z1,z2) の結果を示します。

表 13.14: complex(0.0, 0.0)、ComplexInf、および ComplexNaN に対する関数 fmod(z1, z2) の結果

fmod(z1, z2)				
z1 値	z2 値			
	complex(0.0, 0.0)	z2	ComplexInf	ComplexNaN
complex(0.0, 0.0)	ComplexNaN	complex(0.0, 0.0)	complex(0.0, 0.0)	ComplexNaN
z1	ComplexNaN	fmod(z1,z2)	z1	ComplexNaN
ComplexInf	ComplexNaN	ComplexNaN	ComplexNaN	ComplexNaN
ComplexNaN	ComplexNaN	ComplexNaN	ComplexNaN	ComplexNaN

13.6 複素数に関する lvalue

前に定義したように、lvalue は、代入ステートメントの左側に指定する任意のオブジェクトです。複素数に関連する有効な lvalue を、表 13.15に示します。

表 13.15: 複素数に関連する有効な lvalue

Case	lvalue の意味	例
1	単純な変数	`z = complex(1.0, 2);`
2	複素配列の要素	`zarray[i] = complex(1.0,2)+ComplexInf;`
3	complex ポインタ変数	`zptr = malloc(sizeof(complex) * 3;`
		`zptr = &z;`
4	複素数によって指し示されるアドレス	`*zptr = complex(1.0, 2) + z;`
5	complex ポインタ配列の要素	`zarrayptr[i] = malloc(sizeof(complex)*3;`
		`zarrayptr[i] = &z;`
6	complex ポインタ配列の要素によって指し示されるアドレス	`*zarrayptr[i] = complex(1.0, 2);`
7	複素変数の実部	`real(z) = 3.4;`
	複素変数の実部	`real(*zptr) = 3.4;`
	複素変数の実部	`real(*(zptr+1)) = 3.4;`
	複素変数の実部	`real(*zarrayptr[i]) = 3.4;`
8	複素変数の虚部	`imag(z) = complex(1.0, 2);`
	複素変数の虚部	`imag(*zptr) = 3.4;`
	複素変数の虚部	`imag(*(zptr+1)) = 3.4;`
	複素変数の虚部	`imag(*zarrayptr[i]) = 3.4;`
9	float ポインタ変数	`fptr = &z;`
		`fptr = zptr;`
	複素変数の実部へのポインタ	`*fptr = 1.0;`
	複素変数の虚部へのポインタ	`*(fptr+1) = 2.0;`

これらの lvalue のすべてに対して、代入演算+=、-=、*=、/=、増分演算++、および減分演算-- を適用できます。ケース 1 の単純な変数に加え、複素配列の要素を lvalue にすることができます。これは、表 13.15のケース 2 に該当します。ケース 3 では、complex へのポインタを lvalue として使用して、complex オブジェクトのメモリの取得またはメモリのポイントを行っています。

ケース 4 では、ポインタ *zptr* によってポイントされたメモリに、代入ステートメントの右側の式の値が割り当てられます。単一のポインタ変数に加えて、complex ポインタの配列を使用できます。

ケース 5 と 6 は、complex ポインタ配列の要素を使用してメモリにアクセスする方法を示しています。関数 real() は、rvalue またはオペランドとしてだけではなく、その引数のメモリにアクセスするための lvalue として使用することもできます。

ケース 7 では、**real**() の引数は、複素変数、または complex ポインタまたはポインタ式によってポイントされるアドレスでなければなりません。複素数定数または定数式は、rvalue またはオペランドであるときのみ、関数 **real**() の入力引数として使用できます。

ケース 8 では、complex の虚部は、関数 **imag**() によって、関数 **real**() と同じ方法でアクセスできます。複素数は内部的には 2 つの float を占有するので、この記憶域へは、関数 **real**() と **imag**() だけではなく、ケース 9 に示すように float へのポインタでもアクセスできます。ケース 9 では、変数 $fptr$ が float へのポインタです。

ケース 7〜9 では、右側の ±0.0、±Inf、および NaN を含む実数には、lvalue がフィルタ処理なしで正式に代入されます。したがって、complex(Inf, NaN)、complex(Inf,0.0) などの異常な複素数が作成される場合があります。たとえば、2 つの Ch コマンド **real**(z) = NaN と **imag**(z) = Inf は、z を complex(NaN,Inf) と等しくします。**real**(z) = −0.0 と imag(z) = NZero は、z に値 complex(−0.0, −0.0) を与えます。

13.7 ユーザーによる複素関数の作成

Ch では、ISO C の精神に基づいてユーザーの複素関数を作成できます。これを、以下の複素関数 $f(z_1, z_2)$ の計算によって説明します。

$$\mathbf{f}(z_1, z_2) = \frac{(4z_1 + 3 + i5) * \sin(z_1 * z_2) * e^{i2.5}}{z_1(z_2 - 2 - i2)} \tag{13.3}$$

複素関数 $f(z_1, z_2)$ は、以下のように Ch で簡単にプログラミングできます。

```
complex f(complex z1, complex z2) {
  complex z;
  z = (4*z1+3+complex(0,5))*sin(z1*z2)*polar(1, 2.5))/
       (z1*(z2-complex(2,2)));
  return z;
}
```

上でプログラミングした外部ガンマ関数を使用すると、以下のコマンドは、

```
printf("f(0, 0) = %f \n", f(0, 0));
printf("f(0, 1) = %f \n", f(0, 1));
printf("f(1, 1) = %f \n", f(1, 1));
printf("f(0, complex(2, 2)) = %f \n", f(0, complex(2,2)));
printf("f(1, complex(2, 2)) = %f \n", f(1, complex(2,2)));
```

以下の出力を生成します。

f(0, 0) = CcomplexNaN
f(0, 1) = ComplexNaN
f(1, 1) = complex(1.385598,-2.925680)
f(0, complex(2, 2)) = ComplexInf
f(1, complex(2, 2)) = ComplexInf

関数 $f(z_1, z_2)$ は、特異点 $z_2 = 2 + i2$ で ComplexInf を取得し、$f(0, z_2)$ は complex ゼロの complex ゼロによる除算になることに注意してください。

第14章　ポインタと配列

配列は一般的に使用されるプログラミング機能です。**配列**は、1 つまたは複数の次元を持つ要素で構成され、列、平面、立方体などを表します。配列の次元数は配列の**ランク**と呼ばれ、1 つの次元にある要素の数はその次元での配列の**エクステント**と呼ばれます。配列の**形状**はベクトルであり、ベクトルの各要素は配列の対応する次元のエクステントです。配列の**サイズ**は、配列のすべての要素を格納するために使用されるバイト数です。

本章では、ポインタを使用して配列の要素にアクセスする方法について最初に説明します。次に、1 次元および 2 次元配列にメモリを割り当てる方法について説明します。数学的な観点からは、この 2 種類の配列はベクトルと行列を表現する際にとても便利です。その後、C90 の関数に配列を渡すためのメカニズムについて説明します。可変長の多次元配列を C90 規格に準拠する方法で関数に渡すと、煩雑でエラーが起こりやすいことがわかります。

14.1　ポインタを使用した配列要素へのアクセス

配列はポインタと密接に結び付いています。ポインタを使用して配列にアクセスできるだけでなく、配列の変数名自体をポインタとして処理することができます。A1 が長さ 10 の int 型 1 次元配列であり、p が int へのポインタであると仮定すると、次に示す対話型の Ch シェルの実行において、3 つの方法で A1 の要素にアクセスできます。

```
> int i
> int A1[10], *p
> A1[3]=3       // method 1
> for(i=0; i<10; i++) printf("%d ", A1[i])
0 0 0 3 0 0 0 0 0 0
> *(A1+4)=4     // <==> A1[4]=4, method 2
> for(i=0; i<10; i++) printf("%d ", A1[i])
0 0 0 3 4 0 0 0 0 0
> p = A1
> *(p+5)=5      // <==> A1[5]=5, method 3
> for(i=0; i<10; i++) printf("%d ", A1[i])
0 0 0 3 4 5 0 0 0 0
>
```

9章で説明したポインタ演算に従って、p+5 は、A1 の 6 番目の要素を指します。次のステートメントの変数名 A1

```
*(A1+4)=4
```

は、実際には int へのポインタとして処理されます。2 次元または多次元配列では、配列の変数名は、配列へのポインタとして処理されます。たとえば、A2 が int 型のサイズ (3×4) の 2 次元配列であり、p が int へのポインタである場合、A2 の要素にアクセスする方法は次のとおりです。

```
> int i, j
> int A2[3][4], *p;
> A2[1][1]=3      // method 1
> for(i=0; i<3; i++) for(j=0; j<4; j++) printf("%d ", A2[i][j])
0 0 0 0 0 3 0 0 0 0 0 0
> p = A2
> *(p+1*4+2)=4    // <==>A2[1][2]=4, method 2
> for(i=0; i<3; i++) for(j=0; j<4; j++) printf("%d ", A2[i][j])
0 0 0 0 0 3 4 0 0 0 0 0
> *(*(A2+1)+3)=5  // <==>A2[1][3]=5, method 3
> for(i=0; i<3; i++) for(j=0; j<4; j++) printf("%d ", A2[i][j])
0 0 0 0 0 3 4 5 0 0 0 0
>
```

p+1*4+2 の値は、A2[1][2] で配列 A2 の 7 番目の要素のアドレスを指します。変数 A2 は、4 個の要素&A2[0][0]、&A2[1][0]、&A2[2][0] および&A2[3][0] を持つ配列へのポインタです。ポイント式 (A2+1) は、配列 A2 の 5 番目 A2[1][0] の要素のアドレスを指定します。したがって、*(*(A2+1)+3) は、配列要素 A2[1][3] と同等です。「2 次元配列の動的割り当て」も参考にしてください。

14.2 配列の動的割り当て

多くのアプリケーションにおいて、工学と科学の分野では特に、配列または行列のサイズはプログラムを実行する時点で初めて明らかになります。そのため、大きな固定サイズの配列を宣言するのではなく、配列の動的な割り当て機能を使用すると効率的です。このセクションでは、1 次元および 2 次元配列の動的割り当てを実装する方法について説明します。 9章で説明した標準関数 **malloc()**、**calloc()**、および **realloc()** を使用して、配列のメモリを動的に割り当てることができます。動的に割り当てたメモリが不要になった場合は、**free()** 関数を使用して割り当てを解除できます。

14.2.1 1 次元配列の動的割り当て

通常、1 次元配列はベクトルを表すときに使用します。たとえば行ベクトルは、n を正の整数値とする $(1 \times n)$ の行列であり、列ベクトルは $(n \times 1)$ の行列です。どちらも 1 次元配列の形で記述できます。

例として、ベクトル**A1** が double 型であり、その長さ vectLen は実行時に決定するとします。次のコード例は、**malloc()** 関数を使用して A1 用のメモリを動的に割り当てる方法を示しています。

```
double *A1;
/* ... source code to obtain vectLen at runtime */
A1 = (double *)malloc(vectLen*sizeof(double));
```

```
if(A1 == NULL) {
  fprintf(stderr, "ERROR: %s(): no enough memory\n", __func__);
  exit(1);
}
/* ... source code to handle vector A1 */
free(A1);
/* ... source code no longer use A1 */
```

変数 `A1` を double へのポインタとして宣言します。実行時に `vectLen` の値を取得した後、`vectLen * sizeof(double)` バイトのメモリを動的に割り当てます。`A1` を 1 次元配列として扱い、`A1` という名前を使用して配列の要素にアクセスすることができます。

```
> int i
> double *A1
> A1 = (double *)malloc(10*sizeof(double))
40070280
> A1[5] = 10      // method 1
> for(i=0; i<10; i++) printf("%1.1f ", A1[i])
0.0 0.0 0.0 0.0 0.0 10.0 0.0 0.0 0.0 0.0
> *(A1+6) = 20;  // <==> A1[6]=20, method 2
0.0 0.0 0.0 0.0 0.0 10.0 20.0 0.0 0.0 0.0
> p = A1
40070280
```

上記の例は、配列の要素にアクセスするための 2 つの異なる方法を示しています。`vectLen` 個の要素を持つ配列 `A1` では、`A1[i]` または`*(A1+i)` という形式の添字 i が取り得る値の範囲は、0 から `vectLen-1` までになります。要素 `A1[vectLen]` にアクセスする試みは、配列 `A1` の境界外のメモリを指すため無効です。

1 次元配列を使用して 2 次元の行列を表すこともできます。たとえば、ポインタ `p` は、サイズ $(n \times m)$ の 2 次元配列に対してメモリを割り当てることができます。行列 $(n \times m)$ の要素 `(i, j)` にポインタ間接操作`*(p+i*m+j)` によってアクセスできます。次の例では、`n` は 3、`m` は 4 です。

```
> int i
> double *A1, *p
> A1 = (double *)malloc(3*4*sizeof(double))
40070280
> p = A1
40070280
> *(p+1*4+2) = 30;    // assigned 30 to element (1,2)
>
```

14.2.2　2 次元配列の動的割り当て

数学的な観点からは、$(m \times n)$ の行列または配列とは、縦方向の `m` 個の行と横方向の `n` 個の列で構成される長方形のブロックに一連の数値を配置したものです。プログラミングの観点からは、行列の

各行は連続するメモリブロックに配置されたメモリブロックです。このセクションでは、2 次元配列の動的割り当てを行う 2 つの方法について説明します。最初の方法では、行列に割り当てるメモリを単一の連続したブロックに配置します。2 番目の方法では、行列の各行を連続したメモリブロックに配置しますが、行列全体は連続したメモリブロックに配置されません。

A2 はサイズ $(m \times n)$ の 2 次元配列であるとします。m と n の値は実行時に決定します。次のコード例は、配列 A2[m][n] に単一の連続したメモリを動的に割り当てる方法を示しています。

```
int i;
double **A2;
/* ... source code to obtain m and n at runtime */
A2 = (double **)malloc(m * sizeof(double*));
if(A2 == NULL) {
  fprintf(stderr, "ERROR: %s(): no enough memory\n", __func__);
  exit(1);
}
A2[0] = (double *)malloc(m * n * sizeof(double));
if(A2[0] == NULL) {
  fprintf(stderr, "ERROR: %s(): no enough memory\n", __func__);
  exit(1);
}
for(i = 1; i < m; i++) {
  A2[i] = A2[0] + i * n;
}
/* ... source code to handle vector A2 */
free(A2[0]);
free(A2);
/* ... source code no longer use A2 */
```

最初に、A2 を double へのポインタを参照するポインタとして宣言します。m と n の値を取得した後、double への m 個のポインタに対してメモリが A2 に割り当てられます。したがって、A2 は、図 14.1に示すように、m 個の要素を持つ 1 次元ポインタ配列と見なすことができます。

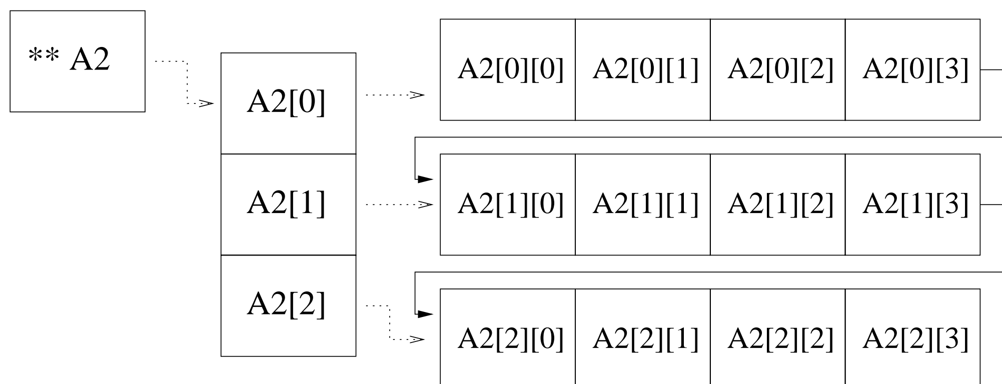


図 14.1: 2 次元配列の動的割り当て (方法 1)

次に、`m * n` 個の要素に対して、`m * n * sizeof(double)` バイトの単一の連続したメモリを割り当てます。各ポインタ `A[i]` は、このブロックのセグメントを指します。したがって、`A2` は、2 次元配列になります。次の例では、サイズ (3×4) の配列の要素に 2 つの異なる方法でアクセスしています。

```
> int i, j
> double **A2
> A2 = (double **)malloc(3 * sizeof(double*)); // with size (3 X 4)
4006cdc0
> for(i=0; i<3; i++) printf("%p ", A2[i])
00000000 00000000 00000000
> A2[0] = (double *)malloc(3 * 4 * sizeof(double));
> A2[1] = A2[0] + 4                          // A2[i] = A2[0] + i * n
> A2[2] = A2[0] + 2*4                        // A2[i] = A2[0] + i * n
> for(i=0; i<3; i++) printf("%p ", A2[i])    // 1-dimension of pointer
4007bee8 4007bf08 4007bf28
> for(i=0; i<3; i++) for(j=0; j<4; j++) printf("%1.1f ", A2[i][j])
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
> A2[1][1]=3            // method 1
> *(*(A2+1)+2)=4        // <==> A2[1][2]=4, method 2
> for(i=0; i<3; i++) for(j=0; j<4; j++) printf("%1.1f ", A2[i][j])
0.0 0.0 0.0 0.0 0.0 3.0 4.0 0.0 0.0 0.0 0.0 0.0
> pp = A2
4006cdc0
```

`m * n * sizeof(double)` バイトのメモリブロックを `A2` に割り当てた後、`A2[i][j]` または `*(*(A2+i)+j)` という形式の添字 `i` と `j` が取り得る値の範囲は、それぞれ 0 から `m-1` までと 0 から `n-1` までになります。`A2[m][n]` へのアクセスの試みは無効です。

次のコード例は、2 次元配列 `A2` を動的に割り当てる 2 番目の方法を示しています。行列の各行のメモリを別々に割り当てます。したがって、各行のメモリブロックは連続しない可能性があります。配列 `A2` のメモリレイアウトを図 14.2に示しています。

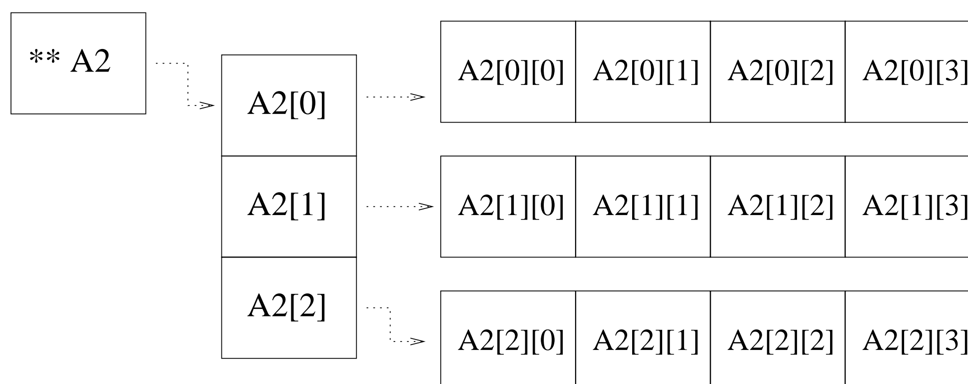


図 14.2: 2 次元配列の動的割り当て (方法 2)

```
int i;
double **A2;
/* ... source code to obtain m and n at runtime */
A2 = (double **)malloc(m * sizeof(double*));
if(A2 == NULL) {
  fprintf(stderr, "ERROR: %s(): no enough memory\n", __func__);
  exit(1);
}
for(i = 0; i < m; i++) {
  A2[i] = (double *)malloc(n * sizeof(double));
  if(A2[i] == NULL) {
    fprintf(stderr, "ERROR: %s(): no enough memory\n", __func__);
    exit(1);
  }
}
/* ... source code to handle vector A2 */
for(i = 0; i < m; i++)
  free(A2[i]);
free(A2);
/* ... source code no longer use A2 */
```

このセクションで説明した 2 次元配列を動的に割り当てるための 2 つの方法の大きな違いは、2 番目の方法では行列の行ごとに **malloc**() 関数を呼び出していることです。

14.3 固定長の 1 次元および多次元配列を渡す処理

14.3.1 1 次元配列

配列を関数に渡すとき、実際に渡すのは配列の最初の要素のアドレスだけです。呼び出された関数では、この引数はポインタ型のローカル変数です。プログラム 14.1では、それぞれが double データ型の 10 個の要素を持つ 1 次元配列 `d1` と `d2` を加算し、その結果を `oneDadd()` 関数により配列 `d1` に要素ごとに格納します。

```
int main() {
  double d1[10], d2[10];
  double d3[5], d4[5];
  void oneDadd(double dd1[], double *dd2, int n);
  oneDadd(d1, d2, 10);
  oneDadd(d3, d4, 5);
}

void oneDadd(double dd1[], double *dd2, int n) {
  int i;
  for(i=0; i<=n-1; i++)
    dd1[i] += dd2[i]; /* the same as *(dd1+i) += *(dd2+i) */
}
```

プログラム 14.1: 可変長の 1 次元配列を関数に渡す処理

プログラム 14.1の関数定義 `void oneDadd(double dd1[], *dd2, int n)` では、`dd1` を double 型の配列変数として、`dd2` を double へのポインタ変数として定義しています。そのプログラムに示したように、可変長の 1 次元配列をこの関数に渡すことができます。配列 `d1` と `d2` のエクステントはどちらも 10 であり、配列 `d3` と `d4` のエクステントは 5 です。通常、呼び出された関数は配列のサイズを取得できないので、`oneDadd()` 関数の配列のサイズは、プログラムでパラメータ `n` によって渡されます。ただし、文字列は特殊なケースです。**strlen**() 関数でサイズを簡単に計算できるように、文字列はゼロで終了します。`dd1[-2]` および `dd2[20]` の式は、それぞれ`*(dd1-2)` および`*(dd2+20)` と同等です。プログラム 14.1の `oneDadd()` 関数で使用した場合、これらの式は渡された配列の境界外にあるオブジェクトを参照します。仮配列の宣言内でエクステントを指定していないため、ステートメント `dd1[-2] += dd2[20+n]` は、 `oneDadd()` 関数内に記述されていれば問題を含んでいても構文上は有効です。この種のステートメントに対して警告やエラーメッセージを生成することはできません。呼び出された関数内の仮配列の各要素が、呼び出し元の関数内の実配列の配列境界内にあることを確認するのは、プログラマの責任です。

ただし、呼び出し元の関数内で渡す配列のエクステントが既知の場合は、関数内の仮配列の引数の宣言でエクステントを指定できます。たとえば、プログラム 14.2では、`dd1` と `dd2` をどちらも 10 個の要素を持つ配列の変数として定義しています。

```
int main() {
  double d1[10], d2[10];
  double d3[5], d4[5];
  void oneDadd(double  dd1[10], double dd2[10], int n);
  oneDadd(d1, d2, 10);   /* OK */
  oneDadd(d3, d4, 5);    /* WARNING: incompatible dimensions */
}

void oneDadd(double  dd1[10], double dd2[10], int n) {
  int i;
  for(i = 0; i <=n-1; i++)
    dd1[i] += dd2[i];
}
```

プログラム 14.2: 固定長の 1 次元配列を関数に渡す処理

oneDadd(d3, d4, 5) の関数呼び出しによって呼び出されたとき、渡される配列 d3 と d4 のエクステントが、dd1 と dd2 の仮定義のエクステントと一致しないために、システムによって 2 つの警告メッセージが生成される可能性があります。さらに、仮引数にエクステントを指定しているので、dd1[-2] += dd2[20] というステートメントを oneDadd() 関数内で使用した場合、Ch での実行時に配列境界エラーによるエラーメッセージが生成されます。システムクラッシュの可能性を避けるために、配列インデックスがゼロより小さい場合、Ch では配列インデックスとして下限ゼロが使用されます。同様に、インデックスが仮配列の上限より大きい場合は、仮配列の上限がインデックスとして使用されます。したがって、割り当てステートメント dd1[-2] += dd2[20] は、ステートメント dd1[0] += dd2[9] として処理されます。
関数定義 oneDadd(double dd1[10], dd2[10], int n) 内の dd1[10] と dd2[10] は、エクステントを指定して宣言されていますが、この場合も呼び出された関数に渡されるのはポインタだけです。

14.3.2 固定長の多次元配列

前のセクションで説明したように、C では 1 次元配列を関数に簡単に渡すことができます。このセクションでは、固定形状の多次元配列を関数に渡す方法について説明します。

プログラム 14.3で twoDadd() 関数を使用して 2 つの 2 次元配列を加算するときに、関数に 2 次元配列を渡す例について検討します。固定長の 2 次元配列を関数に渡す場合、関数の引数では、配列定義のパラメータに配列の 2 番目の次元である列数を含める必要があります。関数の仮配列引数に使用する 3 つの異なる形式をプログラム 14.3に示します。

```
int main() {
  double d1[4][5], d2[4][5], d3[4][5];
  double d4[3][5], d5[3][5], d6[3][5];
  double d7[4][6];
  void twoDadd(double (*dd1)[5], double dd2[][5], double dd3[4][5],
             int n, int m);
  twoDadd(d1, d2, d3, 4, 5); /* OK */
  twoDadd(d4, d5, d6, 3, 5); /* WARNING: incompatible first dimension
                                d6[3][5] != dd3[4][5] */
  twoDadd(d7, d2, d3, 4, 5); /* WARNING: incompatible second dimension
                                d7[4][6] != (*dd1)[5] */
}

void twoDadd(double (*dd1)[5], double dd2[][5], double dd3[4][5],
             int n, int m) {
  /* dd1[n][m] = dd2[n][m] + dd3[n][m] */
  int i, j;

  for(i=0; i<=n-1; i++)
    for(j=0; j<=m-1; j++)
      dd1[i][j] = dd2[i][j]+dd3[i][j];
}
```

プログラム 14.3: 固定長の 2 次元配列を関数に渡す処理

次のような宣言を使用します。

```
void twoDadd(double (*dd1)[5], double dd2[][5], double dd3[4][5],
             int n, int m)
```

宣言の形式は異なりますが、dd1、dd2@および dd3 はすべて、double データ型の 5 つの要素を持つ**配列へのポインタ**として定義されています。1 次元配列とは異なり、変数 dd1 と変数 dd2 の内部データ構造は同じであり、これらの変数は両方とも実配列の行数を無視します。ただし、変数 dd3 に対応する呼び出し元の関数の引数の行数が 4 でない場合、システムの警告メッセージが生成される可能性があります。通常、配列を関数に渡すときは、実配列のランクが関数定義および関数プロトタイプの仮配列引数のランクと一致している必要があります。そうでない場合、システムの警告メッセージが生成される可能性があります。配列の次元のエクステントを関数定義内または関数プロトタイプ内で指定する場合、実配列のエクステントが、仮定義内の配列の対応するエクステントと一致している必要があります。そうでない場合、やはりシステムの警告メッセージが生成される可能性があります。関数定義内の配列の形状は、関数プロトタイプ内の配列の形状と同じである必要があります。そうでない場合は構文エラーになります。例外は、可変長の 1 次元配列の宣言です。たとえば、次の 2 つの関数プロトタイプは互換性があると見なされます。

```
void funct(double *d);
void funct(double d[]);
```

プログラム 14.3の twoDadd(d4, d5, d6, 3, 5) の関数呼び出しでは、最初の行の次元のエクステントが仮引数 dd3 と異なる配列 d6[3][5] に対する警告メッセージが生成される可能性があります。配列 d7[4][6] についても、列数が仮定義と異なるために警告メッセージが生成される可能性があります。行数が異なる配列 d1、d2、d4、および d5 は、twoDadd() 関数に正常に渡すことができ、警告メッセージは生成されません。

同じ方法で、2 次元より高次の多次元配列を処理できます。通常は、配列の第 1 次元のサイズだけを変更できます。その他すべてのサイズは、関数定義内および関数プロトタイプ内に指定します。プログラム 14.4は、3 次元配列を関数に渡す方法を示しています。

```
int main() {
  double d1[3][5][7], d2[3][5][7], d3[3][5][7];
  void threeDadd(double (*dd1)[5][7], double dd2[][5][7],
                 double dd3[3][5][7], int n, int m, int r);
  threeDadd(d1, d2, d3, 3, 5, 7);   /* d1 = d2 + d3 */
}

void threeDadd(double (*dd1)[5][7], double dd2[][5][7],
               double dd3[3][5][7], int n, int m, int r) {
  /* dd1[n][m][r] = dd2[n][m][r] + dd3[n][m][r] */
  int i, j, k;

  for(i=0; i<=n-1; i++)
    for(j=0; j<=m-1; j++)
      for(k=0; k<=r-1; k++)
        dd1[i][j][k] = dd2[i][j][k]+dd3[i][j][k];
}
```

プログラム 14.4: 固定長の 3 次元配列を関数に渡す処理

このプログラムでは、配列引数を 3 つの異なる構文形式で記述しています。プログラム 14.4では、関数呼び出し `threeDadd(d1, d2, d3, 4, 5, 6)` は、配列 `d2` と `d3` の各要素の合計を、配列 `dd1` の対応する要素に格納します。

関数内の配列パラメータは配列へのポインタとして処理されるので、配列へのポインタは、関数の実引数としても使用できます。これをプログラム 1 14.5に示します。

```
void funct(double dd[][7]){ }
int main() {
  double d[3][5][7], (*dp)[7], (*dp2)[7];
  dp = d[1];              /* dp = &d[1][0][0]; dp = d+1; dp = *(d+1); */
  funct(dp);              /* funct(&d[1][0][0]); funct(d[1]); */
                          /* funct(d+1); funct(*(d+1)); */
  dp[2][3] = dp[3][5]+6;  /* treat dp as an array */
  dp2 = malloc(sizeof(double)*5*7);
  funct(dp2);
}
```

プログラム 14.5: ポインタと配列へのポインタを実引数として使用し、関数に配列を渡す処理

プログラム 14.5では、変数 `dp` は 7 個の要素を持つ配列へのポインタであり、`d[1]` は 5x7 個の要素を持つ配列です。割り当てステートメント `dp = d[1]` は、式 `dp[i][j]` と式 `d[1][i][j]` が同じオブジェクトを参照するように、5x7 配列の開始アドレスで `dp` を指します。プログラム 14.5では、ステートメント `dp = d[1]` は、以下の割り当てステートメントのいずれかと置き換えることができます。

```
dp = &d[1][0][0];
dp = d+1;
```

```
dp = *(d+1);
```

ここで、&d[1][0][0] は、配列 d の最初の要素 d[1][0][0] のアドレス、d+1 は、5x7 個の要素を持つ配列へのポインタ、*(d+1) は、d[1][0][0] から d[1][4][6] までの 5x7 個の要素を持つ配列です。同様に、ステートメント funct(dp) を以下のプログラミングステートメントのいずれかに置き換えた場合、

```
funct(&d[1][0][0]);
funct(d[1]);
funct(d+1);
funct(*(d+1));
```

funct() 関数内で参照される要素 dd[i][j] がメインルーチンで配列 d の有効範囲内にある限り、結果は同じになります。

配列は、C では連続したメモリセグメントで構成されます。配列へのポインタが指すメモリは、メモリ割り当て関数 **malloc**()、**calloc**()、および **realloc**() により、動的に割り当てることができます。ポインタの間接指定を使用して、n 次元の配列へのポインタを (n+1) 次元の配列として処理できます。プログラム 14.5の変数 dp2 でこの処理を確認できます。変数 dp2 は、double データ型の 7 個の要素を持つ配列へのポインタです。dp2 のメモリは動的に割り当てられ、dp2 は dp と同じ方法で funct() 関数に渡されます。

同様に、データ型が異なる配列を関数に渡すことができます。たとえば、次のコード例は、データ型が異なるポインタ配列を funct() 関数に渡す方法を示しています。

```
char *c[]={"strings", "with different", "length", ""};;
int **i[2][4];
float ***f[3][5][7];
int funct(char *cc[], int **ii[2][4], float ***ff[3][5][7]);
funct(c, i, f);
```

c は char へのポインタ配列、i は 2x4 個の int 要素を持つ二重ポインタ配列、f は 3x5x7 個の float 要素を持つ三重ポインタ配列です。char へのポインタ配列は可変長の文字列を格納できるので、システムのプログラミングで有用です。

第15章　可変長配列

C 言語の配列はポインタと密接に結び付いています。配列の変数をポインタとして扱う C の手法は、非常に洗練されたシステムプログラミングの手法です。

C は数値計算用として設計されていないため、多くの場合、C で多次元配列を扱うのは面倒です。たとえば、C が簡潔性と明瞭性で知られているのに比べ、C90 で可変長の配列を関数に渡す処理は直感的に把握できず、理解が難しく、非常に面倒です。可変長配列は、FORTRAN では当初から使用可能でした [1]。FORTRAN を体験した科学技術計算プログラマは、多くの場合、可変長配列を C で扱えないことに失望します。

可変長配列 (VLA) を C に追加することは、科学および工学のアプリケーション開発用の主要なプログラミング言語として C が発展するための重要なステップです。プログラム実行時にのみサイズが確定する可変長配列は、C99 と Ch で追加されています。本章では、可変長配列の詳細について説明します。

Ch では、以下のコードに示す 4 種類の可変長配列があり、形状無指定配列、形状引継ぎ配列、形状引継ぎ配列へのポインタ、および参照配列と呼ばれています。

```
int funct(int a[&][&], int (*b)[:], int c[:][:], int n, int m) {
  int (*d)[:] = a;
  int e[n][m];
}
```

ここで、a は参照配列、b と c は形状引継ぎ配列、d は形状引継ぎ配列へのポインタ、e は形状無指定配列です。Ch は、数値計算用に上限値と下限値を明示する配列で C を拡張しています。次のコードは、上限値と下限値を明示する配列の例です。

```
int funct(int a[1:5], int b[1:][1:], int n, int m) {
  int c[3][1:3];
  int d[n:m];
}
```

ここで、a は 1〜5 までの添字範囲を持つ固定長配列です。b は単位オフセットした 2 次元の形状引継ぎ配列へのポインタです。添字範囲が固定されている 2 次元配列 c の先頭の次元の添字範囲は 0〜2 で、配列 c の 2 番目の次元の添字範囲は 1〜3 です。変数 d は、添字範囲の指定がない配列であり、形状無指定配列でもあります。C では配列内の先頭の要素は、a[0] というように、インデックス 0 で示されます。C とは対照的に、FORTRAN 77 では配列内の先頭の要素は、a[1] というように、インデックス 1 で示されます。

上限値と下限値を明示する配列では、配列の要素は任意のインデックス番号で始めることができます。上限値と下限値を明示する配列には、多くの用途があります。たとえば、多項式 $a_0 + a_1x +$

$a_2x^2 + ... + a_nx^n$ の係数 a_i は、インデックス 0 から始まる配列 a[0:n] を使用して適切に処理できます。これに対し、$i = 1$ から N までの N 個のデータ x_i から成るベクトルではインデックス 1 から始まるた配列 x[1:N] が必要です。

15.1 記憶期間と配列の宣言

15.1.1 オブジェクトの記憶期間

記憶期間は、オブジェクトの存続期間を決定します。外部または内部リンケージを使用して宣言されているか、記憶クラス指定子 static によって宣言されている配列は、**静的な記憶期間**を持つ配列です。このような配列では、記憶域が確保されており、格納された各要素の値は 1 回だけ初期化されます。配列内に存在する各要素は、プログラム全体の実行で最後に格納された値を保持します。静的記憶期間を持つ配列の形状は、関数 main() が実行される前に解決される必要があります。したがって、次のサンプルプログラムに示すように、静的記憶期間を持つ配列定義の各エクステントを、ゼロ以上の値を含む整数定数式にする必要があります。

```
int n = 5;
int a[4][5], aa[3] = {1,2,3};
extern int b[6][7], c[8], d[][9], e[]; /* d and e are incomplete */
/* complete shape for d and e in the following definition */
int  b[6][7], c[8], d[4][9], e[10], ee[2][3] = {1,2,3,4,5,6};
int main(){
  static int s[4], ss[2+3] ={1,2,3,4,5};
  extern int a[4][5];
  extern int b[6][7], c[8], d[][9], e[];
}
```

関数内または入れ子にされた関数内でリンケージや記憶クラス指定子 static を使用しないで宣言されている配列は、自動記憶期間を持つ配列です。このような配列では、関連付けられたブロックへ通常のエントリがあるたびに、配列の新しいインスタンス用に記憶域が確保されます。自動記憶期間を持つ配列に初期化が指定されていれば、通常のエントリがあるたびに初期化が実行されます。ただし、goto、continue、break、return などのステートメントによって何らかの方法で外側のブロックの実行が終了した場合は、配列用の記憶域が確保されるとは限りません。たとえば、次のコードは自動記憶期間を持つ配列です。

```
int n = 5;
void funct1(){
  int m = 6;
  int a[4][5], aa[3] = {1,n,m};      /* n==5, m==6 */
  void funct2(){
    int s[4], ss[2+3] ={1,2,3,n,m}; /* n==5, m==6 */
  }
}
```

このサンプルコードでは、配列の形状は、整数定数式によってエクステントごとに完全に指定されています。プログラムの実行時に、配列が宣言されているブロックにエントリが発生するタイミングで自動記憶期間を持つ配列にメモリ領域が割り当てられるため、ブロックまたは関数が呼び出されるたびに配列の長さを変更できることが望まれます。サイズがプログラムの実行時に確定される**配列を可変長配列** (VLA) と呼びます。

15.1.2　配列の宣言

変数は、次の形式で宣言できます。

$$\texttt{T D1} \tag{15.1}$$

ここで、T は `int` 型などのデータ型を指定する宣言指定子を含みます。D1 は配列変数の識別子（以下 "*ident*" で表します）を含む宣言子です。配列の宣言では、"[ ]"で式または ":" を囲みます。D1 が次の形式

$$\mathrm{D}[expression_{opt}] \tag{15.2}$$

で指定される場合、*ident* が表す配列の要素に指定される型は、T で指定されるデータ型となります。サイズ式がない場合、配列型は不完全型となります。配列のサイズを指定する**式**を "[ ]"で囲む場合、式は整数型である必要があります。式が整数定数式の場合は、サイズはゼロより大きい値を指定する必要があります。配列のサイズ式が定数式でない場合は、式はプログラム実行時に評価されますが、ゼロ以上の値で評価される必要があります。配列型は**形状無指定配列型**です。サイズ式が整数定数式で、要素の型が固定サイズの場合は、配列型は**固定長配列型**になります。D1 が次の形式

$$\mathrm{D}[:] \tag{15.3}$$

で指定される場合、ident が表す配列の要素に指定される型は、T で指定されるデータ型となります。この配列は、**形状引継ぎ配列型**と呼ばれます。配列の形状はプログラム実行時に引き継がれます。下限値を指定して配列を宣言する場合は、次の 2 つのいずれかの形式で行う必要があります。

$$\texttt{T D[lower:upper]} \tag{15.4}$$

$$\texttt{T D[lower:]} \tag{15.5}$$

ここで、`T` は `int` などの型を指定する宣言指定子、`D` は識別子 *ident* を含む宣言、`lower` は配列の下限値、`upper` は配列の上限値をそれぞれ表します。`lower` および `upper` の式は、整数型の式である必要があります。固定長配列へのポインタを宣言する場合は、次の形式で行う必要があります。

$$\texttt{T (*D)[expr]} \tag{15.6}$$

ここで、`T` には型を指定する宣言指定子を指定します。`D` は識別子 *ident* を含む宣言子です。式 `expr` は、整数定数式でなければなりません。形状引継ぎ配列へのポインタを宣言する場合は、次の 2 つのいずれかの形式で行う必要があります。

```
T (*D)[:]          (15.7)
T (*D)[lower:]     (15.8)
```

配列の下限値を示す式 `lower` は、整数定数式でなければなりません。参照配列を宣言する場合は、次の形式で行う必要があります。

```
T D[&]             (15.9)
```

参照配列を使用すると、データ型の異なる配列を関数に渡すことができます。参照配列は、関数パラメータのスコープ内または typedef 宣言のみで宣言する必要があります。

可変長配列型には、形状引継ぎ配列、形状引継ぎ配列へのポインタ、形状無指定配列、および参照配列の 4 つがあります。次のコード例を使用して、これらのさまざまな配列定義の概念を詳しく説明します。

```
void funct(int a[:][:], (*b)[:], c[&], d[], e[1:], n, m){
/* a: assumed-shape array */
/* b: pointer to array of assumed-shape */
/* c: array of reference */
/* d: incomplete array completed by function call */
/* e: assumed-shape array with explicit lower bound */
/* n, m: int */
  int f[4][5];     // f: fixed-length array
  int g[n][m];     // g: deferred-shape array
  int (*h)[4];     // h: pointer to array of 4 elements.
  int (*i)[:];     // i: pointer to array of assumed-shape
  extern int j[]; // j: incomplete array completed by external linkage
  int k[] = {1,2};// k: incomplete array completed by initialization
}
```

2 つの配列型に互換性を持たせるには、両方に互換性のある要素型が必要です。さらに、どちらにもサイズ指定子が含まれ、それらが整数定数式である場合は、両方のサイズ指定子に同じ定数値を指定する必要があります。どちらかのサイズ指定子が変数である場合、2 つのサイズは、プログラム実行時に同じ値であると評価されます。互換性を持つことが要求されるコンテキストで 2 つの配列型が使用される場合、プログラム実行時に 2 つのサイズ指定子が異なる値であると評価されたときは、配列の動作は未定義です。

15.2 形状無指定配列

15.2.1 制約と意味

形状無指定配列型のサイズはプログラム実行時に取得され、サイズの値はゼロより大きい必要があります。宣言を含むブロックの実行が完了するまで、形状無指定配列型のサイズは変化してはならないことになっています。したがって、形状無指定配列では、サイズ式の少なくとも 1 つは整数型の非定数式です。サイズ式で使用する変数は、事前に宣言する必要があります。たとえば、次のコードに示す配列 a、b、c、d、および e は形状無指定配列の有効な宣言ですが、配列 f、g、および h は有効な宣言ではありません。

```
int N1;
extern int N;
void funct1(int n, m){
  int i = 8*n;
  int j = 0, k = -9;
  int a[i][4];        /* OK */
  int b[3][m];        /* OK: mix fixed-extent with deferred-extent */
  int c[n*m][n];      /* OK */
  int d[funct2(n)][3*funct2(i)]; /* OK */
  int e[N][N1*n];                /* OK */
  int f[M];          /* ERROR: M has not been defined yet */
  int g[j], gg[0];   /* ERROR: zero size */
  int h[k], hh[-9];  /* ERROR: negative size */
}
int funct2(int i)
{ return i*i;}
int N, M;            /* define N and M */
```

アプリケーション例として、形状無指定配列に対して異なった要素数をもつプロットを以下のように生成することができます。

```
int n;
scanf("%d", &n);
double x[n], y[n];
...
plotxy(x, y, n, "title", "x", "y");
```

ここで、変数 `n` は形状無指定配列 `x` と `y` の要素数を含んでいます。変数 `n` の値は、ユーザーの入力で与えられます。

形状無指定配列は、関数内や入れ子にされた関数内の変数などのブロックスコープ内で宣言する必要があります。ブロックスコープ内で `static` 記憶クラス指定子を使用して宣言されている配列は、形状無指定配列として宣言することはできません。ファイルまたはプログラムのスコープで形状無指定配列を宣言した場合、その宣言の動作は未定義です。次に例を示します。

```
#include <stdio.h>
void funct1(int n, m){
  int funct2(int n, i){
    int a[n][i];          /* OK */
    int b[n];             /* OK */
    return n+m;
  }
  int b[funct2(n,m)][printf("%d\n",n)];  /* OK */
}
extern int n;
int a[n][n];              /* UNDEFINED: not block scope */
static int b[n][n];       /* UNDEFINED: not block scope */
extern int c[n][n];       /* UNDEFINED: not block scope */
int d[2+3][90];           /* OK */
void funct3(int i){
  extern int a[n][n];     /* UNDEFINED: a has linkage */
  static int b[n][n];     /* ERROR: b is static identifier */
  int c[i+3][abs(i)];     /* OK */
}
```

静的記憶期間を持つオブジェクトの初期化子が評価された場合、結果はコンパイル時にオブジェクトに格納されます。これとは異なり、自動記憶期間と形状無指定配列のサイズ式を含むオブジェクトの初期化子は評価された場合、結果はプログラムの実行時にオブジェクトに格納されます。次に例を示します。

```
#include <stdio.h>
int n = 4;               /* compile time n==4 */
int main(){
  int m = 5;             /* runtime m == 5 */
  int a[n++][n++];       /* order of evaluation is undefined */
  int b[n++], c[n++];    /* order of evaluation is undefined */
  int d[n++]; int e[n++];/* order of evaluation is defined */
  printf("%d %d %d", n--, b[n--], c[n--]); /* order of evaluation
                                              is undefined */
}
```

形状無指定配列のサイズは、実行時まで確定されません。形状無指定配列のサイズは、呼び出されるたびに異なります。したがって、形状無指定配列は初期化してはなりません。次に例を示します。

```
void funct1(int n){
  int a[3] = {1,2,3};               /* OK */
  int b[ ] = {1,2,3};               /* OK */
  int c[2][3] = {{1,2,3},{4,5,6}}; /* OK */
```

```
  int d[ ][3] = {{1,2,3},{4,5,6}}; /* OK */
  int e[n] = {1,2};                /* ERROR: initialization */
  int f[n][n] = {1,2,4,5};         /* ERROR: initialization */
}
```

形状無指定配列へのポインタは宣言してはなりません。次に例を示します。

```
void funct(int n){
  int (*p1)[3];      /* OK: pointer to fixed-length array */
  int (*p2)[n];      /* ERROR: pointer to deferred-shape array */
  int (*p3)[n][3];   /* ERROR: pointer to deferred-shape array */
  int (*p3)[3][n];   /* ERROR: pointer to deferred-shape array */
}
```

形状無指定配列が関数プロトタイプのスコープで宣言できるのは、配列インデックスも、整数型として宣言される前に関数プロトタイプのスコープで宣言される場合です。関数プロトタイプでは、配列インデックスの代わりにシンボル"*"を使用できます。次に例を示します。

```
void funct1(int n, a[n]);        // OK
void funct1(int n, a[*]);        // OK
void funct1(int n, a[n]){ }      // OK
void funct2(double n, int a[n]); // ERROR: n is not integral
void funct3(int a[n], n);        // ERROR: n in a[n] not declared
```

形状無指定配列と不完全な配列型とを混在させてはなりません。次に例を示します。

```
 int n;
 int a[][n] = {{1,2},{3,4}}; /* ERROR: initialization */
 void funct(int n, b[][n]);  /* ERROR: function prototype scope */
 extern c[][n];              /* ERROR: static storage duration */
```

2つの配列型に互換性を持たせるには、互換性のある要素型を両方の配列に持たせ、両者を同じ形状にする必要があります。次に例を示します。

```
void funct1(int (*p)[4]])
{int i = sizeof(p);}        /* i == 4 */
void funct2(int p[3][4])
{int i = sizeof(p);}        /* i == 4 */
void funct3(int p[ ][4])
{int i = sizeof(p);}        /* i == 4 */
void funct4(int n)
{
  int i = 3, j = 4;
  int (*p)[4];
```

```
    int a[i][j];
    int b[j][j];
    int c[i][i];
    p = a; funct1(a); funct2(a); funct3(a); /* compatible */
    p = b; funct1(b); funct2(b); funct3(b); /* compatible */
    p = c; funct1(c); funct2(c); funct3(c); /* incompatible */
  }
```

15.2.2 switch ステートメントに関連する形状無指定配列

形状無指定配列の宣言を含む switch ステートメントの制御式では、ブロックの範囲外からブロック内の case ラベルや default ラベルの後のステートメントへジャンプによってブロックに入らないようにする必要があります。そうしないと、ブロック内の形状無指定配列用のメモリが割り当てられません。次に例を示します。

```
int i;
int main(){
  int n = 10;
  switch (n){
    int a[n]; /* ERROR: bypass declaration of a[n] */
    case 10:
      a[0] = 1;
      break;
    case 20:
      a[1] = 2;
      break;
    case 30:
      {
        int b[n]; /* OK */
        b[1] = 90;
      }
      break;
  }
}
```

15.2.3 goto ステートメントに関連する形状無指定配列

goto ステートメントの識別子は、外側のブロック内またはその呼び出し元関数のどこかに配置されたラベルを呼び出します[1]。形状無指定配列の宣言を含む goto ステートメントでは、ブロックの範囲外からブロック内のラベル付きステートメントへジャンプによってブロックに入らないようにする必要があります。次に例を示します。

[1]訳注：しかし、その使用はお勧めできません

```
void funct(int n){
  int i;
label1:
  if(n>10)
    goto label2;      /* ERROR: bypass declaration of a[n] */
  {
    int a[n];
    a[i] = 8;
label2:
    a[i] = 9;
    goto label1;      /* OK */
label3:
    a[i] = 10;
    goto label2:      /* OK */
  }

void funct1(int m){
  void funct2(int r){
    if(r)
      goto label4;    /* OK */
    else
      goto label5;    /* ERROR: bypass declaration of b[m] */
  }
label4:
  {
    int b[m];
label5:
    a[0] = 9;
    goto label5;     /* OK */
  }
}
```

goto ステートメントによって、プログラムの実行が入れ子にされた関数から親関数に渡された[2]場合は、実行中の関数呼び出しを終了する必要があります。形状無指定配列用に割り当てられたメモリ部分を含め、動的に割り当てられたすべてのメモリ割り当てを解除し、以前の呼び出し環境を復元する必要があります。goto ステートメントを含む関数を呼び出した関数が、再度アクティブな関数になります。goto ステートメントに指定されたラベルが現在アクティブな関数でない場合は、現在の関数の非アクティブ化とその親関数のアクティブ化が繰り返し実行されます。最終的に、goto ステートメントのラベルを含む関数がアクティブになり、制御フローは該当するラベルを含むステートメントに移行します。次に例を示します。

```
void funct1(int n){
```

[2]訳注：そのこと自体もお勧めしませんが

```
    local void funct2(int n);
    local void funct3(int n);
    int a[n];
  label:
    funct2(n);
    void funct2(int n){
      int b[n];
      funct3(n);
    }
    void funct3(int n){
      int c[n];
      goto label; /* b[n] and c[n] will be deallocated*/
    }
  }
```

この例では、形状無指定配列`b[n]`と`c[n]`に割り当てられたメモリは、制御フローが関数`funct2()`を介して関数`funct3()`から関数`funct1()`に移行した時点で割り当て解除されます。上記の例で、*label*ステートメントと*goto label*ステートメントを関数`setjmp(buf)`と関数`longjmp(buf)`[3]とでそれぞれ置き換えると、形状無指定配列のメモリ割り当ては解除されません。入れ子にされた関数[4]では、関数`setjmp(buf)`および`longjmp(buf)`は非推奨機能になる可能性が十分あります。

15.2.4 構造体と共用体のメンバとしての形状無指定配列

通常の識別子だけでなく、構造体や共用体のメンバも、形状無指定配列として宣言できます。ただし、形状無指定配列のメンバを含む構造体と共用体は、自動記憶期間を持つように宣言する必要があります。ファイルまたはプログラムのスコープ内で形状無指定配列のメンバを含む構造体や共用体を宣言した場合、その宣言の動作は未定義です。ブロックスコープ内で`static`記憶指定子を使用して宣言されている構造体は、形状無指定配列のメンバとして宣言することはできません。

`sizeof`と同様、`offsetof`も組み込みの演算子です。構造体に形状無指定配列のメンバが含まれていない場合、演算子`offsetof`(*type, member-designator)*が、型`size_t`を持つ整数定数値であると評価します。この整数定数値は、(*type*によって指定される) 構造体の先頭から (*member-designator*で指定される) 構造体メンバへのオフセットをバイトで表した値です。構造体が形状無指定配列のメンバを含む場合、結果は定数式とはならず、プログラム実行時に計算されます。形状無指定配列が可変長であるため、たとえば

`static` *type* t;

のようなコードの場合、構造体が形状無指定配列を含んでいる場合、式`&(t.`*member-designator)*はアドレス定数と評価されません。

構造体と共用体は、関数プロトタイプのスコープでは定義されません。形状無指定配列のメンバを含む構造体や共用体は、入れ子にされた関数の関数プロトタイプのスコープで宣言できます。次に例を示します。

[3]訳注：この 2 つの関数もお勧めできません。
[4]訳注：これもお勧めできません。

```
int n;
struct tag{
  int m;
  int a[n];            /* UNDEFINED: not block scope for tag1 */
  int b[m];            /* UNDEFINED: not block scope for tag1 */
};
void funct1(int m){
  int l;
  static struct tag{
    int m;
    int a[n];          /* ERROR: static block scope for tag1 */
    int b[m];          /* ERROR: static block scope for tag1 */
  };
  struct tag1{         /* structure shared by
                          funct1(), funct2(), and funct3() */
    int r=2*m;         /* initialization of member r */
    int a[n][m][l];                 /* OK */
    int q=2+l, q2;     /* initialization of member q */
    int b[r][q];                    /* OK */
  };
  void funct2(){
     struct tag1 s1;                /* OK */
     int i;
     i = offsetof(struct tag1, r); /* OK: runtime offsetof() */
     i = offsetof(struct tag1, a); /* OK: runtime offsetof() */
     i = offsetof(struct tag1, b); /* OK: runtime offsetof() */
  }
  /* structure with deferred-shape array as function arg */
  void funct3(struct tag1 s){
    int i, j;
    struct tag1 s1;                 /* OK */
    for(i=0; i<s.r; i++)
      for(j=0; j<s.q; j++)
        s1.b[i][j] = s.b[i][j];
  }
  struct tag2{
    int a[2][3];
    int b[4][5];
  };
  l = offsetof(struct tag1, r);     /* OK: runtime offsetof() */
  l = offsetof(struct tag1, a);     /* OK: runtime offsetof() */
  l = offsetof(struct tag1, b);     /* OK: runtime offsetof() */
```

```
    l = offsetof(struct tag2, a);    /* OK: compile time offsetof() */
    l = offsetof(struct tag2, b);    /* OK: compile time offsetof() */
  }
```

15.2.5 Sizeof

組み込みの演算子 `sizeof` を配列型のオペランドに適用すると、演算結果は、配列の要素を格納するために割り当てられたバイトの総数になります。形状無指定配列では、結果は定数式にはならず、プログラム実行時に計算されます。次に例を示します。

```
int funct(int n, m){
  int i;
  int a[3][4];
  int b[n][m];
  int c[sizeof(a)];          /* c is fixed-length array */
  int d[sizeof(b)];          /* d is deferred-shape array */
  i = sizeof(a);             /* compile time sizeof(a) is 48 */
  j = sizeof(b);             /* runtime sizeof(b) is nxmx4 */
  return j;
}
```

`sizeof` 演算子を構造体型または共用体型のオペランドに適用すると、結果は、そのようなオブジェクトに含まれるバイトの総数 (内部埋め込みと末尾埋め込みを含む) になります。構造体または共用体のメンバが形状無指定配列である場合、結果は定数式にはならず、プログラム実行時に計算されます。次に例を示します。

```
int n;
int funct1(int m){
  int l;
  struct tag1{
    int a[2][3];
  };
  struct tag2{
    int r;
    int a[4][5];
    int b[n][m][l][r];
  };
  int i;
  struct tag1 s1;
  struct tag2 s2;
  void funct2(struct tag1 s1, struct tag2 s2){
    int i;
    i = sizeof(s1);          /* compile time sizeof() */
```

```
    i = sizeof(s1.a);       /* compile time sizeof() */
    i = sizeof(s2.a);       /* compile time sizeof() */
    i = sizeof(s2);         /* runtime sizeof() */
    i = sizeof(s2.b);       /* runtime sizeof() */
  }
  i = sizeof(struct tag1); /* compile time sizeof() */
  i = sizeof(s1);          /* compile time sizeof() */
  i = sizeof(s1.a);        /* compile time sizeof() */
  i = sizeof(s2.a);        /* compile time sizeof() */
  i = sizeof(struct tag2); /* runtime sizeof() */
  i = sizeof(s2);          /* runtime sizeof() */
  i = sizeof(s2.b);        /* runtime sizeof() */
}
```

15.2.6 Typedef

形状無指定配列を含む集合体型を指定する Typedef 宣言は、ブロックスコープ内で行う必要があります。ファイルまたはプログラムのスコープ内で形状無指定配列を typedef 宣言した場合、動作は未定義です。配列の形状無指定は、型定義が宣言されるときではなく、実際の宣言子内で型指定子として使用されるときは必ず評価される必要があります。次に例を示します。

```
int n = 5;
typedef int A[n];                    /* UNDEFINED: not block scope */
typedef struct tag{int aa[n]} TAG1;/* UNDEFINED: not block scope */
int main(){
  int n;
  typedef int B[n];                  /* OK */
  B bb;                              /* OK: int bb[n] */
  B *cc;                             /* ERROR: int (*cc)[n] */
}
void funct(int m){
  typedef int A[m];                  /* m is not stored in A */
  typedef struct tag {
    int b[m];                        /* store m in b[m] */
    struct tag *prev;
    struct tag *next;
} TAG1;
A d;                                 /* store m in d */
m++;                                 /* increment m */
{
   A a;              /* a[] has one more element than d[] */
   TAG1 s;           /* s.b[] has one more element than d[] */
```

```
    int c[m];          /* c[] has one more element than d[] */
  }
  }
  funct(6);            /* ==> d[6], a[7], s.b[7], c[7] */
```

15.2.7 その他のデータ型とポインタ演算

固定長配列と同じ方法で、データ型の異なる形状無指定配列を宣言できます。次に例を示します。

```
void funct(int n){
  char c[n], *cp[n];
  int *ip[n][n];
  float f[n], **fp[n][n];
  double d[n], *dp[n][n];
  complex z[n], *zp[n][n];
}
```

固定長配列に関連するポインタ演算は、形状無指定配列の場合でも有効です。次に例を示します。

```
void funct(int n, m){
  int i=0, j=0;
  int a[n][m];
  a[i][j] = 90;
  *(a[i]+j) = 90;          /* a[i][j] = 90 */
  *(*(a+i)+j) = 90;        /* a[i][j] = 90 */
  *(&a[0][0]+i*m+j) = 90;  /* a[i][j] = 90 */
  *((int *)a[i]+j) = 90;   /* a[i][j] = 90 */
  *((int *)(a+i)+j) = 90;  /* a[i][j] = 90 */
  *((int *)a+i*m+j) = 90;  /* a[i][j] = 90 */
  i = a[n-1] - a[n-2];     /* i == m */
}
```

15.3 形状引継ぎ配列

15.3.1 制約と意味

形状引継ぎ配列の宣言は、関数プロトタイプのスコープ内で、または typedef 宣言で行う必要があります。形状引継ぎ配列は、その配列に渡された実引数の形状を引き継ぐ仮引数です。すなわち、実引数と仮引数の配列は、各次元で同じランクと同じエクステントを持ちます。形状無指定配列の形状は、実行時まで確定できません。形状引継ぎ配列のランクは、形状引継ぎの指定に含まれるコロンの数と等しくなります。次に例を示します。

```
void funct(int [:], [:][:])       // OK
void funct(int dummy1[:], dummy2[:][:]) // OK
void funct(int a[:], b[:][:])     // OK
int A[:];                         // ERROR: not function prototype scope
static int B[:][:];               // ERROR: not function prototype scope
extern C[:][:];                   // ERROR: not function prototype scope
void funct(int a[:], b[:][:]){    // OK
  int c[:][:];                    // ERROR: not function prototype scope
  extern int A[:];                // ERROR: not function prototype scope
  void funct2(int a[:], b[:][:]){// OK
     int c[:][:];                 // ERROR: not function prototype scope
  }
  funct2(a, b);                   // OK
}
```

プログラム 15.1に、形状引継ぎ配列のプログラム例を示します。

```
void bxc(double aa[:][:], double bb[:][:], double cc[:][:], int n, int m, int r);
int main() {
  int i, n = 2, m = 4, r =6;
  double a[2][6], b[2][4], c[4][6]; // double a[n][r], b[n][m], c[m][r]
  /* ... */
  bxc(a,b,c,n,m,r);
}

void bxc(double aa[:][:], double bb[:][:], double cc[:][:], int n, int m, int r) {
  /* array multiplication a = b[n][m]*c[m][r] */
  int i, j, k;

  for(i=0; i<=n-1; i++)
    for(j=0; j<=r-1; j++) {
      aa[i][j] = 0;
      for(k=0; k<=m-1; k++)
        aa[i][j] += bb[i][k]*cc[k][j];
    }
}
```

プログラム 15.1: 形状引継ぎ配列を使用して 2 次元配列を関数に渡す処理

このプログラムでは、関数 `bxc()` は 2 つの 2 次元配列 `b` と `c` とを乗算します。引数 `a` で、乗算結果は呼び出し元関数へ返されます。呼び出し元関数内の配列の次元は、wn、`m`、`r` の 3 つのパラメータで関数 `bxc()` に渡されます。

形状引継ぎ配列は、typedef 宣言でも使用できます。次に例を示します。

```
typedef int A[:];
A a;                 /* ERROR: not function prototype scope */
void funct(A a);     /* OK */
```

形状引継ぎ配列型の仮引数に対する実引数として関数パラメータに使用できるのは、固定長配列型、形状無指定配列型、または形状引継ぎ配列型の変数のみです。形状の情報が完全でないポインタまたは配列へのポインタは、形状引継ぎ配列の仮引数に対する実引数として使用することはできません。次に例を示します。

```
funct1(int a[:][:]){
  int n=a[1][1], m = a[1][2];
  int b[3][4];
  int c[n][m];
  int *p1, (*p2)[4], (*p3)[:];
  void funct3(int a[:][:]);
  void funct2(int a[:][:])
  { }
  funct2(a);  funct3(a); // OK a is assumed-shape array
  funct2(b);  funct3(b); // OK b is fixed-length array
  funct2(c);  funct3(c); // OK c is deferred-shape array
  funct2(p1); funct3(p1);// ERROR: p1 is pointer
  funct2(p2); funct3(p2);// ERROR: p2 is pointer to fixed-length array
  funct2(p3); funct3(p3);// ERROR: p3 is pointer to assumed-shape array
}
void funct3(int a[:][:])
{ }
```

配列へのポインタから完全な配列を抽出することは可能ですが、それらを形状引継ぎ配列の実引数として使用してはなりません。次に例を示します。

```
void funct1(int a[3]);
void funct2(int a[5][7]);
void funct11(int a[:]);
void funct22(int a[:][:]);
void funct3(int p2[][5][7]){
  int a[5][3];
  int (*p1)[3];
  p1 = a;
  funct1(p1[0]);   // OK: passed a[0][0], ..., a[0][2]
  funct1(p1[1]);   // OK: passed a[1][0], ..., a[1][2]
  funct1(*(p1+1)); // OK: passed a[1][0], ..., a[1][2]
  funct1(a[4]);    // OK: passed a[4][0], ..., a[4][2]
  funct1(p1+1);    // OK: p1+1 is a pointer to array of 3 ints
  funct2(p2[1]);   // OK: passed p2[1][0][0], ..., p2[1][4][6]

  funct11(p1[0]);  // ERROR: passing array a[0][0], ..., a[0][2]
  funct11(p1[1]);  // ERROR: passing array a[1][0], ..., a[1][2]
```

```
  funct11(*(p1+1));// ERROR: passing array a[1][0], ..., a[1][2]
  funct11(a[4]);   // ERROR: passing array a[4][0], ..., a[4][2]
  funct11(p1+1);   // ERROR: p1+1 is a pointer to array of 3 floats
  funct22(p2[1]);  // ERROR: passing array p2[1][0][0],...,p2[1][4][6]
}
```

ここで、p1[0]、p1[1]、
t *(p1+1)、a[4] は 12 バイトの配列です。p2[1] は 140 バイトの配列で、p+1 は 4 バイトの配列へのポインタです。

形状引継ぎ配列を、固定長配列型や不完全な配列型と混在させてはなりません。次に例を示します。

```
void funct(int a[:][3]);// ERROR: mix assumed-shape with fixed-length
void funct(int a[3][:]);// ERROR: mix assumed-shape with fixed-length
void funct(int a[n][:]);// ERROR: mix assumed-shape with deferred-shape
void funct(int a[:][n]);// ERROR: mix assumed-shape with deferred-shape
void funct(int a[ ][:]);// ERROR: mix assumed-shape with incomplete
```

ポリモーフィックな演算や関数のオペランドが形状引継ぎ配列の要素である場合、結果および演算のデータ型は仮引数のデータ型によって異なります。ただし、仮引数と実引数とでデータ型が異なっても互換性がある場合は、演算が行われる前に、オペランドは仮引数のデータ型を含むオペランドにキャストされます。要素が lvalue として使用される場合、データ型が異なるときは、rvalue は実引数のデータ型にキャストされます。つまり、実際の配列の要素は、フェッチされるときには、プログラム実行時に形状引継ぎ配列のデータ型に強制されますが、格納されるときには、実引数のデータ型に強制されます。次に例を示します。

```
float A[3] = {1, 2};
complex Z[3] = {complex(1,0), complex(2,0)};
void funct(float a[:], complex z[:]){
  a[2] = a[0] + a[1];      /* addition of floats */
  z[2] = z[0] + z[1];      /* addition of complexs */
}
funct(Z, A);               /* A[2]==3.0, Z[2]=3.0+i0.0 */
```

仮引数が形状引継ぎ型であれば、実引数も形状引継ぎ配列にできます。次に例を示します。

```
void funct2(complex aa[:], b[:][:], (*c)[6], d[][6], e[4][6]){
  aa[1] = b[1][2];
}
void funct1(complex a[:], b[:][:]){
  if(real(a[1]) == 0)
  funct2(a,b,b,b,b);       /* a and b are assumed-shape arrays */
}
int main(){
  complex  A[2], B[4][6];
```

```
    funct1(A,B);            /* A and B are fixed-length arrays */
  }
```

関数 funct2() の funct2(a,b,b,b,b) の関数呼び出しで呼び出されると、主ルーチンの配列 A に割り当てられたメモリは関数 funct1() の形状引継ぎ配列 a で使用され、その後、関数 funct2() の形状引継ぎ配列 aa に渡されます。形状引継ぎ配列は、関数内の固定長配列へのポインタの実引数としても使用できます。上記の例では、主ルーチンの配列 B に割り当てたメモリは、関数 funct1() では b として、関数 funct2() では b、c、d、e として使われています。異なる識別子の a と aa が、配列 A の宣言で割り当てられた同じ配列オブジェクトに使用されています。ところが、配列オブジェクト B の関数 funct1() と関数 funct2() では、どちらにも同じ識別子 b が使われています。これは、識別子の名前が、関数の引数の関連付けとは無関係であることを示しています。

15.3.2 Sizeof

sizeof() 演算のオペランドが形状引継ぎ配列型である場合、結果は、プログラム実行時に計算された配列の要素を格納するために使用されたバイトの総数になります。また、データ型が異なる配列を形状引継ぎ配列に渡すことが可能なため、形状引継ぎ配列の要素のサイズもプログラム実行時に計算されます。次に例を示します。

```
  int funct(complex z[:]){
    int i, numOfElement;
    numOfElement = sizeof(z)/sizeof(z[0]);
    return numOfElement; /* sizeof(z)=80, sizeof(z[0])=4 */
  }
  int main(){
    int num;
    float a[20];
    num = funct(a);       /* num == 20 */
  }
```

15.3.3 その他のデータ型とポインタ演算

その他のデータ型の形状引継ぎ配列は、int 型の形状引継ぎ配列と同じ方法で扱われます。たとえば、次のステートメントでは、変数 a、b、c を、それぞれランク 1、2、3 の複素数をデータに持つ形状引継ぎ配列として宣言しています。

```
int funct(complex a[:], b[:][:], c[:][:][:]);
```

同じ方法で、データ型の異なる形状引継ぎ配列を扱うことができます。たとえば、次のコード例

```
char *cc[10]; float **ff[2][4]; double ***dd[3][5][7];
int funct(char *c[:]; float **f[:][:], double ***d[:][:][:]);
funct(cc, ff, dd);
```

では、次の関数プロトタイプ

```
int funct(char *c[:], float **f[:][:], double ***d[:][:][:]);
```

を使用して、変数 c、f、d を、それぞれ、char 型へのポインタを示すランク 1 の形状引継ぎ配列、float 型への二重ポインタを示すランク 2 の形状引継ぎ配列、および double 型への三重ポインタを示すランク 3 の形状引継ぎ配列として定義しています。配列 cc、ff、dd は、関数 funct() 内の形状引継ぎ配列 c、f、d にそれぞれ渡されます。関数内では、形状引継ぎ配列は固定長配列と同じように扱われます。次に例を示します。

```
  void funct(complex z1[:], z2[:][:]){
    complex z, *zp, **zp2;
    zptr = z1;              /* the address of the array */
    zptr = &z1[2];          /* the address of the third element */
    /* z2[2][1] -= 1; z1[1] = z2[2][1] + z1[2]; z1[2] += 1; */
    z1[1] = --z2[2][1]+z1[2]++;
    z = *z1;                /* z = z1[0] */
    z = *(z1+5);            /* z = z1[5] */
    z = **z2;               /* z = z2[0][0] */
    zp = z2[2];             /* zp = &z2[2][0] */
    zp2 = (complex **)z2;
    /* z2[1][1] = z2[1][3] + z2[2][3] - z2[2][4]; */
    zp2[1][1] = z2[1][3]+ *(*(z2+2)+3) - *(4+*(z2+2));
    /* z2[2][3] = z2[1][3] + z2[2][3] */
    *(*(z2+2)+3) = z2[1][3]+  *(3+*(z2+2));
}
```

15.4 形状引継ぎ配列へのポインタ

15.4.1 宣言

ポインタ型は、関数型、オブジェクト型、または**参照型**と呼ばれる不完全な型から派生されます。ポインタ型とは、参照型のエンティティを参照する値を持つオブジェクトのことです。参照型 *T* から派生するポインタ型は、“*T* へのポインタ”と呼ばれることがあります。参照された型からポインタ型を作成することをポインタ型の作成と呼びます。

セクション 15.1.2で説明されている宣言 “T D1”で、D1 が次の形式

$\star$ *type-qualifier-list*$_{opt}$ D

で指定される場合、*ident* が表す配列の要素に指定する型は T で指定されるデータへのポインタ型となります。リストの各型修飾子では、*ident* はこのように型修飾されたポインタとなります。

固定長配列へのポインタは、次の形式で宣言します。

T ($\star$D)[*assignment-expression*]

ここで、T に型を指定する宣言指定子を指定します。代入式は整数定数式です。次に例を示します。

```
int (*p1)[3];      /* p1 is pointer to array of 3 ints */
int *(*p2)[3];     /* p2 is pointer to array of 3 pointer to int */
int (*p3)[3][4];   /* p3 is pointer to 3x4 array of ints */
int *(*p4)[3][4]; /* p4 is pointer to 3x4 array of pointer to int */
```

形状引継ぎ配列へのポインタは、次の形式で宣言します。
T (*D)[:]
ここで、T に型を指定する宣言指定子を指定します。次に例を示します。

```
int (*p1)[:];      /* OK */
int (*p2)[:][:];   /* OK */
int *(*p3)[:][:]; /* OK */
int n = 8;
int (*p4)[3][:];   /* ERROR: mix fixed-length with assumed-shape */
int (*p5)[:][3];   /* ERROR: mix fixed-length with assumed-shape */
int (*p6)[n][:];   /* ERROR: mix deferred-shape with assumed-shape */
int (*p7)[:][n];   /* ERROR: mix deferred-shape with assumed-shape */
int (*p8)[ ][:];   /* ERROR: mix deferred-shape with incomplete type */
```

ここで、p1 は int 型のランク 1 を示す形状引継ぎ配列のポインタ、p2 は int 型のランク 2 を示す形状引継ぎ配列のポインタ、p3 は int 型のランク 2 を示す形状引継ぎ配列のポインタです。

形状引継ぎ配列へのポインタが参照する配列の形状は、プログラム実行時に確定されます。形状引継ぎ配列へのポインタがファットポインタと呼ばれることがあるのは、プログラム実行時に、スカラ型のオブジェクトへのポインタや固定長配列へのポインタよりも多くの情報を格納できるためです。

15.4.2 制約と意味

形状引継ぎ配列へのポインタ型を除き、void へのポインタは、ポインタ間で、不完全な型またはオブジェクト型に変換することができます。不完全な型またはオブジェクト型へのポインタは、形状引継ぎ配列へのポインタ型を除き、void へのポインタに変換し、それからもう一度変換できます。変換結果が元のポインタと同じかどうかを比較する必要があります。

任意の修飾子 q に対して、q で修飾されていない型へのポインタは、同じ型の q で修飾されたバージョンに変換できます。元のポインタと変換後のポインタに格納されている値が同じかどうかを比較する必要があります。

値 0 を含む整数定数式 (型 void *にキャストされた式など) は、null ポインタと呼ばれます。*null* **ポインタ**定数が、ポインタへ代入されるか、ポインタと同じであるかどうかが比較される場合、定数はその型のポインタに変換されます。*null* **ポインタ**と呼ばれるこのようなポインタで、オブジェクトや関数へのポインタとの不一致が比較されるようになっています。

おそらくさまざまなシーケンスのキャストによってポインタ型に変換された 2 個の null ポインタについて、それらの値が等しいかどうかを比較する必要があります。

null ポインタを形状引継ぎ配列へのポインタに変換する場合、null ポインタは、形状引継ぎ配列のベースポインタでインストールされ、形状引継ぎ配列の境界は未定義です。

固定長配列、形状無指定配列、形状引継ぎ配列などの配列は、形状引継ぎ配列へのポインタに変換できます。また、固定長配列へのポインタや形状引継ぎ配列へのポインタも、形状引継ぎ配列へのポインタに変換できます。配列へのベースポインタとすべての境界が、形状引継ぎ配列へのポインタに格納されます。これらのポインタ型以外の、配列形状の情報を持たないポインタ型は、形状引継ぎ配列へのポインタに変換することはできません。次に例を示します。

```
void funct(int a[:][:], p1[2][4], (*p2)[4], p3[][4], n, m){
  int *p;
  int b[3][4];
  int c[n][m];
  int (*p4)[4];
  int (*p5)[:];
  int (*p6)[:];
  p6 = NULL;
  p6 = a;        /* OK: a is an assumed-shape array */
  p6 = b;        /* OK: b is a fixed-length array */
  p6 = c;        /* OK: c is a deferred-shape array */
  p6 = p1;       /* OK: p1 is a pointer to array of fixed-length */
  p6 = p2;       /* OK: p2 is a pointer to array of fixed-length */
  p6 = p3;       /* OK: p3 is a pointer to array of fixed-length */
  p6 = p4;       /* OK: p4 is a pointer to array of fixed-length */
  p6 = p5;       /* OK: p5 is a pointer to array of assumed-shape */
  p4 = p;        /* WARNING: array bounds do not match */
  p6 = p;        /* ERROR: p is not array type */
}
```

2つのポインタ型に互換性を持たせるには、両者が同じように修飾され、どちらも互換性のある型へのポインタであることが必要です。固定長配列への2つのポインタに互換性を持たせるには、ポインタが参照する配列の形状を両者とも同じにする必要があります。形状引継ぎ配列への2つのポインタに互換性を持たせるには、ポインタが指す配列のランクを両者とも同じにする必要があり、また、プログラム実行時に、形状が同じ値であることが評価される必要があります。次に例を示します。

```
void funct(int a[:], b[:][:][:], (*p1)[4][5], p2[3], n, m){
  int c[3][4][5];
  int d[n][m][m];
  int (*p3)[4][5];
  int (*p4)[:];
  p4 = a;        /* ERROR: incompatible, wrong rank  */
  p4 = b;        /* ERROR: incompatible, wrong rank  */
  p4 = p1;       /* ERROR: incompatible, wrong rank  */
  p4 = p2;       /* ERROR: incompatible, wrong rank  */
  p4 = c;        /* ERROR: incompatible, wrong rank  */
  p4 = d;        /* ERROR: incompatible, wrong rank  */
```

```
  p3 = p4;       /* WARNING: incompatible, wrong rank */
}
```

形状引継ぎ配列へのポインタが、他のオブジェクトへのポインタまたはスカラ値に変換される場合は、形状引継ぎ配列へのベースポインタのみが使用されます。

```
char c, *cp;
int i, *ip;
float f, *fp;
int (*ap)[4];
int (*p)[:];
c  = (char) p;       /* OK */
cp = (char *) p;     /* OK */
i  = (int) p;        /* OK */
ip = (int*) p;       /* OK */
ip = p;              /* OK */
f  = (float) p;      /* OK */
fp = (float*) p;     /* OK */
ap = p;              /* OK */
```

15.4.3 関数プロトタイプのスコープ

形状引継ぎ配列へのポインタを関数の引数パラメータとして使用し、サイズの異なる配列を関数に渡すことができます。次に例を示します。

```
void funct(int (*)[:]);
void funct(int (*dummy)[:]);
void funct(int (*p)[:]);
int a[3][4], b[4][3];
int (*p1)[:];
funct(a,3,4);       /* passing fixed-length array a[3][4] */
funct(b,3,4);       /* passing fixed-length array b[4][3] */
p1 = a;
funct(p1,3,4);      /* passing fixed-length array a[3][4] */
funct(NULL,0,0);    /* passing NULL */
void funct(int (*p)[:], n, m){
  int i, j;
  int a[n][m];
  if(p == NULL)
    return;
  for(i=0; i<n; i++)
    for(i=0; i<m; i++)
      a[i][j] = p[i][j];
}
```

また、形状無指定配列や形状引継ぎ配列も、形状引継ぎ配列へのポインタに渡すことができます。次に例を示します。

```
void funct1(int a[:][:], n, m){
  int b[n][m];
  void funct2(int (*p)[:])
  {
    int i, j;
    int c[n][m];
    if(p == NULL)
      return;
    for(i=0; i<n; i++)
      for(i=0; i<m; i++)
        c[i][j] = p[i][j];
  }
  funct2(a);  /* a is an assumed-shape array */
  funct2(b);  /* b is a deferred-shape array */
}
```

15.4.4 Typedef

typedef 宣言では、形状引継ぎ配列および形状引継ぎ配列へのポインタは、固定長配列および固定長配列へのポインタと同様に扱われます。次に例を示します。

```
typedef int A[5];
typedef int B[:];
A a;              // OK: int a[5]
A *ap;            // OK: int (*ap)[5]
B b;              // ERROR: not function prototype scope for 'int b[:]'
B *bp;            // OK: int (*bp)[:]
/* void funct(int a[5], (*ap)[5], int b[:], (*bp)[:]); */
void funct(A a, *ap, B b, *bp); // OK
```

ここで、a は形状引継ぎ配列、ap は形状引継ぎ配列へのポインタです。

15.4.5 メモリ割り当て関数による配列の動的割り当て

次の例に示すように、配列を動的に割り当てることができます。

```
funct(int n, int m){
  double a[n][m];
  double (*p1)[:] = a;   // OK p[i][j] = a[i][j]
  double (*p2)[:] = (double [n][m])malloc(sizeof(double)*n*m);// OK
```

```
    double (*p3)[:] = (double [ ][m])malloc(sizeof(double)*n*m);// OK
    double (*p4)[:] = (double [][m])malloc(sizeof(double)*n*m); // OK
    /* ERROR: pointer to deferred-shape array is not allowed */
    double (*p5)[:] = (double(*)[m])malloc(sizeof(double)*n*m);// ERROR
  }
```

ここで、a は形状無指定配列です。p1、p2、p3 は double データ型の形状引継ぎ配列へのポインタです。p1、p2、p3、p4 が参照するメモリはすべて動的に割り当てられます。ただし、p2、p3、および p4 のメモリは、メモリ割り当て関数 malloc() によって明示的に取得されます。

15.4.6 固定長配列へのポインタと形状引継ぎ配列へのポインタとの類似点

形状引継ぎ配列へのポインタと固定長配列へのポインタとは、動作が非常によく似ています。たとえば、オブジェクトの各要素を参照できるようになるためには、ポインタとしてオブジェクトを指している必要があります。一見簡単そうでない他のポイントについても、このセクションで詳しく説明します。

静的記憶期間と自動記憶期間

形状無指定配列とは異なり、形状引継ぎ配列へのポインタを宣言できるスコープに制約はありません。静的記憶期間か自動記憶期間のどちらかで宣言が可能です。次に例を示します。

```
int (*p1)[:];
extern int (*p2)[:];
static int (*p3)[:];
int main(){
  int (*p4)[:];
  static (*p5)[:];
  extern int (*p1)[:];
}
```

初期化

形状引継ぎ配列へのポインタは、コンパイル時とプログラム実行時のどちらでも初期化できます。次に例を示します。

```
int a[3][4];
int (*p1)[:] = NULL;         /* runtime initialization */
extern int (*p2)[:];
static int (*p3)[:] = NULL; /* runtime initialization */
int main(){
  int b[3][4];
  int (*p4)[:] = NULL;       /* compile time initialization */
```

```
    int (*p5)[:] = b;          /* compile time initialization */
    int (*p6)[:] = p1;         /* compile time initialization */
    static (*p7)[:] = a;       /* runtime initialization */
    static (*p8)[:] = p1;      /* runtime initialization */
    static (*p8)[:] = b;       /* ERROR: b is variable of auto class */
  }
```

構造体と共用体のメンバ

通常の識別子だけでなく、クラス、構造体、共用体のメンバも形状引継ぎ配列へのポインタとして宣言できます。次に例を示します。

```
  struct tag1{
    int (*p1)[3];     /* pointer to fixed-length array */
    int (*p2)[:];     /* pointer to assumed-shape array */
  };
  int main(){
    struct tag2{
      int (*p1)[3];  /* pointer to fixed-length array */
      int (*p2)[:];  /* pointer to assumed-shape array */
    } s;
  }
```

ここで、構造体 `tag1` には静的記憶期間が指定され、構造体 `tag2` には自動記憶期間が指定されています。以下の Ch シェルで実行される対話型コマンドでは、メンバ s.a は、配列 a1 と同じメモリを共有した後に配列 a2 のメモリを共有します。

```
  > struct tag{ int (*a)[:];} s
  > int a1[2][3] = {1,2, 3, 4, 5, 6}, a2[3][4]
  > s.a = a1;  // s.a and a1 share the memory
  > a1[1][1]
  5
  > s.a[1][1]
  5
  > s.a = a2;  // s.a and a2 share the memory
  s.a[1][1] = 10
  > a2[1][1]
  10
  > a1[1][1]
  5
```

Sizeof

形状引継ぎ配列へのポインタのサイズは、固定長配列へのポインタのサイズと同じです。形状引継ぎ配列へのポインタのサイズは、配列のデータ型へのポインタのサイズと同じです。ポインタのサイズはコンパイル時に評価されます。次に例を示します。

```
int (*a)[5];
int (*p1)[:];
int main(){
  int (*b)[5];
  int (*p2)[:];
  void funct(int (*p3)[:], (*p4)[:][:])
  {
    int i;
    i = sizeof(a);  /* i == 4 */
    i = sizeof(b);  /* i == 4 */
    i = sizeof(p1); /* i == 4 */
    i = sizeof(p2); /* i == 4 */
    i = sizeof(p3); /* i == 4 */
    i = sizeof(p4); /* i == 4 */
  }
}
```

その他のデータ型とポインタ演算

int 型の形状引継ぎ配列へのポインタと同じ方法で、データ型の異なる形状引継ぎ配列へのポインタを宣言できます。次に例を示します。

```
void funct(int n){
  char    (*cp1)[:], *(*cp2)[:], **(*cp3)[:];
  int     (*ip1)[:], *(*ip2)[:], **(*ip3)[:];
  float   (*fp1)[:], *(*fp2)[:], **(*fp3)[:];
  double  (*dp1)[:], *(*dp2)[:], **(*dp3)[:];
  complex (*zp1)[:], *(*zp2)[:], **(*zp3)[:];
}
```

固定長配列へのポインタに関連するポインタ演算は、形状引継ぎ配列へのポインタにも有効です。次に例を示します。

```
int main(){
  int i=2, j=3;
  int n=4, m=5;
  int a[4][5];
  int (*p)[:]
```

```
    p = a;
    p[i][j] = 90;           /* a[i][j] = 90 */
    *(p[i]+j) = 90;         /* a[i][j] = 90 */
    *(*(p+i)+j) = 90;       /* a[i][j] = 90 */
    *(&p[0][0]+i*m+j) = 90; /* a[i][j] = 90 */
    *((int *)p[i]+j) = 90;  /* a[i][j] = 90 */
    *((int *)(p+i)+j) = 90; /* a[i][j] = 90 */
    *((int *)p+i*m+j) = 90; /* a[i][j] = 90 */
    i = p[n-1] - p[n-2];    /* i == m */
}
```

15.5 上限値と下限値を明示した配列

ここまでのセクションの説明からわかるとおり、C で可変長の添字範囲の配列を扱うことは難しく、特に高次元配列の場合は困難です。データ型や次元が異なる配列では、メモリ割り当てとメモリ割り当て解除に `mallocMatrix()` と `freeMatrix()` に相当するような関数を個別に使用する必要があります。C を数値計算で重要な役割を担う言語に発展させるには、可変長の添字範囲を持つ可変長配列を扱えるようにする以外に方法はありません。これは明らかです。このセクションでは、 Ch プログラミング言語に現在実装されている、上限値と下限値を明示して可変長配列を扱うための簡単なメカニズムを説明します。なお、ここで説明する新しい機能は、C 規格および既存の C コードを無効にするものではないことを強調しておきます。

15.5.1 固定添字範囲の配列

下限値を指定して配列を宣言する場合は、次のいずれかの形式で行います。

```
T D[lower:upper]     (15.10)
T D[expr]            (15.11)
T D[lower:]          (15.12)
```

ここで、`T` は `int` などの型を指定する宣言指定子、`D` は識別子 *ident* を含む宣言子、`lower` は配列の下限値、`upper` は配列の上限値、`expr` は配列の要素の数をそれぞれ表します。式 `lower`、`upper`、`expr` は、整数型の式にする必要があります。次に例を示します。

```
int a[1:3], b[0:2][1:5], *c[1:3][1:4][0:5];
```

ここで、配列 a の下限値と上限値は、それぞれ 1 と 3 です。要素 `a[0]` と `a[4]` は配列境界の外にあります。

宣言 (15.11) で下限値を指定しない場合、配列の下限値には既定値のゼロが使用されます。上限値は、”[ ]”で囲んだ値から 1 を引いたもの (`expr`-1) になります。次に例を示します。

```
int b[0:2][5];          /* equivalent to int b[0:2][0:4] */
int a[3];               /* equivalent to int a[0:2] */
int *c[1:3][4][0:5];    /* equivalent to int *c[1:3][0:3][0:5]; */
```

ここで、配列 a の下限値と上限値は、それぞれ 0 と 2 です。要素 `a[-1]` と `a[3]` は配列境界の外にあります。

下限値と上限値の両方を負の整数値にすることもできます。次に例を示します。

```
int a[-5:5], b[-5:0], c[-10:-5];
```

固定添字範囲の配列では、下限値と上限値の式は両方とも整数定数です。上限値は、下限値より大きい値であると評価される必要があります。次に例を示します。

```
#define N 0
float ff = 5;
int a[5.0];            /* ERROR: expression double type*/
int b[ff];             /* ERROR: expression float type */
int c[0];              /* ERROR: lower and upper bounds are equal */
int d[N];              /* ERROR: lower and upper bounds are equal */
int e[5:5];            /* ERROR: lower and upper bounds are equal */
int f[5:0];            /* ERROR: upper is not greater than lower */
int g[5:-5];           /* ERROR: upper is not greater than lower */
```

宣言 (15.12) のように上限値を指定しない場合、配列型は不完全な型になります。式が “[ ]”で囲まれていない場合、下限値には既定値のゼロが使用されます。次に例を示します。

```
/* incomplete array completed by external linkage
   same as extern int a[0:], b[0:][5]; */
extern int a[], b[][5];
extern int c[1:], b[1:][1:5]; /* completed by external linkage */
void funct1(int e[]);         /* completed by function call */
void funct2(int f[][5]);      /* completed by function call */
void funct3(int g[1:]);       /* completed by function call */
void funct4(int h[1:][1:5]);  /* completed by function call */
void funct5(int i[1:][5]);    /* completed by function call */
int j[] = {1,2,3};            /* completed by initialization */
int k[][2] = {{1,2}, {3,4}};  /* completed by initialization */
int l[1:] = {1,2,3};          /* completed by initialization */
int m[1:][2] = {{1,2}, {3,4}};/* completed by initialization */
int a[3], b[4][5];            /* external linkage */
int c[1:3], b[1:4][1:5];      /* external linkage */
```

関数 `funct3()`、`funct4()`、および `funct5()` に下限値を指定して配列を渡す方法の詳細については、次のセクションで説明します。

配列は、下限値を指定せずに、上限値だけで宣言してはなりません。次に例を示します。

```
int a[:5];              /* ERROR: without lower bound */
int funct(int b[:5]);   /* ERROR: without lower bound */
```

C ではポインタと配列は密接に結び付いています。また、式内の配列の変数名は、配列の先頭の要素に使用されるメモリへのポインタにもなります。ポインタと配列とのこの強い結び付きは保持されます。配列の下限値がゼロの場合、ポインタとしての配列名の意味は変わりません。たとえば、添字はポインタからのオフセットと同等です。

```
int a[5], b[0:4], *p;
p = &b[0];              /* p = b */
*(a+0) = *(b+0);        /* a[0] = b[0] */
*(a+4) = *(b+4);        /* a[4] = b[4] */
*(p+1) = p[1]*2;        /* b[1] = b[1]*2 */
```

ただし、配列の下限値がゼロ以外の場合は、配列の添字付けとポインタ演算の間に違いが生じます。添字は、ポインタからのオフセットから配列の下限値を引いた値と等しくなります。次に例を示します。

```
#define i 1
int b[i:5], *p, j=3;
p = &b[i];              /* p = b */
*(b+j) = b[j];          /* b[j+i] = *(b+j-i) */
*(p+j) = b[j];          /* p[j]   = *(b+j-i) */
```

同じ原理が、多次元配列にも適用されます。次に例を示します。

```
#define n 1
#define m 2
int a[n:8][m:9], i=3, j=4;
a[i][j] = 90;
*(&a[i][j]) = 90;                        /* a[i][j] = 90 */
*(a[i]+j-m) = 90;                        /* a[i][j] = 90 */
*(*(a+i-n)+j-m) = 90;                    /* a[i][j] = 90 */
*((int *)a[i]+j-m) = 90;                 /* a[i][j] = 90 */
*((int *)(a+i-n)+j-m) = 90;              /* a[i][j] = 90 */
*((int *)a+(i-n)*(9-m+1)+j-m) = 90;      /* a[i][j] = 90 */
*(&a[n][m]+(i-n)*(9-m+1)+j-m) = 90;      /* a[i][j] = 90 */
i = a[i+1] - a[i];                       /* i = 9-m+1  is 8 */
```

同じように、上限値および下限値を明示したポインタを扱うことができます。次に例を示します。

```
int a[3][1:5], b[0:5][1:5];
int (*p)[1:5];
p = a;          /* p[i][j] = a[i][j] */
p = b;          /* p[i][j] = b[i][j] */
```

ここで、p は下限値 1 を含む 10 個の要素を持つ配列へのポインタです。形状引継ぎ配列へのポインタを使用して、ポインタ p が配列 a をポイントするときに、要素 p[i][j] と要素 a[i][j] とが同じオブジェクトを参照するようにする方法については、次のセクションで説明します。また、要素 p[i][j] と要素 b[i][j] は、同じポインタ p が配列 b をポイントするときにも、同じオブジェクトを参照します。

上限値と下限値を明示した配列は、キャスト演算に使用できます。次に例を示します。

```
int a[3][1:5], b[1:5][2:6];
int (*p)[1:5];
p = (int (*)[1:5])a;          /* p[i][j] == a[i][j] */
p = (int (*)[1:5])b;          /* p[i][j] == b[i][j+1] */
p = (int (*)[1:5])malloc(3*5*sizeof(int)); free(p);
p = (int [][1:5])malloc(3*5*sizeof(int));  free(p);
p = (int [0:][1:5])malloc(3*5*sizeof(int));
```

上限値と下限値を明示した配列は、typedef 宣言に使用できます。次に例を示します。

```
typedef int A[1:5];
A a;                 /* int a[1:5] */
```

2 つの配列型に互換性を持たせるには、互換性のある要素型を両方の配列に持たせ、両者を同じ形状にする必要があります。2 つの配列の形状が同じであると言えるのは、2 つの配列の各次元で添字の下限値と上限値が同じになっている場合のみです。次に例を示します。

```
extern int a[3], c[0:2], b[1:5];
int a[0:2], c[3];        // OK
int b[5];                // ERROR
int funct(int aa[1:3]);
int funct(int aa[0:2]);  // ERROR: change array bounds
int e[3][1:5], f[10][1:5], g[3][5], h[1:3][1:5], i[3][0:5];
int (*p)[1:5];
p = e; // OK: compatible
p = f; // OK: compatible
p = g; /* incompatible second dimension p[i][j+1] == g[i][j],
           no warning or error message */
p = h; /* incompatible first dimension p[i][j] == h[i+1][j],
          no warning or error message */
p = i; // WARNING: incompatible second dimension p[i][j+1] != i[i][j]
```

要素 p[i][j+1] と要素 g[i][j] は、配列の 2 番目の次元の値 5 のエクステントが同じであるため、同じオブジェクトを参照します。しかし、要素 p[i][j+1] と要素 i[i][j] は同じオブジェクトを参照しません。

15.5.2 可変添字範囲の配列

ここまでのセクションでは、プログラム実行時のみにサイズが確定される可変長の配列について説明しました。可変長配列型には、形状無指定配列、形状引継ぎ配列、および形状引継ぎ配列へのポインタなどがあります。このセクションでは、上限値および下限値を明示することに関して、この可変長配列型をさらに詳しく説明します。これまでのセクションで説明した、形状無指定配列と形状引継ぎ配列へのポインタの構文と意味はすべて有効です。形状引継ぎ配列については、意味に変更はありませんが、構文は変更されています。これについては次のセクションで説明します。

添字範囲無指定配列

配列の添字の上限値または下限値が整数型の非定数式である場合、プログラム実行時に評価され、配列型は**添字範囲無指定配列**となります。次に例を示します。

```
int funct(int n, int m) {
  int i = n;
  int a[n:m], b[i:m], c[-n:2*m][i:n+m];
  int d[1:n], e[n:10], f[1:5][0:n];
}
```

ここで、a、b、c、d、e、f は、添字範囲無指定配列です。添字範囲無指定配列は、実行時に、上限値が下限値より大きい値であると評価される必要があります。次に例を示します。

```
int funct(int n, int m) {
  int a[n:m];
}
funct(1,5);                 // OK: int a[1:5]
funct(5,1);                 // ERROR: int a[5:1]
funct(5,5);                 // ERROR: int a[5:5]
```

また、添字範囲無指定配列が形状無指定配列であるため、これまでのセクションで説明した形状無指定配列に関する制約と意味は、すべて添字範囲無指定配列に適用できます。たとえば、添字無指定範囲配列へのポインタは宣言してはなりません。

```
/* ERROR: pointer to deferred-shape array */
int funct(int n, int m, int a[n:m], int (*b)[n:m]) {
  int (*p1)[n:m];          // ERROR: pointer to deferred-shape array
  int (*p2)[1:m];          // ERROR: pointer to deferred-shape array
  int (*p3)[n][1:m];       // ERROR: pointer to deferred-shape array
}
```

添字範囲無指定配列を不完全な配列型と混在させてはなりません。次に例を示します。

```
int n=4, m=5;
int a[][n:m]={{1,2}, {3,4}};          // ERROR: initialization
int b[1:][n:m]={{1,2}, {3,4}};        // ERROR: initialization
int funct(int n, int m, c[][n:m]);  // ERROR: func parameter scope
int funct(int n, int m, d[1:][n:m]);// ERROR: func parameter scope
int funct(int n, int m, e[n:][n:m]);// ERROR: func parameter scope
extern int f[][n:m];              // ERROR: static storage duration
extern int g[1:][n:m];            // ERROR: static storage duration
```

形状引継ぎ配列へのポインタ

固定長配列へのポインタを宣言する場合は、次の形式で行う必要があります。

$$\texttt{T (*D)[expr]} \tag{15.13}$$

ここで、`T` には型を指定する宣言指定子を指定します。`D` は識別子 `ident` を含む宣言子です。式 `expr` は、整数定数式でなければなりません。形状引継ぎ配列へのポインタを宣言する場合は、次の 2 つのいずれかの形式で行う必要があります。

$$\texttt{T (*D)[:]} \tag{15.14}$$

$$\texttt{T (*D)[lower:]} \tag{15.15}$$

配列の下限値を示す式 `lower` は、整数定数式でなければなりません。次に例を示します。

```
int n=10;
int (*p1)[:];         /* OK */
int (*p2)[:][:];      /* OK */
int *(*p3)[:][:];     /* OK */
int (*p4)[3][:];      /* ERROR: mix with fixed-length */
int (*p5)[n][:];      /* ERROR: mix with deferred-shape */
int (*p5)[n:];        /* ERROR: mix with deferred-shape */
int (*p6)[ ][:];      /* ERROR: mix with incomplete */
```

配列の形状が形状引継ぎ配列へのポインタによって引き継がれる場合、形状引継ぎ配列の添字の下限値と上限値のどちらも引き継がれます。次に例を示します。

```
int n=3, m=4;
int a[3][4], b[1:n][1:m], c[3][1:4];
int (*p)[:];
p = a;          /* p[i][j] == a[i][j] */
p = b;          /* p[i][j] == b[i][j] */
p = c;          /* p[i][j] == c[i][j] */
```

下限値が指定されている宣言 (15.15) は、関数パラメータスコープ外で使用してはなりません。次に例を示します。

```
int a[1:3][1:4];
int (*p1)[1:4];      /* OK: pointer to fixed-length array */
int (*p2)[1:];       /* ERROR: pointer to incomplete array
                               not at function parameter scope */
p1 = a;              /* OK */
p2 = a;              /* ERROR */
```

この例では、変数 p1 は、オフセット 1 から始まる int 型の 3 つの要素を持つ配列へのポインタです。変数 p2 の宣言は無効です。p2 が無効であるため、配列の下限値の代入ステートメント p2 = a には、一貫した文法を構成できません。問題は、p2 などのポインタ変数の場合、ポインタを間接的に計算して求めた添字の下限値を、宣言の指定に従って、明示的に変数の宣言に提供することはできないということです。したがって、一貫性を保てるよう、形状引継ぎ配列へのポインタに添字の下限値を指定してはなりません。ただし、関数パラメータスコープで形状引継ぎ配列へのポインタを宣言する場合を除きます。これについては、次のセクションで説明します。これまでのセクションで説明した形状引継ぎ配列へのポインタに関するその他の制約と意味はすべて有効です。たとえば、動的に割り当てられた配列にアクセスするために、形状引継ぎ配列へのポインタを使用できます。

```
int funct(int n, int m) {
  double a[1:n][1:m];
  /* OK */
  double (*p1)[:] = a;
  double (*p2)[:] = (double [1:n][1:m])a;
  double (*p3)[:] = (double [1:n][1:m])malloc(n*m*sizeof(double));
  double (*p4)[:] = (double [1:][1:m])malloc(n*m*sizeof(double));
  double (*p5)[:] = (double [ ][1:m])malloc(n*m*sizeof(double));
  /* ERROR */
  double (*p6)[:] = (double (*)[1:m])malloc(n*m*sizeof(double));
}
```

この例では、キャスト演算 (double [ ][1:m]) は、(double [0][1:m]) または (double [0:][1:m]) と同じです。funct() の最後のプログラミングステートメントでは、形状無指定配列へのポインタが誤って使用されています。

15.6 上限値と下限値を明示した配列を関数に渡す方法

このセクションでは、上限値と下限値を明示した配列を関数に渡す処理の言語的な特徴について説明します。このセクションで示す構文と意味は、すべて C 規格と既存の C コードを無効にするものではありません。

15.6.1 固定添字範囲の配列を渡す方法

固定添字範囲の配列を関数に渡す場合、呼び出された関数に実際に渡された配列引数に、関数パラメータスコープ内で宣言された配列引数と互換性を持たせる必要があります。どちらにも互換性のある要素型を持たせ、同じ形状にする必要があります。次に例を示します。

```
int a[1:3][1:5], b[0:3][1:5], c[3][1:5], d[1:3][1:6], e[1:3];
float f[1:3][1:5];
int funct(int aa[1:3][1:5]);
funct(a);    /* OK */
funct(b);    /* WARNING: incompatible first dimension */
funct(c);    /* incompatible first dimension c[i][j] == aa[i+1][j]
                no warning or error message  */
funct(d);    /* WARNING: incompatible second dimension */
funct(e);    /* WARNING: incompatible shape */
funct(f);    /* WARNING: incompatible data type */
```

配列の添字の下限値を指定しない場合は、既定値のゼロが使用されます。配列の先頭の次元は関数呼び出し func(c) で互換性はありませんが、警告メッセージは表示されません。これは、関連付けられた配列のエクステントが同じであれば、呼び出し元関数と呼び出された関数との間に意味のある関係を確立できるためです。エクステントが異なる場合は、互換性がないことを示す警告メッセージが表示されます。次に例を示します。

```
int a[3][1:5], b[0:2][1:5], c[1:3][1:5];
int funct1(int aa[3][1:5]);
int funct2(int (*bb)[1:5]);
funct1(a);  /* OK */
funct1(b);  /* OK */
funct1(c);  /* incompatible first dimension c[i+1][j] == aa[i][j]
               no warning or error message  */
funct2(a);  /* OK */
funct2(b);  /* OK */
funct2(c);  /* incompatible first dimension c[i+1][j] == bb[i][j]
                no warning or error message  */
```

関数パラメータの宣言内の配列名は、配列の先頭の要素へのポインタとして扱われます。ただし、配列名は、関数パラメータで配列の下限値を指定する場合にも使用できます。関数パラメータスコープ内では、不完全な配列型を使用できます。不完全な配列は、関数呼び出し時に定義が完了されます。次に例を示します。

```
int a[1:5], b[1:10], c[0:5], d[3];
int funct(int aa[1:]);
funct(a);    /* OK */
funct(b);    /* OK */
```

```
funct(c);   /* Ok */
funct(d);   /* OK */
```

関数パラメータスコープ内で不完全な 1 次元配列を形状引継ぎ配列へのポインタとして扱うケースについては、次のセクションで説明します。したがって、添字範囲の異なる配列を不完全な 1 次元配列に渡すことも互換性があります。w また、不完全な配列型は多次元配列にも使用できます。関連付けられた不完全な多次元配列の先頭の次元のエクステントの互換性はチェックされません。次に例を示します。

```
int a[1:3][1:5], b[1:2][1:5], c[0:3][1:5], d[1:3][5], e[1:3][0:5];
int funct(int aa[1:][1:5]);
funct(a);   /* OK */
funct(b);   /* OK */
funct(c);   /* incompatible first dimension c[i][j] == aa[i+1][j]
               no warning or error message  */
funct(d);   /* incompatible second dimension d[i][j] == aa[i][j+1]
               no warning or error message  */
funct(e);   /* WARNING: incompatible second dimension */
```

可変長配列が固定添字範囲の配列に渡される場合、形状の互換性を実行時にチェックすることが可能です。次に例を示します。

```
int n = 3, m = 4;
int a[1:n][1:m], b[n][1:m], c[0:n][1:m], d[1:n][0:m], e[1:n][1:m+1];
int funct(int aa[1:3][1:4]);
funct(a);   /* OK */
funct(b);   /* incompatible first dimension b[i][j] == aa[i+1][j]
               no warning or error message  */
funct(c);   /* WARNING: incompatible first dimension */
funct(d);   /* WARNING: incompatible second dimension */
funct(e);   /* WARNING: incompatible second dimension */
```

現在の実装では、ランタイム時のチェックはできません。したがって、上記のプログラムに示されている警告メッセージは表示されません。形状引継ぎ配列へのポインタの形状が実行時に引き継がれるため、互換性もランタイム時にチェックすることが可能です。同じ理由から、次のサンプルコードでは警告メッセージは表示されません。

```
int n = 3, m = 4;
int a[1:3][1:4], b[3][1:4], c[0:3][1:4], d[1:n][1:m], e[1:n][0:m];
int (*p)[:];
int funct(int aa[1:3][1:4]);
p = a;
funct(p);   /* OK */
p = b;
```

```
funct(p);   /* incompatible first dimension p[i][j] == aa[i+1][j]
               no warning or error message  */
p = b;
funct(p);   /* WARNING: incompatible first dimension */
p = d;
funct(p);   /* OK */
p = e;
funct(p);   /* WARNING: incompatible second dimension */
```

15.6.2 形状引継ぎ配列へのポインタを使用して可変長添字範囲の配列を渡す方法

前のセクションでは、不完全な配列型に渡された配列の先頭の次元の上限値を除き、関数に渡される配列の形状は固定されています。このセクションでは、下限値と上限値を明示した可変長配列を渡すためのリンケージについて説明します。

添字範囲が可変の可変長配列を関数に渡すには、形状引継ぎ配列へのポインタを使用できます。関数パラメータスコープで、形状引継ぎ配列へのポインタを宣言する場合は、次の形式で行います。

```
T (*D)[lower:]      (15.16)
T (*D)[:]           (15.17)
T D[lower:]         (15.18)
T D[:]              (15.19)
```

ここで、`T` は型を指定する宣言指定子、`D` は識別子 *ident* を含む宣言子、整数定数型の `lower` は配列の下限値をそれぞれ示します。宣言 (15.18) では、関数の引数に配列パラメータの先頭の次元の下限値を指定できます。宣言 (15.17) および (15.19) のように下限値が指定されていない場合は、既定値のゼロが使用されます。つまり、`T (*D)[:]` は `T (*D)[0:]` と同じで、`T D[:]` は `T D[0:]` と同じになります。これまでのセクションで説明した形状引継ぎ配列へのポインタに関する言語的な特徴は、すべて、下限値がゼロであるかのように配列境界が明示された形状引継ぎ配列へのポインタに適用できます。したがって、以降の説明では、配列境界の明示に関連する新機能だけを説明します。関数パラメータの宣言内の配列名は、配列の先頭の要素へのポインタとして扱われます。宣言 (15.18) では、関数の引数に配列パラメータの下限値が指定されています。次に例を示します。

```
int funct1(int a[1:]);        // OK: pointer to assume-shape (pass)
int funct2(int a[1:][1:]);    // OK:
int funct3(int a[1:][:]);     // OK: pass a[1:][0:]
int funct4(int a[:][1:]);     // OK: pass a[0:][1:]
int funct5(int a[:][:]);      // OK: pass a[0:][0:]
int funct6(int a[][:]);       // OK: pass a[0:][0:]
int funct7(int a[][1:]);      // OK: pass a[0:][1:]
int funct8(int (*a)[1:]);     // OK: pass a[0:][1:]
int funct9(int (*a)[:]);      // OK: pass a[0:][0:]
```

```
int funct11(int a[0:]);         // OK:
int funct12(int a[]);           // OK: incomplete array type as pass
                                // the same as int funct11(int a[0:]);
int funct13(int a[:][5]);       // OK: incomplete array type a[0:][5]
int funct14(int a[1:][5]);      // OK: incomplete array type
int funct15(int a[1:][1:5]);    // OK: incomplete array type
int funct16(int a[1:][1:5]);    // OK: incomplete array type
/* ERROR: fixed-length array no upper bound */
int funct17(int a[1:5][1:]);
int funct18(int a[5][1:]);
int funct19(int a[1:][1:][5]);
int funct20(int a[:5]);         // ERROR: upper bound only
int funct21(int n, int m, int a[n:m]);// ERROR: deferred-shape array
int a[:], b[:][:];        // ERROR: not in function prototype scope
```

関数パラメータ内の不完全な 1 次元配列は、上記の例の funct12() に示されているように、内部的には形状引継ぎ配列へのポインタとして扱われます。ただし、不完全な 1 次元配列は、外部リンケージと初期化においては、固定長配列として扱われます。

形状引継ぎ配列へのポインタへ下限値の異なる配列を渡しても、互換性がないとは見なされません。呼び出された関数内の形状引継ぎ配列へのポインタの上限値は、関数呼び出し時に調整されます。上限値は、渡された配列のエクステントと、関数パラメータで宣言された形状引継ぎ配列へのポインタの下限値との合計になります。たとえば、次のコード例

```
#define low 1
int n = 3, m = 5;
int a[n:m], b[n:2*m];
int funct(int aa[low:]);
funct(a); /* OK */
funct(b); /* OK */
```

では、関数 funct() 内の配列 aa の下限値は 1 です。上限値は 4 で、funct(a) の関数呼び出しの low+m-n+1 と等しくなります。funct(b) の関数呼び出しでは、関数内の配列 aa の下限値は同じく 1 ですが、上限値は 9 になり、low+2*m-n+1 と等しくなります。

上限値の動的調整によって、添字範囲が異なる配列を関数に渡すことが可能になります。これは、前のセクションで説明した固定添字範囲の配列では実現できません。形状引継ぎ配列へのポインタを使用して、上限値だけが、追加パラメータとして明示的に関数に渡される必要があります。この動的な機能は、数値計算で有用です。

たとえば、オフセット 1 から始まるインデックスを持つ配列をパラメータに含む FORTRAN 関数を移植する場合、オフセット 0 から始まるインデックスを持つ従来の C 配列とオフセット 1 から始まるインデックスを持つ FORTRAN スタイルの配列の両方を渡すことによって、関数を呼び出すことができます。次に例を示します。

```
int n=3, m=4;
```

```
int a[n][m], b[1:2*n][1:2*m];
int funct(int aa[1:][1:], int n, int m) {
  int i, j;
  for(i=1; i<=n; i++)
    for(j=1; j<=m; j++)
      aa[i][j] += 2;
}
funct(a, n, m);           /* passing a[0:2][0:3] */
funct(b, 2*n, 2*m);       /* passing b[1:6][1:8] */
```

同様に、オフセット 1 から始まるインデックスを持つ配列の引数で、オフセット 0 から始まるインデックスを持つ配列のパラメータを含む関数を呼び出すことができます。次に例を示します。

```
int n=3, m=4;
int a[n][m], b[1:2*n][1:2*m];
int (*p)[:] = a;
int funct(int aa[:][:], int n, int m);
int funct(int   [:][:], int,   int  );   /* OK */
int funct(int bb[:][:], int l, int r);   /* OK */
int funct(int aa[:][:], int n, int m) {
  int i, j;
  for(i=0; i<=n-1; i++)
    for(j=0; j<=m-1; j++)
      aa[i][j] += 2;
}
funct(a, n, m);           /* passing a[0:2][0:3] */
funct(p, n, m);           /* passing a[0:2][0:3] */
funct(b, 2*n, 2*m);       /* passing b[1:6][1:8] */
```

また、上記のプログラムでは、形状無指定配列の `a` と `b` などの可変長配列と、形状引継ぎ配列 `p` へのポインタを、関数の引数にある形状引継ぎ配列 `aa` へのポインタに渡すことができることが示されています。上記の例では、関数プロトタイプに異なる構文形式が使用されています。

FORTRAN で一般的なプログラミングスタイルの 1 つは、実引数である配列の要素を含む関数を参照呼び出しで呼ぶことによって、配列のセグメントを関数に渡すことです。この種の FORTRAN コードを移植する方法を、以下の例に示します。

```
int n=10;
double X[1:n];
void funct(double A[1:], int n);
funct(&X[5], n);
```

プログラム 15.2の要素 `pa[i+1][j+1]` と要素 `a[i][j]` は、同じオブジェクトを参照します。

```
#include <stdio.h>
int main() {
  int oldrlow = 0, oldrup = 3, oldclow = 0, oldcup = 5;
  int newrlow = 1, newrup = 4, newclow = 1, newcup = 6, i, j;
  double  a[oldrlow:oldrup][oldclow:oldcup], (*pa)[:];
  void funct(double aa[1:][1:], int rup, int cup);

  pa = (double [newrlow:newrup][newclow:newcup])a;
  for(i=oldrlow; i<=oldrup; i++)
    for(j=oldclow; j<=oldcup; j++)
      a[i][j] = 2;
  funct(pa,newrup,newcup);
  for(i=newrlow; i<=newrup; i++)
    for(j=newclow; j<=newcup; j++)
      printf("pa[i][j] = %f \n", pa[i][j]);
}

void funct(double aa[1:][1:], int rup, int cup) {
  int i, j;

  for(i=1; i<=rup; i++)
    for(j=1; j<=cup; j++)
      aa[i][j] += 2;
}
```

プログラム 15.2: プログラム 15.2: 配列の添字範囲の変更

プログラム 15.2の次の関数呼び出し

```
funct(pa,newrup,newcup);
```

は、

```
funct(a,newrup,newcup);
```

または、

```
funct((int [newrlow:newrup][newclow:newcup])a,newrup,newcup);
```

のどちらかで置き換えることができます。

第16章　計算配列と行列計算

Cの配列はポインタと密接に結び付いています。Chの配列と区別する目的で、これらの配列を*C*
配列と呼びます。Chには、数値計算とデータ分析のために、多くの情報を扱えるファーストクラスオブジェクトである計算配列が実装されています。**計算配列**を扱うために、算術演算子などの多くの演算子がオーバーロードされます。

$\mathbf{A}_1$と$\mathbf{A}_2$の2つの配列がある場合、一般に配列式$\mathbf{A}_1/\mathbf{A}_2$は、線形代数では数学的に定義されません。これとは対照的に、Fortran 90では、$\mathbf{A}_1/\mathbf{A}_2$は要素単位の除算として定義されますが、MATLABでは、$\mathbf{A}_1$と逆行列$\mathbf{A}_2$の乗算として定義されます。すなわち、$\mathbf{A}_1/\mathbf{A}_2$と$\mathbf{A}_1\mathbf{A}_2^{-1}$は同じになります。除算のためにこのような演算子がオーバーロードされているため、線形代数の学習者は非常に混乱します。こうした混乱から学習者は、連立一次方程式$\mathbf{A}\mathbf{x} = \mathbf{b}$の解答として、$\mathbf{x} = \mathbf{b}/\mathbf{A}$を使用することになるのかもしれません。このような誤りを避けるため、Chの設計で指針にされている原則の1つは、数学の従来の規則に従うということです。

たとえば、Chでは、同じランクの$\mathbf{A}_1$と$\mathbf{A}_2$の2つの行列を要素単位で除算する場合は`A1./A2`と書き、$\mathbf{A}_1$と逆行列$\mathbf{A}_2$の乗算は`A1*inverse(A2)`と書きます。式$s = \mathbf{v}^T\mathbf{A}\mathbf{v}$は、Chでは`transpose(v)*A*v`と書き換えられます。

本章で使用される記法は、表16.1に示します。

表 16.1: 形状とデータ型の表記法

シンボル	意味
	形状
A	char、int、float、double、complex、または double complex のベクトル、行列、または高次元の配列
B	char、int、float、double のベクトル、行列、または高次元の配列
I	char、int の整数データ型を持つベクトル、行列、または高次元の配列
M	char、int、float、double、complex、または double complex の 2 次元行列
V	char、int、float、double、complex、または double complex の 2 次元ベクトル
a	char、int、float、double、complex、または double complex のスカラ
	データ型
b	bool
c	char
s	short
i	int
f	float
d	double
z	complex
p	演算内のオペランドまたは関数内の引数の高次データ型
k	元のオペランドまたは引数と同じデータ型
m	元のオペランドまたは引数と同じデータ型、元のオペランドまたは引数のデータ型が char または int の場合は double
	データ型修飾子
u	符号なし
l	long

デジタル値は、複数の変数のシンボルに従っています。たとえば、シンボル **V**、**V1**、**V2** はベクトルを表し、シンボル **A**、**A1**、**A2** はベクトル、行列、または高次元配列を表します。

16.1 計算配列の宣言と初期化

形状が完全指定された配列では、各次元の添字のエクステントと範囲を完全に指定します。計算配列は型修飾子 **array** で宣言します。この型修飾子は、ヘッダーファイル **array.h** でマクロとして定義されています。計算配列を使用するプログラムには、このヘッダーファイルを組み込みます。コマンドモードでは、型修飾子 **array** は既定でエイリアスとして定義されます。以下の計算配列は完全指定されています。

```
array int a1[10];              // a1[0], ..., a1[9]
array int a2[0:9];             // a2[0], ..., a2[9]
```

```
array int a3[1:10];            // a3[1], ..., a3[10]
array double  a4[10][10];      // a4[0:9][0:9]
array complex a5[1:10][1:10]; // a5[1:10][1:10]
```

ここで、シンボル ‘:’ は、配列の添字の範囲を指定するために使用されています。配列の添字範囲は、既定で 0〜n-1 です。n は演算子 ‘[]’ で囲み、配列のサイズを指定します。

2 つの計算配列で各次元の要素の数が同じ場合は、次のコマンド実行例に示すように、配列を要素単位で割り当てるときに’=’ という割り当て演算子を使用できます。

```
> array double a[0:3]
> array int b[4] = { 0, 1, 2, 3}
> a = b
0.00 1.00 2.00 3.00
>
```

C 配列の場合と同様に、計算配列は宣言時に初期化できます。既定で、計算配列はゼロに初期化されます。次に例を示します。

```
array int a1[3] = {1, 2, 3};
array int a2[3]= { 2.3e3d, 2.2F, 3.D }; // a2 = {1,2,3}, data cast
array int a3[] = {0.0, -0.0, -0.0};      // a3 = {0.0, -0.0, -0.0}
array double a4[][3] = {{1, 2, 3}, {1, 2, 3}};
array double a5[3][3] = {1, 2, 3, 1, 2, 3};
```

16.2　配列参照

16.2.1　配列全体

配列全体にアクセスするために、計算配列の名前を使用できます。たとえば、次のコード

```
array int a[20], b[20];
b = a+b;
```

は、a の各要素に対応する b の要素に a の要素を追加します。配列 a と b は、線形代数と同様に、ベクトルとして扱われます。この機能によって、通常の C 配列を使用するプログラムと比べ、はるかに簡素なプログラムを作成できます。例として、プログラム 16.1と16.2は、配列 a を要素単位で加算しそれを 3 倍して表示する同一のタスクを実行します。

```
/* File: declare.ch */
#include <stdio.h>
#include <array.h>
#define N 2
#define M 3

int main() {
    array int a[N][M] = {1,2,3,
                         4,5,6};
    array int b[N][M];

    b = a+a;
    printf("b = \n%d", b);
    b = 3*a;
    printf("b = \n%d", b);
    return 0;
}
```

プログラム 16.1: 計算配列の宣言と使用

```
/* File: declare.c */
#include <stdio.h>
#define N 2
#define M 3

int main() {
  int a[N][M] = {1,2,3,
                 4,5,6};
  int b[N][M];
  int i, j;

  printf("b = \n",);
  for(i=0; i<N; i++) {
    for(j=0; j<M; j++) {
       b[i][j] = a[i][j]+a[i][j];
       printf("%d ", b[i][j]);
    }
    printf("\n",);
  }
  printf("b = \n",);
  for(i=0; i<N; i++) {
    for(j=0; j<M; j++) {
       b[i][j] = 3*a[i][j];
       printf("%d ", b[i][j]);
    }
    printf("\n",);
  }

  return 0;
}
```

プログラム 16.2: C でのプログラム declare.ch の実装

プログラム 16.1では計算配列を使用していますが、プログラム 16.2では使用していません。明らかに、プログラム 16.2の方が、コード行が少なくて読みやすく、メンテナンスも簡単です。ヘッダーファイル **array.h** で、**array** 修飾子がマクロとして定義されていることに注意してください。計算配列を使用するには、プログラムにこのヘッダーファイルを含める必要があります。プログラム 16.1と 16.2の出力は、次のように同じ結果になります。

```
b =
2 4 6
8 10 12
b =
3 6 9
12 15 18
```

16.2.2 配列の要素

C 配列と同様、計算配列の要素にアクセスするには、演算子 `[n]` を使用できます。この `n` には有効な添字を指定します。たとえば、次のコードは、`a` の 3 番目の要素を `b` の 3 番目の要素に追加し、その結果を `b` の 2 番目の要素に格納します。

```
array int a[20], b[20];
b[1] = a[2]+b[2];
```

C 配列と同じく、計算配列も行単位で配置されます。たとえば、次のように宣言されている計算配列 `B`

```
array int B[2][3];
```

では、図16.1に示す、次元 2x3 の計算配列 B のアドレスが `0x10000` の場合、配列 `B` の内部メモリレイアウトは図 16.2のようになります。

B[0][0]	B[0][1]	B[0][2]
B[1][0]	B[1][1]	B[1][2]

図 16.1: 計算配列 `B`

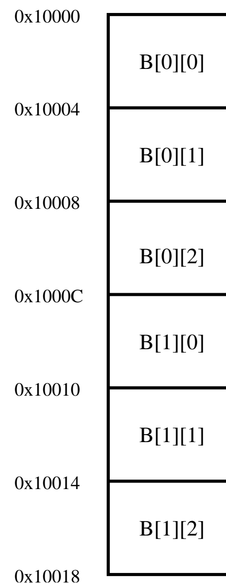


図 16.2: 2 次元計算配列 B のメモリレイアウト

16.3 計算配列の代入形式と出力

C 配列と同様、計算配列の入力は関数 **scanf()** により要素単位で扱うことができます。次に例を示します。

```
> array int  a[2]
> scanf("%d", &a[0])
10
> a
10 0
```

計算配列は、配列全体に対して関数 **scanf()** で扱う場合にも便利です。配列のデータ型が char または unsigned char でない場合、入力する数値は 1 つまたは複数の空白、文字コード ' ' ',' ';' ':' 、または改行コードで区切ることができます。次に例を示します。

```
> array int  b[6]
> scanf("%d", &b)
10 11, 12
13; 14: 15
> b
10 11 12 13 14 15
```

fprintf()、**sprintf()**、**printf()** などの出力関数ファミリを使用すると、計算配列のすべての要素を一度に出力することができます。形式指定子は、配列の各要素に適用されます。次に例を示します。

```
> array int a[3] = {1,2,3}
```

```
> array int b[2][3] = {1,2,3,4,5,6}
> printf("a = %d", a);
a = 1 2 3
> printf("b = \n%d", b);
b =
1 2 3
4 5 6
```

大量の計算配列の場合、配列の各要素を含む文字がスペースで区切られて、1 行あたり 74 文字出力されます。たとえば、次の配列 a の各要素はすべて同じ値 (90) です。74 文字を超えると、出力結果は改行されます。

```
> array int a[2][50] = 90
> a
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
>
```

多次元配列は、次のように、配列の最後の 2 つのエクステントの行と列を含む複数の 2 次元配列で出力されます。

```
> array int a[2][2][3] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12}
> a
1 2 3
4 5 6

7 8 9
10 11 12
```

Ch では、1 次元配列は既定で列ベクトルです。列ベクトルまたは行ベクトルの 1 次元配列では、ベクトルが列ベクトルであっても行ベクトルとして出力されます。次に例を示します。

```
> array int a[3] = {1,2,3}
> a                // column vector
1 2 3
> transpose(a)     // row vector
1 2 3
```

ここでは、ベクトル a が列ベクトルで、a の転置行列”transpose(a)”は行ベクトルです。出力では、どちらも行ベクトルとして表示されます。

16.4　計算配列の明示的データ型変換

計算配列の演算では、オペランドのデータ型に互換性があるかどうかがチェックされます。データ型が一致しない場合、Ch はエラーを示し、プログラムのデバッグに役立つ通知メッセージを出力します。また、 Ch にはデータ型変換規則がいくつか組み込まれており、必要なときにいつでも、それらの規則を呼び出すことができます。これにより、プログラムに明示的な型変換コマンドを多用しなくても済むようになります。計算配列のデータ型の順位は、図 16.3に示すとおり、char 型が最も下位のデータ型で、double complex 型が最も上位のデータ型になっています。

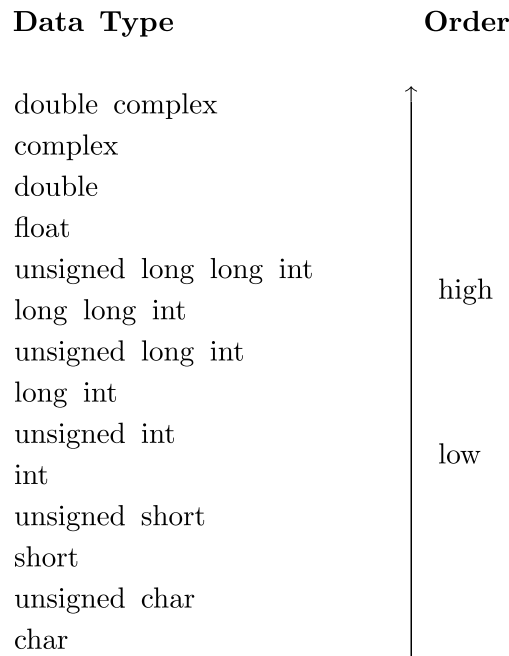


図 16.3: データ型の順位

既定の変換規則の概要は次のとおりです。

1. char 型、int 型、float 型、double 型の配列は、対応するスカラ型のデータ変換規則に従って変換できます。

2. char 型、int 型、float 型、double 型の配列は、各要素の虚数部がゼロである complex 型の配列に変換できます。実数の配列を複素数の配列にキャストする場合、Inf および −Inf の要素の値は ComplexInf になり、NaN の要素の値は ComplexNaN になります。double 型から complex 型に変換すると、情報が失われることがあります。

3. 異なるデータ型が混在する配列での二項演算 (加算、減算、乗算、除算など) では、演算結果は 2 つのオペランドのうちの上位のデータ型になります。たとえば、int 型の配列と double 型の配列との加算結果は、double 型の配列になります。

次のコードは、データ型の異なる配列が自動変換される例を示します。

```
> array int i[2] = {1, 2}
> array float f[2]
> array double d [2]
> f = i             // float = int
1.00 2.00
> d = f + i         // double = float + int
2.0000 4.0000
>
```

`d = f + i` の演算では、結果は float 型の計算配列になります。ここで、配列 `f` と配列 `i` の各要素は、それぞれ float 型と int 型を表しています。float 型の計算配列の結果は、次に、double 型の計算配列にキャストされて double 型の計算配列の変数 `d` に割り当てられます。さまざまな配列演算のデータ型変換については、セクション 16.5で詳しく説明します。

16.5　配列演算

16.5.1　算術演算

計算配列の算術演算は、表 16.2に示すとおりです。

表 16.2: 配列の算術演算

定義	演算	結果
単項プラス	$+\mathbf{A}$	**A**/k
単項マイナス	$-\mathbf{A}$	**A**/k
加算	$\mathbf{A1} + \mathbf{A2}$	**A**/p
加算	$\mathbf{A} + [s]$	**A**/p
加算	$[s] + \mathbf{A}$	**A**/p
減算	$\mathbf{A1} - \mathbf{A2}$	**A**/p
減算	$\mathbf{A} - [s]$	**A**/p
減算	$[s] - \mathbf{A}$	**A**/p
乗算	$\mathbf{A1} * \mathbf{A2}$	**A**/p または a/p
乗算	$\mathbf{A} * s$	**A**/p
乗算	$s * \mathbf{A}$	**A**/p
除算	$\mathbf{A}/s$	**A**/p
配列の乗算	$\mathbf{A1}. * \mathbf{A2}$	**A**/p
配列の除算	$\mathbf{A1}./\mathbf{A2}$	**A**/p
配列の除算	$[s]./\mathbf{A2}$	**A**/p

表 16.2の一番右の欄にあるシンボル **A**/k は、結果が、オペランドと同じ形状で同じデータ型の配列になることを示します。シンボル **A**/p は、結果が、同じ形状で、データ型が 2 つのオペランドのうちの上位のデータ型になることを示します。これらのシンボルについては、表 16.1を参照してくだ

さい。算術演算には、単項プラス演算子 '+'、単項マイナス演算子 '−'、加算演算子 '+'、減算演算子 '−'、乗算演算子 '*'、除算演算子 '/'、配列乗算演算子 '.*'、および配列除算演算子 './' があります。演算子 '*' は、1 次元ベクトルまたは 2 次元行列の 2 つの配列の乗算に使用します。2 つの配列の乗算は、線形代数の規則に従います。要素単位の配列乗算演算子 '.*' と配列除算演算 W 子 './' は、2 つの配列オペランドの対応する各要素の演算に使用します。これらの要素は同じ形状 (次元とエクステント) にする必要があります。

単項プラス演算子または単項マイナス演算子の演算結果のデータ型は、オペランドのテータ型と同じです。表 16.2の他の演算の結果のデータ型は、演算内のオペランドのデータ型よりも上位になります。加算演算子または減算演算子のオペランドの一方がスカラ型で、もう片方が計算配列である場合、スカラ型は、対応する配列演算の計算配列に上位変換されます。配列除算演算子 './' の分子がスカラ型の場合は、計算配列に上位変換されます。

これらの演算のコマンドでの使用例を以下に示します。次に例を示します。

```
> array int a1[2][2] = {1, 0, 2, 3}
> array int a2[2][2] = {0, 5, 2, 2}
> float s = 2.0
> a1 * a2
0 5
6 16
> a1 .* a2
0 0
4 6
> a1/s
0.50 0.00
1.00 1.50
> a1 +2
3 2
4 5
```

2 つの配列の乗算では、次に示すように、配列の次元は線形代数の規則に従う必要があります。

```
> array int a1[2][3] = {1, 2, 3, 4, 5, 6}
> array int a2[3][2] = {1, 2, 3, 4, 5, 6}
> array int b[3] = {1, 2, 3}
> a1*a2
22 28
49 64
> a1*b
14 32
> a1*a1
ERROR: array dimensions do not match for matrix operations
```

特殊なケースとして、たとえば、A1 および A2 の形状が (1 × n) および (n × 1) の場合 (n は 1、2、3... など)、2 つの配列の乗算結果は配列にはならず、スカラになります。次に例を示します。

```
> int i
> array int a[1] = {10}
> array int b[1] = {20}
> array int c[2] = {1, 2}
> i = a * b  // (1x1) * (1x1), the result is a scalar
200
> b = a + b  // it's an array
30
> transpose(c) * c // (1x2) * (2x1), the result is a scalar
5
> c * transpose(c) // (2x1) * (1x2), the result is an array
1 2
2 4
```

`a * b` の結果は整数であり、`transpose(c) * c` も同様です。1 次元配列は既定で、宣言と計算上 $(n \times 1)$ の形状をとる列ベクトルとして扱われます。たとえば、`c` は (1×2) ではなく、(2×1) の形状をとります。

配列乗算演算子 '`.*`' と配列除算演算子 '`./`' は、For ループや While ループなどのループがない計算式を扱う場合に有用です。たとえば、$0.1 \leq x \leq 6.2$ の範囲で関数 $y(x) = 2/x + \sin(x^2)$ のプロットを 100 プロット点で作成するときは、次の式を使用します。

```
> array double x[100], y[100]
> lindata(0.1, 6.2, x)
> y =  2.0./x +sin(x.*x)
> plotxy(x, y)
```

図 16.4にプロットの出力を示します。

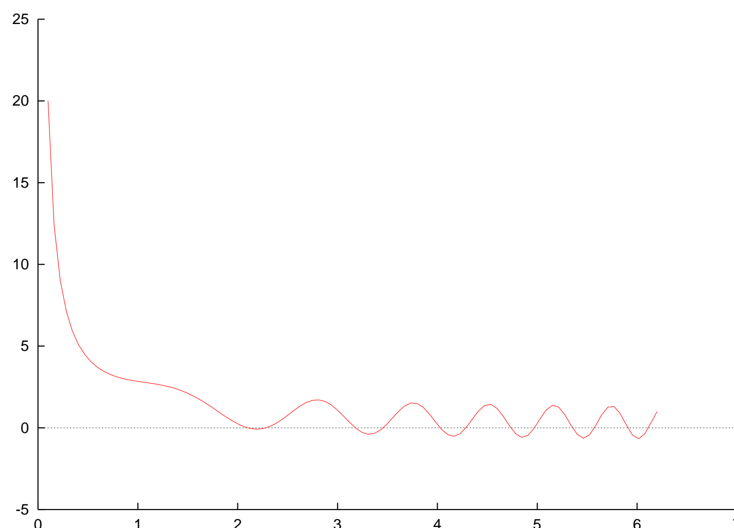


図 16.4: 関数 $y(x) = 2/x + \sin(x^2)$.

lindata(0.1, 6.2, x) の関数呼び出しは、スペースで区切られた 0.1〜6.2 までの等間隔の値（100 個）を配列 x の要素として代入します。配列型の引数を扱うための関数 **lindata()** および汎用数学関数 **sin()** については、後のセクションで詳しく説明します。計算配列 `x` では、式 `2./x` が、配列演算 `2 ./ x` ではなく、`2.0/x` として解釈されることに注意してください。したがって、`2./x` は、配列次元が一致しないため、無効となります。

16.5.2　代入演算

計算配列の代入演算は、表 16.3に示すとおりです。

表 16.3: 配列の代入演算

定義	演算	結果
代入	$\mathbf{A1}$=$\mathbf{A2}$	$\mathbf{A}$/k
assign	$\mathbf{A}$=$[s]$	$\mathbf{A}$/k
assign sum	$\mathbf{A1}$+=$\mathbf{A2}$	$\mathbf{A}$/k
assign difference	$\mathbf{A1}$−=$\mathbf{A2}$	$\mathbf{A}$/k
assign product	$\mathbf{A1}\star$=$\mathbf{A2}$	$\mathbf{A}$/k
assign product	$\mathbf{A1}\star$=s	$\mathbf{A}$/k
assign quotient	$\mathbf{A1}$/=s	$\mathbf{A}$/k

代入演算には、単純代入演算子 ‘=’ と、assign sum 演算子 ‘+=’、assign difference 演算子 ‘−=’、assign product 演算子 ‘$\star$=’、および assign quotient 演算子 ‘/=’ などの複合代入演算子があります。これらの演算子の演算結果のデータ型は、演算子の左側のオペランドと同じデータ型になります。

これらの演算のコマンドでの使用例を以下に示します。

```
   > array int a1[4] = {1, 0, 2, 3}
   > array int a2[4] = {0, 5, 2, 2}
   > a1 += a2
wa   1 5 4 5
```

16.5.3　インクリメント演算とデクリメント演算

計算配列のインクリメント演算とデクリメント演算は、表 16.4に示すとおりです。インクリメント演算とデクリメント演算には、配列の各要素に 1 を加算するインクリメント演算子 ‘++’ と、配列の各要素から 1 を減算するデクリメント演算子 ‘−−’ があります。これらの演算子の結果のデータ型は、元のオペランドと同じデータ型です。

表 16.4: 配列の増分演算と減分演算

定義	演算	結果
増加	**A**++	**A**/k
増加	++**A**	**A**/k
減少	**A**−−	**A**/k
減少	−−**A**	**A**/k

これらの演算のコマンドでの使用例を以下に示します。

```
> array int a1[4] = {1, 0, 2, 3}
> array int a2[4] = {0, 5, 2, 2}
> a1++
1 0 2 3
> a1
2 1 3 4
> --a2
-1 4 1 1
```

多くの場合、計算配列には、1 以上のランクがあります。状況によって、計算配列は NULL の値を持つこともあります。メモリを割り当てる前、計算配列へのポインタには NULL 値が含まれています。NULL 値は、関数内の参照型配列の引数に渡すこともできます。NULL 値を含む計算配列を、等値演算子 '==' または非等値演算子 '!=' の演算子として使用することができます。他の演算子のオペランドとしては使用できません。等値演算子 '==' または非等値演算子 '!=' の 2 つのオペランドのうちの一方が計算配列または参照配列へのポインタになっている場合、もう片方のオペランドは NULL となります。この場合の演算結果は、**true** または **false** のいずれかを示すブール型です。NULL が参照配列に渡されたかどうか、または計算配列へのポインタが有効なオブジェクトを参照しているかどうかテストするのに、これを使用できます。計算配列へのポインタおよび参照の計算配列については、後のセクションで詳しく説明します。

プログラム 16.11では、NULL は、関数 `func()` の参照配列の引数 `a` に渡されます。プログラム 16.12では、計算配列へのポインタの変数 `a` は、配列へポイントされるまでは、NULL の既定値を持っています。これらの 2 つのプログラムの出力結果は、次のようになります。

```
a==NULL is true
a!=NULL is false
```

16.5.4 関係演算

計算配列の関係演算は、表 16.5に示したとおりです。

表 16.5: 配列の関係演算

定義	演算	結果
より小さい	$\mathbf{B1} < \mathbf{B2}$	I/i
より小さい	$\mathbf{B1} < [s]$	I/i
より小さい	$[s] < \mathbf{B2}$	I/i
より小さいか等しい	$\mathbf{B1} <= \mathbf{B2}$	I/i
より小さいか等しい	$\mathbf{B1} <= [s]$	I/i
より小さいか等しい	$[s] <= \mathbf{B2}$	I/i
等しい	$\mathbf{A1} == \mathbf{A2}$	I/i
等しい	$\mathbf{A1} == [s]$	I/i
等しい	$[s] == \mathbf{A2}$	I/i
等しい	$NULL == \mathbf{A1}$	b
等しい	$\mathbf{A1} == NULL$	b
より大きいか等しい	$\mathbf{B1} >= \mathbf{B2}$	I/i
より大きいか等しい	$\mathbf{B1} >= [s]$	I/i
より大きいか等しい	$[s] >= \mathbf{B2}$	I/i
より大きいか等しい	$\mathbf{B1} > \mathbf{B2}$	I/i
より大きいか等しい	$\mathbf{B1} > [s]$	I/i
より大きいか等しい	$[s] > \mathbf{B2}$	I/i
等しくない	$\mathbf{A1}\ != \mathbf{A2}$	I/i
等しくない	$\mathbf{A1}\ != [s]$	I/i
等しくない	$[s] != \mathbf{A2}$	I/i
等しくない	$\mathbf{A1}\ != NULL$	b
等しくない	$NULL != \mathbf{A1}$	b

関係演算には、'より小さい <'、'より小さいか等しい <='、'等しい =='、'より大きいか等しい >='、'より大きい >'、'等しくない !=' などの演算子があります。これらの演算子を使用すると、結果は int 型の配列となり、値は、配列の各要素の比較内容によって 0 または 1 になります。これらの二項演算子では、オペランドの一方が計算配列で、もう片方がスカラである場合、スカラは、配列オペランドの形状を持つ計算配列に上位変換されます。これらの演算のコマンドでの使用例を以下に示します。

```
> array int a1[4] = {1, 0, 2, 3}
> array int a2[4] = {0, 5, 2, 2}
> a1 < a2
0 1 0 0
> a1 >=  a2
1 0 1 1
```

多くの場合、計算配列には、1 以上のランクがあります。状況によって、計算配列は NULL の値を持つこともあります。メモリを割り当てる前、計算配列へのポインタには NULL 値が含まれていま

す。NULL 値は、関数内の参照型配列の引数に渡すこともできます。NULL 値を含む計算配列を、等値演算子 '==' または非等値演算子 '!=' の演算子として使用することができます。

他の演算子のオペランドとしては使用できません。等値演算子 '==' または非等値演算子 '!=' の 2 つのオペランドのうちの一方が計算配列または参照配列へのポインタになっている場合、もう片方のオペランドは NULL となります。この場合の演算結果は、**true** または **false** のいずれかを示すブール型です。これを **if** ステートメントの制御式として使用し、NULL が参照配列に渡されたかどうか、または計算配列へのポインタが有効なオブジェクトをポイントしているかどうかをテストすることができます。計算配列へのポインタおよび参照配列については、後のセクションで詳しく説明します。

16.5.5 論理演算子

計算配列の論理演算は、表 16.6に示すとおりです。論理演算には AND'&&'、XOR'^^'、OR'||'、および NOT'!' の演算子を使用します。これらの演算子による評価結果は、int 型の配列で、値は 0 か 1 のいずれかです。これらの 2 項演算子では、オペランドの一方が計算配列で、もう片方がスカラである場合、スカラは、配列オペランドの形状を持つ計算配列に上位変換されます。

表 16.6: 配列の論理演算

定義	演算	結果
AND	$\mathbf{A1}$ && $\mathbf{A2}$	**I**/i
AND	$\mathbf{A1}$ && $[s]$	**I**/i
AND	$[s]$&& $\mathbf{A2}$	**I**/i
XOR	$\mathbf{A1}$ ^^ $\mathbf{A2}$	**I**/i
XOR	$\mathbf{A1}$ ^^ $[s]$	**I**/i
XOR	$[s]$^^ $\mathbf{A2}$	**I**/i
OR	$\mathbf{A1}$ || $\mathbf{A2}$	**I**/i
OR	$\mathbf{A1}$ || $[s]$	**I**/i
OR	$[s]$|| $\mathbf{A2}$	**I**/i
NOT	!**A**	**I**/i

これらの演算のコマンドでの使用例を以下に示します。

```
> array int a1[4] = {1, 0, 2, 3}
> array int a2[4] = {0, 5, 2, 2}
> a1 && a2
0 0 1 1
> a1 ||  a2
1 1 1 1
```

16.5.6　条件演算

Ch では、条件演算子 '?:' を計算配列に適用できます。この場合、条件式の先頭のオペランドはスカラ型となり、残りの 2 つのオペランドは同じ形状の計算配列となります。結果は、これらの 2 つのオペランドのうち上位のオペランドの型を持つ計算配列になります。
これらの条件演算のコマンドでの使用例を以下に示します。

```
> array int a[2][3] = 1, b[2][3]=2
> array float f[2][3] = 3.0
> 1 ? a:b // operands of array
1 1 1
1 1 1
> 0 ? f:b  // operands of array
2.00 2.00 2.00
2.00 2.00 2.00
```

これらの 2 つの例では、2 番目と 3 番目のオペランドの両方が同じ形状で、ともに (2×3) の値を持っています。後者の例の結果は、2 番目のオペランドの float 型が、3 番目のオペランドの int 型よりも上位にあるため、float 型の計算配列となります。

16.5.7　アドレス演算子

アドレス演算子 '&' を使用すると、計算配列のアドレスや計算配列の要素のアドレスを取得できます。このアドレス演算のコマンドでの使用例を以下に示します。

```
array int a[0:9], b[2][3];
int *ptr;
ptr = &a;         // the address of a
ptr = &a[2]       // the address of third element of a
ptr = &b;         // the address of b
ptr = &b[1][2];   // the address of an element of b
```

計算配列に適用されたアドレス演算子'&' で、配列の先頭の要素のアドレスが得られます。次のサンプルコマンドでは、&a が a[0][0] のアドレスを、&b が b[0][0] のアドレスを取得します。このため、後のセクションで説明するように、計算配列へのポインタ用にメモリが割り当てられていない場合は、アドレス演算の結果は NULL となります。また、計算配列の要素の前にアドレス演算子 '&' を指定すると、この要素のアドレスが得られます。たとえば、以下に示すように、&b[1][0][0] を指定すると b[1][0][0] のアドレスが得られます。

```
> array int a[2][2] = {1, 2, 3, 4}
> array int b[2][2][2] = {1, 2, 3, 4, 5, 6, 7, 8}
> &a
4005e3e0
> &a[0][0]            // same as &a
```

```
4005e3e0
> &b
4005e4e0
> &b[0][0][0]        // same as &b
4005e4e0
> &b[1][0][0]
4005e4f0
```

16.5.8 キャスト演算子

Ch では、型の異なる計算配列の配列演算だけでなく、計算配列と C 配列を含む演算も可能にしているため、混乱を防ぐためにキャスト演算が重要になります。
以下に、計算配列のキャスト演算の例をいくつか示します。

```
array double a[3][1], b[3], c[4][3];
array int d[3][1];
a = (array double [3][1])b;      // cast [3] to [3][1]
b = (array double [3])a;         // cast [3][1] to [3]
b = (array double [3])&c[1][0];  // cast 2nd row of c to vector b
b = (array double [3])&c[2][0];  // cast 3rd row of c to vector b
c = (array double [4][3])4;      // cast scalar to array
d = (array int [3][1])a;         // cast double to int
```

キャスト演算を使用すると、2 つの計算配列の代入演算を実行できます。たとえば、配列 c の最後の次元のエクステントは配列 b と同じになっています。配列 a は、最後の次元のエクステントは異なりますが、配列 b と同じ量のメモリを持っています。スカラは計算配列としてキャストできることに注意してください。上記のステートメント c = (array double [4][3])4 は、配列 c のすべての要素を 4 に設定します。また、上記の最後の演算のように、1 つのデータ型の計算配列を別のデータ型にキャストすることも可能です。
キャスト演算の配列の要素の数がオペランドの配列の要素の数より少ない場合、オペランドの余分な要素は無視されます。キャスト演算の配列の要素数がオペランドの配列の要素数より大きい場合は、結果の配列の残りの要素に 0 が埋められます。次に例を示します。

```
> array double a[3] = {1,2,3}
> (array int [2])a
1 2
> (array int [4])a
1 2 3 0
```

配列の前にキャスト演算子を指定すると、配列の先頭の要素のアドレスまたは値が得られます。型がポインタ型の場合は、配列の先頭の要素のアドレスが得られます。ポインタ型以外の場合は、先頭の要素の値が得られます。次に例を示します。

```
> array int a[2][2] = {1, 2, 3, 4}
> (int *)a
4005ef10
> &a
4005ef10
> (int)a
1
```

16.6 演算でのスカラから計算配列への上位変換

スカラ値を明示的に計算配列にキャストすることができます。しかし、スカラ型オペランドは、加算、減算、配列分割、代入演算、論理演算、および関係演算などの演算で、他のオペランドが計算配列である場合は、暗黙的に計算配列に上位変換されます。配列の上位変換は、2 つの配列オペランドの演算に使用されます。すなわち、スカラのオペランドは、演算の内部処理として、各要素の値がこのスカラ値と同値にされている配列として扱われます。場合によっては、配列を上位変換することで、プログラミングが簡単になります。次のステートメントの例を考えます。ここでは、1 つのステートメントだけで計算配列 a の各要素に 2 を追加しています。a が通常の C 配列の場合なら、この処理にはある種のループが必要になるでしょう。

```
> array int a[2][2] = {1, 0, 2, 3}
> a1 + 2                // 2 is promoted to array
3 2
4 5
```

表 16.7に、暗黙的な配列上位変換による演算を示します。

表 16.7: 配列の上位変換

定義	演算	上位変換	結果
代入	$\mathbf{A} = s$	$\mathbf{A} = [s]$	**A**/k
加算	$\mathbf{A} + s$	$\mathbf{A} + [s]$	**A**/p
加算	$s + \mathbf{A}$	$[s] + \mathbf{A}$	**A**/p
減算	$\mathbf{A} - s$	$\mathbf{A} - [s]$	**A**/p
減算	$s - \mathbf{A}$	$[s] - \mathbf{A}$	**A**/p
除算	s ./ $\mathbf{A}$	$[s]$ ./ $\mathbf{A}$	**A**/p
より小さい	$\mathbf{B} < s$	$\mathbf{B} < [s]$	**I**/i
より小さい	$s < \mathbf{B}$	$[s] < \mathbf{B}$	**I**/i
より小さいか等しい	$\mathbf{B} <= s$	$\mathbf{B} <= [s]$	**I**/i
より小さいか等しい	$s <= \mathbf{B}$	$[s] <= \mathbf{B}$	**I**/i
等しい	$\mathbf{A} == s$	$\mathbf{A} == [s]$	**I**/i
等しい	$s == \mathbf{A}$	$[s] == \mathbf{A}$	**I**/i
より大きいか等しい	$\mathbf{B} >= s$	$\mathbf{B} >= [s]$	**I**/i
より大きいか等しい	$s >= \mathbf{B}$	$[s] >= \mathbf{B}$	**I**/i
より大きい	$\mathbf{B} > s$	$\mathbf{B} > [s]$	**I**/i
より大きい	$s > \mathbf{B}$	$[s] > \mathbf{B}$	**I**/i
等しくない	$\mathbf{A}$!= s	$\mathbf{A}$!= $[s]$	**I**/i
等しくない	s != $\mathbf{A}$	$[s]$!= $\mathbf{A}$	**I**/i
XOR	$\mathbf{B}$ ^^ s	$\mathbf{B}$ ^^ $[s]$	**I**/i
XOR	s ^^ $\mathbf{B}$	$[s]$ ^^ $\mathbf{B}$	**I**/i
OR	$\mathbf{B} \mid\mid s$	$\mathbf{B} \mid\mid [s]$	**I**/i
OR	$s \mid\mid \mathbf{B}$	$[s] \mid\mid \mathbf{B}$	**I**/i
AND	$\mathbf{B}$ && s	$\mathbf{B}$ && $[s]$	**I**/i
AND	s && $\mathbf{B}$	$[s]$ && $\mathbf{B}$	**I**/i

16.7 計算配列を関数に渡す方法

計算配列を関数に渡すには、4 つの方法があります。それらの方法と各方法の特性についての簡単な説明を表 16.8に示します。

表 16.8: 計算配列を関数に渡す方法

メソッド	サンプルコード	次元	エクステント	データ型
完全指定配列	`array double a[2][3]`	固定	固定	固定
形状引継ぎ配列	`array double a[:][:]`	固定	可変	固定
可変長変数	type1 func(type2 a, ...)	可変	可変	可変
固定次元の参照配列	`array double a[&][&]`	固定	可変	可変
参照配列	`array double &a`	可変	可変	可変

固定次元は、指定した次元の配列だけが関数に渡されることを意味します。表に示すサンプルコードでは、4 番目の方法 (参照の配列) を使用しない限り、2 次元配列の引数のみが許容されます。この方法を使用すると、どの次元の配列でも関数の引数に渡すことができます。配列引数のエクステントは、次元の要素の数を示します。完全に指定された配列を除き、他のすべての配列引数型では、エクステント数は可変です。このため、要素の数はこれらの引数の型によって異なります。完全に指定された配列または形状引継ぎ配列の場合、計算配列を関数に渡すときにデータ型を固定する必要がありますが、他の 2 つは必要ありません。

16.7.1 形状が完全指定された配列

プログラム 16.3に、形状が完全指定された配列を関数の引数として使用する方法を示します。

```
/* File: sum1.ch */
#include <array.h>
#define N 2
#define M 3

double sum1(array double a[N][M], array double b[N][M]){
    double sum = 0;
    int i, j;

    b = a + 2 * a;  // b = 3*a
    for(i=0; i<N; i++)
      for(j=0; j<M; j++)
        sum += a[i][j];
    return sum;
}

double main() {
    double sum;
    array double b1[N][M], a1[N][M] = {1, 2, 3,
                                       4, 5, 6};

    sum = sum1(a1, b1);
    printf("b1 = \n%g", b1);
    printf("sum = %g\n",  sum);
    return 0;
}
```

プログラム 16.3: 形状とデータ型が固定されている計算配列を渡す処理

プログラム 16.3では、形状が完全指定された配列の引数を持つ関数 `sum1()` を呼び出し、次の次元 2x3 の行列の式

$$\mathbf{b} = \mathbf{a} + \mathbf{2} * \mathbf{a}, \tag{16.1}$$

を計算して、配列 `a` の各要素の値の合計を返します。形状が完全指定された配列として関数の引数が定義されている場合、配列のアドレスはこの関数に渡されます。図 16.5に、プログラム 16.3の出力結果を示します。

```
b1 =
3 6 9
12 15 18
sum = 21
```

図 16.5: プログラム 16.3の出力結果

16.7.2 形状引継ぎ配列

前のコード例では、形状が完全指定された配列として関数 `sum1()` の引数を宣言しています。この方法は、次元ごとに異なるエクステントを持つ配列を扱うには柔軟性に欠けます。Ch には、可変長

の配列を扱うために形状引継ぎ配列が用意されています。形状引継ぎ配列として引数を宣言すれば、同じ次元を持ちながら、次元ごとに要素の数が異なる配列を扱うことができます。

配列の添字にコロン ':' を使用して宣言する形状引継ぎ配列の例を以下に示します。

```
int funct1(array int a[:][:], b[:]);
int func2(array double c[:]);
```

プログラム 16.3を書き直して、形状引継ぎ配列の引数を持つ関数を使用することができます。プログラム 16.4では、同じ行列の式

$$\mathbf{b} = \mathbf{a} + \mathbf{2} * \mathbf{a}, \tag{16.2}$$

を計算するために、形状引継ぎ配列の 2 つの引数を受け取る関数 `sum2()` を呼び出し、配列 `a` の各要素の値の合計をあわせて返します。

```
/* File: sum2.ch */
#include <array.h>

double sum2(array double a[:][:], array double b[:][:]){
    int n = shape(a, 0), m = shape(a, 1);
    /* or array int dim[2] = shape(a);
          int n = dim[0], m = dim[1]; */
    double sum = 0;
    int i, j;

    printf("n = %d, m = %d\n", n, m);
    b = a + 2 * a;  // b = 3*a
    for(i=0; i<n; i++)
       for(j=0; j<m; j++)
          sum += a[i][j];
    return sum;
}

double main() {
    double sum;
    array double b1[2][3], a1[2][3] = {1, 2, 3,
                                       4, 5, 6};
    array double b2[3][4], a2[3][4] = {1, 2, 3, 4,
                                       5, 6, 7, 8,
                                       9, 10, 11, 12};

    sum = sum2(a1, b1);
    printf("b1 = \n%g", b1);
    printf("sum = %g\n\n",  sum);
    sum = sum2(a2, b2);
    printf("b2 = \n%g", b2);
    printf("sum = %g\n",  sum);
    return 0;
}
```

プログラム 16.4: 形状が異なりデータ型が固定されている計算配列を渡す処理

図 16.6に、プログラム 16.4の出力結果を示します。

```
n = 2, m = 3
b1 =
3 6 9
12 15 18
sum = 21

n = 3, m = 4
b2 =
3 6 9 12
15 18 21 24
27 30 33 36
sum = 78
```

図 16.6: プログラム 16.4の出力結果

関数の引数を形状引継ぎ配列として定義すれば、アドレスだけでなく、配列の境界もこの関数に渡されます。このようにして、次元ごとに要素の数が異なる配列を同じ関数に渡すことができます。

たとえば、プログラム 16.4では、配列の a1 と a2 は同じ次元を持っていますが、エクステントが異なっています。関数 sum2() の同じ引数へそれらの配列を渡すことができます。同様に、エクステントの異なる配列 b1 と配列 b2 も同じ引数に渡されます。図 16.6にプログラム 16.4の出力結果を示します。

汎用関数形 **shape()** を使用して、形状無指定配列の各次元のエクステントを得ることができます。関数 **shape()** の 1 つの引数が配列型の場合、関数が次のようにプロトタイプ化されていたかのように、int 型の計算配列として形状を返します。

```
array int shape(array type [:]...[:])[:];
```

ここで、type は、計算配列のいずれかの有効な型です。関数 **shape()** の引数が 1 次元配列の場合、戻り値はサイズ 1x1 の計算配列となります。したがって、戻り値はスカラにキャストすることができます。また、関数 **shape()** を使用して、配列の指定した次元のエクステントを得ることもできます。この場合、関数が次のようにプロトタイプ化されていたかのように動作します。

```
int shape(array type [:]...[:], int index);
```

次に例を示します。

```
> array int a[3][4], b[5]
> shape(a)
3 4
> shape(a, 0)
3
> shape(a, 1)
4
> shape(b)
5
> (int)shape(b)     // cast 1x1 array to scalar
```

```
5
> (int)shape(shape(a))
2
```

上記の関数の関数呼び出し shape(b) は、サイズ 1x1 の計算配列を返します。式 (int)shape(b) を使用して、それをスカラにキャストすることができます。同様に、式 (int) shape(shape(a)) からスカラ値を得ることができます。

16.7.3 形状無指定配列

Ch では、形状無指定の計算配列がサポートされます。これは、次元ごとに要素の数が異なる配列を実行時に扱う別の方法です。形状無指定配列の場合、整式の配列の添字は実行時に評価されます。形状無指定配列の宣言例を以下に示します。

```
array int A[n][m], B[m];
array double C[m];
```

ここで、n と m は **int** 型の変数です。

プログラム 16.5に、形状無指定配列の関数内での使用方法を示します。関数 defshape() の配列 b の形状は指定されません。配列 b の形状は配列 a の形状から取得されます。プログラム 16.5の出力結果は、図 16.3に示すプログラム 16.5の出力結果と同じです。

```
/* File: defshape.ch */
#include <array.h>
#define N 2
#define M 3

double defshape(array double a[:][:], int n, int m) {
    array double b[n][m];   // b is deferred-shape array
    double sum = 0;
    int i, j;

    b = a + 2 * a;  // b = 3*a
    for(i=0; i<n; i++)
      for(j=0; j<m; j++)
        sum += a[i][j];
    printf("b = \n%g", b);
    return sum;
}

double main() {
    double sum;
    array double a1[N][M] = {1, 2, 3,
                             4, 5, 6};

    sum = defshape(a1, N, M);
    printf("sum = %g\n",  sum);
    return 0;
}
```

プログラム 16.5: 形状無指定の計算配列の使用

16.7.4　可変長の引数の配列

可変長の引数とヘッダーファイル **stdarg** で定義されたマクロを使用して、形状とデータ型の異なる配列を関数に渡すことができます。第 10章のセクション10.7では、プログラム 10.30の関数 **lindata()** で示した呼び出し元関数で、型の異なる配列を変更する方法を説明しています。

プログラム 16.6に、可変長の引数を使用して、形状と型が異なる配列 `a` と配列 `b` を関数 `func()` に渡す方法を示します (出力結果は図 16.7を参照)。

```
#include <stdarg.h>
#include <array.h>

void func(int k, ...) {
    int i, m, n, vacount, num;
    ChType_t dtype;
    void *vptr;
    va_list ap;
    va_start(ap, k);
    vacount = va_count(ap);
    printf("va_count(ap) = %d\n", vacount);
    for(i = 0; i<vacount; i++) {
      if(va_arraytype(ap)==CH_CARRAYTYPE ||
         va_arraytype(ap)==CH_CHARRAYTYPE) {
        printf("va_arraydim(ap)= %d\n", va_arraydim(ap));
        num = va_arraynum(ap);
        printf("va_arraynum(ap)= %d\n", num);
        m = va_arrayextent(ap, 0);
        printf("va_arrayextent(ap, 0)= %d\n", m);
        if(va_arraydim(ap) > 1) {
          n = va_arrayextent(ap, 1);
          printf("va_arrayextent(ap, 1)= %d\n", n);
        }
        if(va_datatype(ap) == CH_INTTYPE) {
          int a[num], *p;
          dtype = va_datatype(ap);
          vptr = va_arg(ap, void *);
          printf("array element is int\n");
          p = vptr;
          printf("p[0] = %d\n", p[0]);
          arraycopy(a, CH_INTTYPE, vptr, dtype, num);
          printf("a[0] = %d\n", a[0]);
        }
        else if(va_datatype(ap) == CH_DOUBLETYPE) {
          array double b[m][n];
          dtype = va_datatype(ap);
          vptr = va_arg(ap, void *);
          printf("array element is double\n");
          arraycopy(&b[0][0], CH_DOUBLETYPE, vptr, dtype, num);
          printf("b = \n%f", b);
        }
      }
      else  if(va_datatype(ap) == CH_INTPTRTYPE)
        printf("data type is pointer to int\n");
    }
    va_end(ap);
}

int main() {
   int i, a[4]={10, 20, 30}, *p;
   array double b[2][3]={1, 2, 3, 4, 5, 6};
   p = &i;
   func(i, a);
   func(i, b,a);
   func(i, p);
}
```

プログラム 16.6: 形状とデータ型の異なる配列を関数に渡す処理

```
va_count(ap) = 1
va_arraydim(ap)= 1
va_arraynum(ap)= 4
va_arrayextent(ap, 0)= 4
array element is int
p[0] = 10
a[0] = 10
va_count(ap) = 2
va_arraydim(ap)= 2
va_arraynum(ap)= 6
va_arrayextent(ap, 0)= 2
va_arrayextent(ap, 1)= 3
array element is double
b =
1.000000 2.000000 3.000000
4.000000 5.000000 6.000000
va_arraydim(ap)= 1
va_arraynum(ap)= 4
va_arrayextent(ap, 0)= 4
array element is int
p[0] = 10
a[0] = 10
va_count(ap) = 1
data type is pointer to int
```

図 16.7: プログラム 16.6の出力結果

これらの配列の内容は、関数 **arraycopy()** を使用して、関数 `func()` 内の配列 `a` と配列 `b` に一時的にコピーされます。また、呼び出し元関数内の渡された配列のメモリは呼び出された関数内で使用でき、配列へのポインタを使用することで直接メモリを使用できます。可変長の引数と配列へのポインタを使用したポリモーフィックな関数の扱い方の詳細については、第 19章のセクション 19.9.3を参照してください。

16.7.5 参照配列

形状とデータ型が異なる配列を渡す場合は、参照配列を使用するのではなく、第 10章のセクション 10.7で説明されている可変長の引数を使用する方法を推奨します。参照配列は旧式になったため、将来、段階的に廃止されます。

可変長の配列を処理するために形状引継ぎ配列を使用する方法についてはこれまで説明しました。長さが異なる配列だけでなく、データ型が異なる配列も扱えるよう、Ch には参照配列が導入されています。参照配列は関数オーバーロードに有効に使用できます。参照配列は、配列の添字であるアンパサンド '&' の符号で宣言します。また、次元、長さ、データ型が異なる配列を扱うのに、添字のない参照配列も使用できます。次の参照配列 `a`、`b`、`c` の宣言

```
int fun(array int a[&], array int b[&][&], array int &c);
```

では、`a` と `b` の参照配列は次元が固定されていますが、`c` の参照配列は次元の制約がありません。参照型の引数に関しては、呼び出される前に、関数は引数で定義されるかプロトタイプ化される必要があります。関数の引数が参照配列として定義されている場合は、アドレスと境界に加え、配列のデータ

型もこの関数に渡されます。このようにして、同じ関数でデータ型が異なる配列を扱うことができます。参照配列を使用するには、関数の引数リストで、最大メモリ要件と最上位のデータ型を持つ配列を宣言する必要があります。たとえば、double 型、float 型、および整数型の配列を扱うには、double 型の参照配列を宣言する必要があります。渡された配列の値は、通常、double 型の計算配列に一時的に代入されます。この一時的な配列は関数内の計算に使用されます。結果を呼び出し元関数に返すには、この一時的な配列を、関数の引数リストで宣言されている配列変数に代入する必要があります。
プログラム 16.7は、参照配列を使用してデータ型の異なる配列を扱う例を示します。

```
#include <array.h>

double sum3(array double a[&][&], array double b[&][&]){
    int n = shape(a, 0), m = shape(a, 1);
    double sum = 0;
    int i, j;
    array double aa[n][m];

    printf("n = %d, m = %d\n", n, m);
    b = a + 2 * a;  // b = 3*a
    aa = a;
    for(i=0; i<n; i++)
      for(j=0; j<m; j++)
        sum += aa[i][j];
    return sum;
}

int main() {
    double sum;
    array double b1[2][3], a1[2][3] = {1, 2, 3,
                                        4, 5, 6};
    array float b2[3][4], a2[3][4] = {1, 2, 3, 4,
                                      5, 6, 7, 8,
                                      9, 10, 11, 12};

    sum = sum3(a1, b1);
    printf("b1 = \n%g", b1);
    printf("sum = %g\n\n",  sum);
    sum = sum3(a2, b2);
    printf("b2 = \n%g", b2);
    printf("sum = %g\n",  sum);
    return 0;
}
```

プログラム 16.7: 形状とデータ型の異なる計算配列を渡す処理

プログラム 16.7では、参照配列の 2 つの引数を受け取る関数 `sum3()` を呼び出して、次の同じ行列の式

$$\mathbf{b} = \mathbf{a} + \mathbf{2} * \mathbf{a}, \tag{16.3}$$

を計算し、配列 `a` の各要素の値の合計を返します。プログラム 16.7では、double 型、float 型、および整数型の配列を扱うため、参照型の配列 `a` と配列 `b` は、double 型として宣言されています。関数 `main()` の配列 `a1` と配列 `a2` は同じ次元を持っていますが、エクステントとデータ型は異なってい

ます。これらの配列は、関数 sum3() の同じ引数に渡すことができます。同様に、エクステントとデータ型の異なる配列 b1 と配列 b2 も同じ引数に渡されます。関数 sum3() の配列 a がまず配列 aa に代入されます。これにより、渡された配列の各要素は、**double** データ型で内部的に加算されます。プログラム 16.7の出力結果は、図 16.6に示すプログラム 16.4の出力結果と同じです。

プログラム 16.8は、参照配列を使用して次元とデータ型の異なる配列を扱う例を示します。プログラム 16.8の関数 sum4() は、参照配列の 2 つの引数と、配列 a の要素の数として **int** 型の 1 つの引数を受け取ります。

```
/* File: sum4.ch */
#include <array.h>
#define N 2
#define M 3

double sum4(array double &a, array double &b, int total_num){
    int i;
    double sum;
    array double aa[total_num];

    b = a + 2 * a;  // b = 3*a
    aa = a;
    for(i=0; i<total_num; i++)
      sum += aa[i];

    return sum;
}

int main() {
    double sum;
    array double b1[N][M], a1[N][M] = {1, 2, 3,
                                       4, 5, 6};
    array float b2[M], a2[M] = {10, 20, 30};


    sum = sum4(a1, b1, N*M);
    printf("b1 = \n%g", b1);
    printf("sum = %g\n\n",  sum);
    sum = sum4(a2, b2, M);
    printf("b2 = \n%g", b2);
    printf("sum = %g\n",  sum);
    return 0;
}
```

プログラム 16.8: 次元とデータ型の異なる計算配列を渡す処理

次元とデータ型が異なる配列 a1 と配列 a2 を同じ引数に渡します。同様に、次元とデータ型が異なる配列 b1 と配列 b2 は、関数 sum4() 内で計算された行列の式の結果を返すために使用されます。図 16.8に 16.8の出力結果を示します。

```
b1 =
3 6 9
12 15 18
sum = 21

b2 =
30 60 90
sum = 60
```

図 16.8: プログラム 16.8の出力結果

添字を使用しても、添字のない参照配列の要素に直接アクセスすることはできません。たとえば、プログラム 16.8の関数 `sum4()` の参照配列 `a` と `b` の要素の後に、`a[2]` や aa`[1][2]` などの添字を続けることはできません。

プログラム 16.8では、関数 `sum4()` の 3 番目の引数に、関数の配列 `a` に渡す配列の要素の数が指定されています。次元の数、各次元のエクステント、および渡された配列の要素の総数は、それぞれ `n = (int)shape(shape(a))`、`dim = shape(a)`、`totnum *= dim[i]` の各式で、プログラム 16.9に示した関数内で求めることができます。

```
/* File: sum5.ch */
#include <array.h>
#define N 2
#define M 3

double sum5(array double &a, array double &b){
    int n, i, total_num;
    double sum;

    b = a + 2 * a;  // b = 3*a
    total_num = 1;
    n = (int)shape(shape(a));  // number of dimensions
    array int dim[n];

    dim = shape(a);             // extent of each dimension
    printf("n = %d\n", n);
    for(i = 0; i < n; i++) {
       printf("dim[%d] = %d\n", i, dim[i]);
       total_num *= dim[i];    // total number of elements
    }
    printf("total_num = %d\n", total_num);
    array double aa[total_num];
    aa = a;
    for(i=0; i<total_num; i++)
      sum += aa[i];

    return sum;
}

int main() {
    double sum;
    array double b1[N][M], a1[N][M] = {1, 2, 3,
                                       4, 5, 6};
    array float b2[3], a2[3] = {10, 20, 30};


    sum = sum5(a1, b1);
    printf("b1 = \n%g", b1);
    printf("sum = %g\n\n",  sum);
    sum = sum5(a2, b2);
    printf("b2 = \n%g", b2);
    printf("sum = %g\n",  sum);
    return 0;
}
```

プログラム 16.9: ランクおよびデータ型の異なる計算配列を渡し、関数 **shape()** を使用するプログラム

プログラム 16.9の出力結果は、図 16.9に示すとおりです。

```
n = 2
dim[0] = 2
dim[1] = 3
total_num = 6
b1 =
3 6 9
12 15 18
sum = 21

n = 1
dim[0] = 3
total_num = 3
b2 =
30 60 90
sum = 60
```

図 16.9: プログラム 16.9の出力結果

プログラム 16.9で、次元と要素の総数を求める出力ステートメントをコメントにした場合、プログラム 16.9の出力結果はプログラム 16.8の出力結果と同じになります。

汎用関数 **elementtype()** を使用して、引数のデータ型を取得することができます。関数 **elementtype()** の引数となるのは、型宣言子、C 配列、計算配列、または参照配列のいずれかです。たとえば、次のような場合、

```
array double a[3][4];
int b[3][4];
```

次の 2 つの関係式が成立します。

```
elementtype(double) == elementtype(a);
elementtype(int) == elementtype(b);
```

多くの場合、複素配列と実数配列とでは数理アルゴリズムが異なります。プログラム 16.10では、関数 `arrayfunc()` は、関数 **elementtype()** を使用することで複素配列と実数配列の両方を扱うことができます。配列の式 $\mathbf{a} + 2\sin(\mathbf{a})$ を計算するために、配列の引数 `a` のデータ型によって、実数関数 `realfunc()` または複素関数 `complexfunc()` のいずれかが関数 `arrayfunc()` 内で呼び出されます。図 16.10にプログラム 16.10の出力結果を示します。

```
#include <array.h>

void complexfunc(array double complex a[:][:], array double complex b[:][:]){
    b = a + 2 * sin(a);
}
void realfunc(array double a[:][:], array double b[:][:]){
    b = a + 2 * sin(a);
}
void arrayfunc(array double complex a[&][&], array double complex b[&][&]){
  int n = shape(a, 0), m = shape(a, 1);
  // or array int dim[2] = shape(a);
  // int n = dim[0], m = dim[1];

  if(elementtype(a) == elementtype(complex) ||
     elementtype(a) == elementtype(double complex)) {
    array double complex aa[n][m], bb[n][m];
    aa = (array double complex [n][m])a;
    complexfunc(aa, bb);
    b = bb;
  }
  else {
    array double aa[n][m], bb[n][m];
    aa = (array double [n][m])a;
    realfunc(aa, bb);
    b = bb;
  }
}

int main() {
  array double complex b1[3][4], a1[3][4] = {1, complex(1,2), 2, 5,
                                             7, complex(3,4), 9, 3,
                                             5, 7, 3, 2};
  array double  b2[2][3], a2[2][3] = {1, 5, 3,
                                      5, 6, 7};
  arrayfunc(a1, b1);
  printf("b1 = \n%.1f", b1);
  arrayfunc(a2, b2);
  printf("\nb2 = \n%.1f", b2);
  return 0;
}
```

プログラム 16.10: データ型の異なる配列を関数に渡す処理

```
b1 =
complex(2.7,0.0) complex(7.3,5.9) complex(3.8,0.0) complex(3.1,0.0)
complex(8.3,0.0) complex(10.7,-50.0) complex(9.8,0.0) complex(3.3,0.0)

complex(3.1,0.0) complex(8.3,0.0) complex(3.3,0.0) complex(3.8,0.0)

b2 =
2.7 3.1 3.3
3.1 5.4 8.3
```

図 16.10: プログラム 16.10の出力結果

また、ポインタ NULL が参照配列の引数として関数に渡される場合は、関数内で引数も NULL と等しくなり、関数 **shape()** は 0 次元の配列を返します。たとえば、関数 `func1()` と `func2()` が次のように定義されているとします。

```
int func1(array double a[&]) {
  if(((int)shape(a)) == 0) {
    printf("shape is zero dimension\n");
  }
  if(a == NULL) {
     printf("a is NULL \n");
  }
  return 0;
}

int func2(array double &a) {
  if((int)shape(shape(a)) == 0) {
    printf("shape is zero dimension\n");
  }
  if(a == NULL) {
     printf("a is NULL \n");
  }
  return 0;
}
```

`func1(NULL)` と `func2(NULL)` の両方の関数呼び出しは次の結果を出力します。

```
shape is zero dimension
a is NULL
```

16.8 値 NULL の計算配列

多くの場合、計算配列には、1 以上のランクがあります。状況によって、計算配列は NULL の値を持つこともあります。メモリを割り当てる前、計算配列へのポインタには NULL 値が含まれていま

す。NULL 値は、関数内の参照型配列の引数に渡すこともできます。NULL 値を含む計算配列は、等値演算子 ‘==’ または非等値演算子 ‘!=’ のオペランドとして使用できます。また、**if** ステートメントやループの制御式としても使用できます。

等値演算子 ‘==’ または非等値演算子 ‘!=’ の 2 つのオペランドのうちの一方が計算配列または参照配列へのポインタになっている場合、もう一方のオペランドは NULL とすることができます。この場合の演算結果は、**true** または **false** のいずれかを示すブール型です。NULL が参照配列に渡されたかどうか、または計算配列へのポインタが有効なオブジェクトをポイントしているかどうかテストするのに、これを使用できます。

計算配列は、**if** ステートメント、**while** ループ、**do-while** ループ、または **for** ステートメントの制御式として使用できます。参照配列または NULL 値を含む計算配列へのポインタが制御式として使用されると、**false** であると評価します。それ以外の使用であれば、配列のすべての要素がゼロであっても、制御式は true であると評価します。

```
/* File: arrayrefnull.ch */
#include <array.h>

void func(array double &a) {
    if(a==NULL) {
        printf("a==NULL is true\n");
    }
    else {
        printf("a==NULL is false\n");
    }
    if(a!=NULL) {
        printf("a!=NULL is true \n");
    }
    else {
       printf("a!=NULL is false\n");
    }
}

int main() {
   func(NULL);
   return 0;
}
```

プログラム 16.11: NULL を参照の計算配列に渡す処理

```
/* File: arrayptrnull.ch */
#include <array.h>

int main() {
    array double *a;

    if(a==NULL) {
        printf("a==NULL is true\n");
    }
    else {
        printf("a==NULL is false\n");
    }
    if(a!=NULL) {
        printf("a!=NULL is true \n");
    }
    else {
       printf("a!=NULL is false\n");
    }
    return 0;
}
```

プログラム 16.12: NULL 値を含む計算配列へのポインタ

プログラム 16.11では、NULL は、関数 `func()` の参照配列の引数 `a` に渡されます。プログラム 16.12では、計算配列へのポインタの変数 `a` は、配列へポイントされるまでは、NULL の既定値を持っています。これらの 2 つのプログラムの出力結果は、次のようになります。

```
a==NULL is true
a!=NULL is false
```

16.9 計算配列を返す関数

関数は、計算配列をファーストクラスオブジェクト (first class object) として返します。計算配列を返す関数の場合、返された配列の関数定義内でのランクと関数内の return ステートメントに続く配列式のランクは同じでなければなりません。

16.9.1 固定長の計算配列を返す関数

固定長の計算配列を返す関数のプロトタイプは、次のとおりです。

```
array datatype funcname(argument_list) [n1]...[nm];
```

ここで、`n1` と `nm` は 2 や 3 などの整数定数で、対応する次元の長さを示します。関数の引数リストの閉じ括弧の後に続くシンボル `[]` の数は、返された計算配列のランクを示します。

プログラム 16.13は、関数が計算配列を呼び出し元関数に返す方法の例を示します。

```
/* File: retfix.ch */
#include <array.h>

int main() {
    array int a[2][3] = {1, 2, 3, 4, 5, 6};
    array int funct(array int a[2][3])[2][3];

    a = funct(a);
    printf("a[1][2] = %d\n", a[1][2]);
    printf("a = \n%d", a);
    return 0;
}

array int funct(array int a[2][3])[2][3] {
    array int b[2][3];

    b = 2*a;
    return b;
}
```

プログラム 16.13: 固定長の計算配列を返す関数

このプログラムの関数 `funct()` は次元 2x3 の行列式の結果を返します。

$$\mathbf{b} = \mathbf{2} * \mathbf{a}, \tag{16.4}$$

この出力結果は、図 16.11に示します。

```
a[1][2] = 12
a =
2 4 6
8 10 12
```

図 16.11: プログラム 16.13の出力結果

16.9.2 可変長の計算配列を返す関数

可変長の計算配列を返す関数のプロトタイプは、次のとおりです。

```
array datatype funcname(argument_list) [:]...[:]
```

関数の引数リストの閉じかっこの後に続くシンボル `[:]` の数は、返された計算配列のランクを示します。

プログラム 16.14は、可変長の計算配列を返す関数の例を示します。

```
/* File: retvla.ch */
#include<array.h>

array int func2(array int a[:])[:] {
    int n = (int)shape(a);
    array int x[n];

    printf("n = %d\n", n);
    x = 2*a;
    return x;
}

int main() {
    array int a[2] = {1, 2};
    array int b[5] = {10, 20, 30, 40, 50};

    a = func2(a);
    printf("a = %d\n", a);
    b = func2(b);
    printf("b = %d", b);
    return 0;
}
```

プログラム 16.14: 可変長の計算配列を返す関数

`func2(a)` と `func2(b)` の関数呼び出しで、返される配列の次元は異なります。出力結果は、図 16.12に示します。

```
n = 2
a = 2 4

n = 5
b = 20 40 60 80 100
```

図 16.12: プログラム 16.14の出力結果

16.10 型の汎用配列関数

セクション 16.7.2で説明されている関数 **shape()** は、配列に関連する汎用関数です。さらに、よく使用される汎用数理関数も、計算配列を扱うためにオーバーロードされます。次元、長さ、およびデータ型の異なる引数を扱う場合、これらの関数はオーバーロードされます。

計算配列型の引数の場合、関数 **abs()** によって各要素の絶対値を含む配列を返します。複素型の引数の場合、各要素には対応する複素数の大きさが含まれています。関数は、次のようにプロトタイプ化されていたかのように扱われます。

```
array int abs(array int a[:]...[:])[:]...[:]
```

```
array float abs(array float a[:]...[:])[:]...[:]
array float abs(array complex a[:]...[:])[:]...[:]
array double abs(array double a[:]...[:])[:]...[:]
array double abs(array double complex a[:]...[:])[:]...[:]
```

次に例を示します。

```
> array int a[2][3] = {-1, 2, 3, -4, -5, 6}
> abs(a)
1 2 3
4 5 6
> array complex b[3] = {complex(3, 4), 4, -5}
> abs(b)
5.00 4.00 5.00
```

数学関数 **acos, acosh, asin, asinh, atan, atanh, ceil, cos, cosh, exp, floor, log, log10, sin, sinh, sqrt, tan, tanh** は、どれも 1 つの引数しか受け取りません。次元、長さ、およびデータ型の異なる引数を扱う場合、これらの関数はオーバーロードされます。入力引数のデータ型が整数型の場合、その引数は float 型に上位変換されて計算されます。配列引数の場合は、次のようにプロトタイプ化されていたかのように動作します。

```
array float func(array float a[:]...[:])[:]...[:]
array double func(array double a[:]...[:])[:]...[:]
array complex func(array complex a[:]...[:])[:]...[:]
array double complex func(array double complex a[:]...[:])[:]...[:]
```

ここで、`func` は上記の数学関数のいずれかです。入力引数のデータ型が整数型の場合、その引数は float 型に上位変換されて計算されます。次に例を示します。

```
> array int a[2][3] = {-1, 2, 3, -4, -5, 6}
> sin(a)
-0.84 0.91 0.14
0.76 0.96 -0.28
> array complex b[3] = {complex(3, 4), 4, -5}
> sin(b)
complex(3.85,-27.02) complex(-0.76,-0.00) complex(0.96,0.00)
```

配列引数の場合、関数 **atan2()** は次のようにプロトタイプ化されていたかのように動作します。

```
array float          atan2(array float y[:]...[:],
                           array float x[:]...[:])[:]...[:]
array double         atan2(array double y[:]...[:],
                           array double x[:]...[:])[:]...[:]
array complex        atan2(array complex y[:]...[:],
```

```
                                 array complex x[:]...[:])[:]...[:]
array double complex atan2(array double complex y[:]...[:],
                                 array double complex x[:]...[:])[:]...[:]
```

関数 **atan2()** には 2 つの引数があります。両方の計算配列のデータ型は同じ型でなければなりません。両方の引数のデータ型が整数型の場合、これらは float 型に上位変換されて計算されます。次に例を示します。

```
> array int   y[4]={1,-2, 3,  -4}
> array float x[4]={5, 6, -7, -8}
> atan2(y, x)
0.20 -0.32 2.74 -2.68
```

関数 **pow(a, x)** の最初の引数が NxN の形状の計算配列で、2 番目の引数が整数型の場合、次のようにプロトタイプされていたかのように、関数は先頭の引数と同じ型と次元の計算配列を返します。

```
array int pow(array int a[:][:], int x)[:][:]
array float pow(array float a[:][:], int x)[:][:]
array double pow(array double a[:][:], int x)[:][:]
array complex pow(array complex a[:][:], int x)[:][:]
array double complex pow(array double complex a[:][:], int x)[:][:]
```

この場合、配列関数 **pow(a, x)** は行列の乗算のように動作します。次に例を示します。

```
> array int a[2][2] = {-1, 2, 3, -4}
> pow(a, 2)
7 -10
-15 22
> a*a
7 -10
-15 22
```

関数 **pow()** の配列の引数が両方とも計算配列型である場合、入力された 2 つの配列の要素がスカラ関数 **pow()** によって計算され、その計算値を対応する各要素に含む配列を返します。この場合、2 つの入力配列のデータ型は、関数が次のようにプロトタイプ化されていたかのように、同じ型でなければなりません。

```
array int            pow(array int y[:]...[:],
                         array int x[:]...[:])[:]...[:]
array float          pow(array float y[:]...[:],
                         array float x[:]...[:])[:]...[:]
array double         pow(array double y[:]...[:],
                         array double x[:]...[:])[:]...[:]
array complex        pow(array complex y[:]...[:],
```

```
                               array complex x[:]...[:])[:]...[:]
   array double complex pow(array double complex y[:]...[:],
                               array double complex x[:]...[:])[:]...[:]
```

次に例を示します。

```
   > array int a[3] = {-1, 2, 3}
   > pow(a, a)
   -1 4 27
```

関数 **real()** と関数 **imag()** は、それぞれ、入力された引数の実部と虚部を与えます。配列引数の場合、これらの関数は次のようにプロトタイプ化されているかのように動作します。

```
   array float func(array float a[:]...[:])[:]...[:]
   array double func(array double a[:]...[:])[:]...[:]
   array float func(array complex a[:]...[:])[:]...[:]
   array double func(array double complex a[:]...[:])[:]...[:]
```

ここで、`func` は **real** または **imag** のいずれかです。入力引数のデータ型が整数型の場合、その引数は float 型に上位変換されて計算されます。次に例を示します。

```
   > array int a[3] = {-1, 2, 3}
   > real(a)
   -1.00 2.00 3.00
   > array complex z[3] = {complex(1,2), complex(-3, -4), complex(0, -6)}
   > real(z)
   1.00 -3.00 0.00
   > imag(z)
   2.00 -4.00 -6.00
```

配列関数 **transpose()** は、1 つまたは 2 つの次元の入力配列の転置行列を返します。入力配列のサイズが N × M の場合、返される配列サイズは M × N です。既定で、1 次元配列は列ベクトルです。入力配列がサイズ N × 1 の列ベクトルであるなら、返される配列はサイズ 1 × N の行ベクトルになります。この逆も同様です。返される配列のデータ型は、次のように関数がプロトタイプ化されていたかのように、入力配列と同じデータ型になります。

```
  array type transpose(array data_type a[:])[:]
  array type transpose(array data_type a[:][:])[:][:]
```

ここで、data_type は、計算配列のいずれかの有効な型です。次に例を示します。

```
   > array float a[2][3]={1,2,3,4,5,6}
   1.00 2.00 3.00
   4.00 5.00 6.00
   > transpose(a)
```

```
1.00 4.00
2.00 5.00
3.00 6.00
> array int b[3] = {1, 2, 3}
> a*b
14.00 32.00
> transpose(b)*b
14
> b*transpose(b)
1 2 3
2 4 6
3 6 9
```

16.11 使用頻度の高い配列関数

Ch には、多くの高度な数値関数が用意されています。これらの関数はヘッダーファイル **numeric.h** でプロトタイプ化されています。このセクションでは、よく使用される数値関数について説明します。

セクション 10.7で説明されている関数 **lindata()** は、ヘッダーファイル **numeric.h** で次のようにプロトタイプ化されています。

```
int lindata(double first, double last, ... /* type a[:]...[:] */);
```

lindata() 関数は、引数 `first` と引数 `last` の入力でそれぞれ指定した初期値と最終値の間を等間隔に区切った連続的なデータを生成します。結果は、データ型の異なる配列型の 3 番目の引数内の呼び出し元関数に戻されます。

正方行列 **A** とその逆行列 $\mathbf{A}^{-1}$ とに対して、$\mathbf{A}^{-1}\mathbf{A} = \mathbf{I}$ および $\mathbf{A}\mathbf{A}^{-1} = \mathbf{I}$ となります (**I** は単位行列)。特異でない限り、関数 **inverse()** は正方行列の逆行列を計算します。関数 **inverse()** は、ヘッダーファイル **numeric.h** で次のようにプロトタイプ化されています。

```
array double inverse(array double x[&][&], ... /* [int *status */)[:][:];
```

関数 **inverse()** で返される行列の次元は、行列の入力引数の次元と同じです。この関数を使用して、連立一次方程式を解くことができます。たとえば、次の連立一次方程式

$$
\begin{aligned}
3x_1 + 6x_3 &= 2 \\
2x_2 + x_3 &= 12 \\
x_1 + x_3 &= 25
\end{aligned}
$$

またはその行列形式

$$
\begin{bmatrix} 3 & 0 & 6 \\ 0 & 2 & 1 \\ 1 & 0 & 1 \end{bmatrix}
\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}
=
\begin{bmatrix} 2 \\ 12 \\ 25 \end{bmatrix}
$$

は次の形式で書くことができます。

$$\mathbf{Ax} = \mathbf{b}$$

$\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$ の解法は、Ch では `x = inverse(A)*b` と書くことができます。解 x_1、x_2、および x_3 は、次のステートメントで求めることができます。

```
> array double a[3][3]={3, 0, 6, 0, 2, 1, 1, 0, 1}
> array double ai[3][3], x[3], b[3]= {2, 13, 25}
> ai = inverse(a)
-0.3333 -0.0000 2.0000
-0.1667 0.5000 0.5000
0.3333 0.0000 -1.0000
> x = ai*b
49.3333 18.6667 -24.3333
```

別の例として、次の 2 つの行列方程式を検討します。

$$(\mathbf{A} + \mathbf{5B^{-1}})\mathbf{x} + \mathbf{2a} = (\mathbf{ab^T})\mathbf{b}, \tag{16.5}$$

$$(\mathbf{5AB})\mathbf{x} + \mathbf{ABy} = \mathbf{Bb}, \tag{16.6}$$

ここで

$$\mathbf{A} = \begin{bmatrix} 1 & 2 & 2 \\ 4 & 4 & 6 \\ 7 & 8 & 9 \end{bmatrix}, \mathbf{B} = \begin{bmatrix} 7 & 8 & 9 \\ 1 & 2 & 2 \\ 4 & 4 & 6 \end{bmatrix}, \mathbf{a} = \begin{bmatrix} 1 \\ 4 \\ 7 \end{bmatrix}, \text{and } \mathbf{b} = \begin{bmatrix} 5 \\ 6 \\ 8 \end{bmatrix},$$

2 つの未知のベクトル $\mathbf{x}$ と $\mathbf{y}$ は、次の式で計算できます。

$$\mathbf{x} = (\mathbf{A} + \mathbf{5B^{-1}})^{-\mathbf{1}}(\mathbf{ab^Tb} - \mathbf{2a}), \tag{16.7}$$

$$\mathbf{y} = (\mathbf{AB})^{-\mathbf{1}}(\mathbf{Bb} - \mathbf{5ABx}), \tag{16.8}$$

プログラム 16.15を使用して、ベクトル $\mathbf{x}$ と $\mathbf{y}$ を計算できます。

```
/* File: matrixeq.ch */
#include <stdio.h>
#include <array.h>    // for array qualifier
#include <numeric.h>  // for inverse()

int main() {
    array double A[3][3] = {{1,2,2},{4,4,6},{7,8,9}};
    array double B[3][3] = {{7,8,9},{1,2,2},{4,4,6}};
    array double a[3] = {1,4,7}, b[3] = {5,6,8}, x[3], y[3];

    x = inverse(A+5*inverse(B))*(a*transpose(b)*b - 2*a);
    y = inverse(A*B)*(B*b - 5*A*B*x);
    printf("  x = %.3f  y = %.3f \n", x, y);
    return 0;
}
```

プログラム 16.15: 連立一次方程式を解くプログラム

プログラム 16.15の出力結果は次のとおりです。

```
x = 51.048 15.170 37.020
y = -544.617 -70.723 13.777
```

ヘッダーファイル **numeric.h** で定義されている関数 **sum()** は、配列のすべての要素を合計します。この関数は、次のようにプロトタイプ化されています。

```
double sum(array double &a, ... /* [array double v[:]] */);
```

配列が 2 次元行列の場合、関数は、各行の合計を計算して 2 番目の引数に任意指定された 1 次元配列に格納することができます。次に例を示します。

```
> double a[3] = {10, 2, 3}
> sum(a)
15.0000
> array double v[3], b[3][2] = {1, 2, 3, 4, 5, 6}
> sum(b, v)
21.0000
> v
3.0000 7.0000 11.0000
```

関数 **sum()** を使用すると、有用な配列関数をいくつか実装できます。たとえば、プログラム 16.16の配列関数 `all()`、`any()`、および `count()` も、関数 **sum()** と配列の上位変換を使用して実装することができます。

```
/* promotion.ch */
#include<array.h>
#include<numeric.h>

array int a[2][3] = {1, 2, 3, 4, 5, 6};
array double b[2][3] = {1, 2, 0, 4, 5, 0};
array int c[2][3];

int all(array double &a) {
    return (int)sum(a != 0) == 0;
}

int any(array double &a) {
    return (int)sum(a == 0) > 0;
}

int count(array double &a) {
    return (int)sum(a == 0);
}
int main () {
    printf("%freturn value of all() is %d\n\n", b, all(b));
    printf("%dreturn value of all() is %d\n\n", c, all(c));
    printf("%dreturn value of any() is %d\n\n", a, any(a));
    printf("%freturn value of any() is %d\n\n", b, any(b));
    printf("%dreturn value of count() is %d\n\n", a, count(a));
    printf("%freturn value of count() is %d\n", b, count(b));
    return 0;
}
```

プログラム 16.16: 配列を上位変換した関数

関数 `all()` の場合、引数配列のすべての要素がゼロなら 1 を返します。それ以外の場合は、0 を返します。関数 `all()` の引数配列 `a` のすべての要素がゼロの場合、配列式 `a!=0` の結果、配列の要素はすべてゼロの値になります。関数 `sum()` からこの配列のすべての要素の合計はゼロになります。関数 `any()` の場合、引数配列のいずれかの要素がゼロなら、1 を返します。それ以外の場合は、0 を返します。関数 `count()` は、配列内のゼロの数を計算します。プログラム 16.16の出力結果を図 16.13に示します。

```
1.000000 2.000000 0.000000
4.000000 5.000000 0.000000
return value of all() is 0

0 0 0
0 0 0
return value of all() is 1

1 2 3
4 5 6
return value of any() is 0

1.000000 2.000000 0.000000
4.000000 5.000000 0.000000
return value of any() is 1

1 2 3
4 5 6
return value of count() is 0

1.000000 2.000000 0.000000
4.000000 5.000000 0.000000
return value of count() is 2
```

図 16.13: プログラム 16.16出力結果

16.12 計算配列へのポインタ

16.12.1 固定長の計算配列へのポインタ

アプリケーションによっては、多次元配列を使用するよりも計算配列へのポインタを使用する方が簡単です。計算配列へのポインタに同じ変数を使用して、さまざまな計算配列にアクセスすることができます。以下に示すように、計算配列へのポインタは、C 配列へのポインタと同じように宣言できます。

```
array double (*p)[10];
```

ここで、p は 10 個の列に double データ型を含む 2 次元計算配列へのポインタとして宣言されています。次のコードは、計算配列へのポインタを使用して多次元の計算配列を扱う方法を示しています。

```
> array int (*p)[3], b1[2][3] ={1, 2, 3, 4, 5, 6}
> int b2[3][3] = {7, 8, 9, 1, 2, 3, 4, 5, 6}, *t
> p == NULL
1
> p = (array int [:][:])(array int [2][3])malloc(2*3*sizeof(int))
0 0 0
0 0 0
> p == NULL
```

```
0
> p[1][1] = 40
40
> p
0 0 0
0 40 0
> p = b1                   // array assignment
1 2 3
4 5 6
> delete p                  // free memory
> p
NULL

> p = (array int [:][:])b1 // p and b1 share the same memory
1 2 3
4 5 6
> b1[0][1] = 30;
30
> b1
1 30 3
4 5 6
> p
1 30 3
4 5 6

> p = (array int [:][:])b2 // p and b2 share the same memory
7 8 9
1 2 3
4 5 6

> t = &b1[0][0]
> p = (array int [:][:])(array int [2][3])t
1 2 3
4 5 6
```

計算配列へのポインタを使用する場合は、先にメモリを割り当てておく必要があります。そうしないと、エラーメッセージが表示されます。一度メモリを割り当てると、ポインタは通常の計算配列として扱われます。計算配列へのポインタは、配列演算のオペランドや関数の引数として使用できます。キャスト演算子の `(array` *`data_type`* `[:]...[:])` または `(array` *`data_type`* `(*)[:]...[:])` を使用し (*data_type* は、計算配列のいずれかの有効なデータ型)、次の 2 つの方法で、計算配列へのポインタにメモリを割り当てることができます。

1. キャスト演算子 `(array data_type [:]...[:])(array data_type [n1]...[ni])`

の直後にメモリへのポインタを指定する。または、

```
(array data_type [:]...[:])new data_type [n1]...[ni];
```

のように演算子 **new** をキャストする。

2. キャスト演算子 `(array data_type [:]...[:])` の直後に C 配列を指定する。

最初の方法で関数 **malloc()**、**calloc()**、または **realloc()** を使用してメモリを割り当てた場合、ポインタへの計算配列に割り当てたメモリは、後で関数 **free()** または演算子 **delete** で解放することができます。演算子 **new** でメモリを割り当てた場合は、演算子 **delete** でメモリを解放する必要があります。前の方法の例として、たとえば次のようなステートメント

```
array int (*p)[3], b1[2][3] ={1, 2, 3, 4, 5, 6}
p = (array int [:][:])(array int [2][3])malloc(2*3*sizeof(int))
```

では、関数 **malloc()** を使用して `p` にメモリを割り当てます。また、メモリを演算子 **new** で割り当て、演算子 **delete** で解放する場合は、次のように指定します。

```
p = (array int [:][:])new int [2][3];
...
delete p;
```

演算子 **new** および **delete** の詳細については、第 19章「ステートメント」を参照してください。

```
t = &b1[0][0]
p = (array int [:][:])(array int [2][3])t
```

または

```
p = (array int [:][:])(array int [2][3])&b1[0][0]
```

または、

```
p = (array int [:][:])(int [2][3])&b1[0][0]
```

で、計算配列 `p` に配列 `b1` のメモリを共有させることができます。

2 番目の方法では、計算配列へのポインタと元の配列との間で同じメモリを共有します。前の方法の例としては、たとえば次のようなステートメント

```
p = (array int [:][:])b1
```

で `p` が計算配列 `b1` を指すようにします。後で、次のようなステートメント

```
p = (array int [:][:])b2
```

で C 配列 b2 の p をポイントします。配列 p、b1、および b2 の 2 番目の次元のエクステントを同じにする必要があることに注意してください。

メモリを p に割り当てれば、代入演算が可能になります。たとえば、次のようなステートメント

```
p = (array int [2][3])b1
```

で配列 b1 の各要素の値を配列 b の対応する要素に代入します。配列 p と b1 のメモリを同じにしないことも可能です。

以下のコマンドで示すように、配列インデックスを付けずに、1 次元計算配列へのポインタを宣言します。

```
> array int *p, a[3] = {1, 2, 3}
> p = (array int [:])a   // p and a share the memory
1 2 3
> p = (array int [:])(array int [4])malloc(4*sizeof(int))
0 0 0 0
>p = (array int [4])10
10 10 10 10
> delete (p)
```

次のコードの対話的な実行は、1 次元計算配列へのポインタを使用して、2 次元配列の行にアクセスする方法を示します。

```
> array int *p, b[2][3] = {1, 2, 3, 4, 5, 6}
> p = (array int [:])(int[3])&b[0][0]
1 2 3
> &p                    // same as &b and &b[0][0]
1e8650
> p = (array int [:])(int[3])(int*)b
1 2 3
> &p                    // same as &a and &a[0][0][0]
1e8650
> p = 10
10 10 10
> b
10 10 10
4 5 6
> p = (array int [:])(int[3])&b[1][0]
4 5 6
> &p                    // same as &b[1][0]
1e865c
```

セクション 16.5.8で説明しているように、ポインタ型とキャスト演算子を使用して配列の先頭の要素のアドレスを取得します。つまり、次のステートメント

```
p = (array int [:])(int[3])(int*)b
```

は次のステートメントと同等です。

```
p = (array int [:])(int[3])&b[0][0]
```

上のコマンドは、p が 2 次元配列 b の先頭の行を参照しますが、下のコマンド

```
p = (array int [:])(int [3])&b[1][0]
```

は、p が配列 b の 2 番目の行を指すようになります。
次の例に示すように、計算配列へのポインタを誤って使用しないように注意してください。以下のような宣言

```
array short h[2][3];
array int (*p)[3], a[2][3], b[3][2], c[6];
array float f[2][3];
array double d[2][3];
int e[2][3], g[3][2];
int *ptr;
```

の場合、次のステートメントは正しくありません。

```
p = a;
```

なぜなら、p にメモリがまだ割り当てられていないからです。計算配列へのポインタを使用するには、その前にメモリを割り当てておく必要があります。次のステートメントも正しくありません。

```
p = (array int [:][:])(a+a);  // bad
```

なぜなら、p は、a ではなく中間的なメモリを参照するからです。同様に、以下のステートメントも正しくありません。

```
p = (array int [:][:])(array int [2][3])b;  // bad
```

p は配列 b ではなく、何らかの中間メモリを参照します。次のステートメントでは、p は配列 b のメモリを参照します。

```
p = (array int [:][:])(array int [2][3])&b[0][0];  // ok
```

また、次に示すように、計算配列へのポインタから通常の C 配列を参照することができます。

```
p = (array int [:][:])e;  // ok
p = (array int [:][:])(array int [2][3])g;  // ok
```

次のコードは正しくありません。

```
p = (array int [:][3])a;  // bad
```

pをaとメモリを共有させるには、キャスト演算子(array int [:][:]) or (array int (*)[:])を使用する必要があります。次のステートメント

```
p = (int (*)[3])a;        // bad
```

では、キーワード**array**が欠けています。スカラ型へのポインタと計算配列へのポインタは互換性がありません。互換性のないlvalueとrvalueでは、以下の代入演算は許可されません。

```
p = ptr;                  // bad
ptr = p;                  // bad
p = (void *)malloc(100);  // bad
```

次のステートメント

```
p = (array int [:][:]) c; // bad
```

では、pとcの次元が一致していません。このため、エラーメッセージが表示されます。次のステートメント

```
p = (array int [:][:])h;  // bad
```

では、pがint型であるため、short型の計算配列hには、pと共有するメモリが足りません。次のステートメント

```
p = (array int [:][:])f;
```

は、fがfloat型で、pと共有できるメモリが十分あるため、メモリの空き容量の観点から正しいと言えます。

ポインタにメモリを割り当てた後、または配列とメモリを共有した後に、アドレス演算子はメモリのアドレス、または配列の先頭の要素のアドレスを取得します。以下の例のコマンド

```
> p = (array int [:][:])a
> &p
```

は、aの先頭の要素のアドレスを取得します。

以下に示すように、計算配列へのポインタを使用して、多次元配列の副配列または"スライス"を得ることができます。

```
> array int a[2][2][2] = {1, 2, 3, 4, 5, 6, 7, 8}
> array int (*p)[2]
4005e3e0
> p = (array int [:][:])(int[2][2])&a[0][0][0]
1 2
3 4
```

```
> &p                    // same as &a and &a[0][0][0]
4005e4e0
> p = (array int [:][:])(int[2][2])(int*)a
1 2
3 4
> &p                    // same as &a and &a[0][0][0]
4005e4e0
> p = (array int [:][:])(int[2][2])&a[1][0][0]
5 6
7 8
> &p                    // same as &a[1][0][0]
4005e4f0
```

1 次元配列と同様に、ポインタ型とのキャスト演算子によって配列の先頭の要素のアドレスを取得できます。つまり、次のステートメント

```
p = (array int [:][:])(int[2][2])(int*)a
```

は次のステートメントと同等です。

```
p = (array int [:][:])(int[2][2])&a[0][0][0]
```

上のコマンドを使用して、p を 3 次元配列 a の一部を参照させることができますが、下のコマンド

```
p = (array int [:][:])(int [2][2])&a[1][0][0]
```

では、p は別の部分を参照します。

16.12.2 形状引継ぎ計算配列へのポインタ

また、計算配列へのポインタ以外に、Ch では、形状引継ぎ計算配列へのポインタをサポートします。計算配列へのポインタとは異なり、形状引継ぎ計算配列へのポインタを使用して、可変長の配列を参照できます。したがって、ユーザーは、参照される配列のエクステントを考慮する必要はありません。形状引継ぎ計算配列へのポインタは、配列の添字にコロン':' を使用して宣言します。形状引継ぎ計算配列へのポインタを使用するには、前のセクションで説明されている固定長の計算配列へのポインタの場合と同じように、事前にメモリを割り当てておく必要があります。

たとえば、次のステートメント

```
array float (*fp)[:];
```

のように、形状引継ぎ計算配列へのポインタとして fp を float 型で宣言します。これで、可変長の 2 次元配列をポイントすることができます。declares

次のコマンドは、形状引継ぎ計算配列にポインタを使用して、長さの異なる多次元配列を扱う方法を示します。

```
> array int (*p)[:]
> array int b1[2][3] ={1, 2, 3, 4, 5, 6}
> array int b2[2][2] ={5, 6, 7, 8}
> p = (array int [:][:])b1
1 2 3
4 5 6
> p = (array int [:][:])b2
5 6
7 8
```

上記のコマンドでは、配列の b1 と b2 とは同じ次元です。b1 と b2 の 2 番目の次元のエクステントは異なります。固定長の計算配列へのポインタと異なり、p(形状引継ぎ計算配列へのポインタ) を使用して、b1 か b2 のどちらかを参照するようにできます。

また、計算配列へのポインタを使用して、多次元配列のサブスペース (副配列) を表すことができます。次に例を示します。

```
> array int a[2][2][3] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12}
> array int b[3][2][2] = {12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1}
> array int (*p)[:]
> a
1 2 3
4 5 6

7 8 9
10 11 12
> p = (array int [:][:])(int [2][3])&a[0][0][0]
1 2 3
4 5 6
> p = (array int [:][:])(int [2][3])&a[1][0][0]
7 8 9
10 11 12
> p = (array int [:][:])(int [2][2])&b[0][0][0]
12 11
10 9
```

ここで、p は、配列 a の先頭と 2 番目の”スライス”を表し、次元が異なる配列 b の先頭の”スライス”を表すために使用されています。アドレス演算子 ‘&’ で、これらの配列の要素のアドレスを取得します。

通常の識別子だけでなく、クラス、構造体、共用体のメンバも形状引継ぎ計算配列へのポインタとして宣言できます。これは、クラス、構造体、共用体のメンバも、可変長の計算配列にできることを意味します。以下の対話的なコマンドを実行すると、メンバ s.a は、配列 a1 とメモリを共有してから配列 a2 とメモリを共有します。

```
> struct tag{ array int (*a)[:];} s
```

```
> array int a1[2][3] = {1, 2, 3, 4, 5, 6}, a2[3][4]
> s.a = (array int [:][:])a1;  // s.a and a1 share the memory
> s.a
1 2 3
4 5 6
> s.a = (array int [:][:])a2;  // s.a and a2 share the memory
s.a[1][1] = 10
> a2[1][1]
10
> a1[1][1]
5
```

16.12.3 計算配列へのポインタを使用して配列を関数に渡す方法

1 次元計算配列へのポインタを使用して、可変長の配列を扱うことができます。また、形状引継ぎ配列の場合と同様に、計算配列へのポインタを関数の引数として使用することもできます。次に例を示します。

```
> void func1(array int *p) {printf("%d", p);}
> int array a[2] = {1,2}
> int array b[3] = {3,4,5}
> func1(a)
1 2
> func1(b)
3 4 5
```

関数 `func1()` は計算配列へのポインタの引数を受け取ります。この関数は、長さの異なる `a` や `b` などの配列を扱うことができます。上記の例の関数定義は、形状引継ぎ配列の引数を受け取る次の関数定義と同等です。

```
void func1(array int p[:]) { printf("%d", p); }
```

また、多次元形状引継ぎ配列の場合と同様に、関数の引数として形状引継ぎ配列へのポインタを使用できます。次に例を示します。

```
> void func2(array int (*p2)[:]) { printf("%d", p2); }
> int array a2[2][2] = {1,2,3,4}
> int array b2[3][3] = {1,2,3,4,5,6,7,8,9}
> func2(a2)
1 2
3 4
> func2(b2)
1 2 3
4 5 6
7 8 9
```

関数 func2() は、2 次元の形状引継ぎ配列へのポインタの引数を受け取ります。これにより、a2 や b2 などのエクステントの異なる 2 次元計算配列を扱うことができます。上記の例の関数 func2() の定義は、以下の定義に同等で、2 次元の形状引継ぎ配列の引数を受け取ります。

```
void func2(array int p2[:][:]) { printf("%d", p2); }
```

16.13 計算配列と C 配列との関係

C 配列はアドレスやポインタにすぎませんが、Ch の計算配列は詳細な情報を含むファーストクラスオブジェクトです。これまで説明したように、計算配列は型修飾子 **array** で宣言します。計算配列は、C 配列がサポートできない多くの演算子をサポートできます。同じエクステントであれば、C 配列の各要素の値を計算配列に代入することができます。次に例を示します。

```
int a[3][4];   // C array, 'a' represents an address or a pointer
int b[4][3];   // C array, 'b' represents an address or a pointer
array int A[3][4]; // Ch computational array
array int (*p)[4]; // point to computational array
array int (*p2)[:];// point to computational array of assumed shape
A = (array int[3][4])a; // OK
p = (array int[:][:])a; // OK
p = (array int[:][:])(array int [3][4])b; // OK
p = (array int[:][:])(int [3][4])b;       // OK
p2 = (array int[:][:])a;// OK
p2 = (array int[:][:])(array int [3][4])b;// OK
p2 = (array int[:][:])(int [3][4])b;      // OK
A = a;                  // OK
a = A;                  // Error
```

上記の例では、次元 4 × 3 を含む配列 b のメモリへは、次元 3 × 4 の配列として計算配列 p と p2 へのポインタでアクセスします。p2 が形状引継ぎ計算配列へのポインタであるため、2 番目の次元に異なるエクステントを持つ配列を参照することができます。たとえば、次のように、p2 は次元 4 × 3 の配列 b を参照することもできます。

```
p2 = (array int[:][:])b;      // OK
```

計算配列を引数として受け取る関数へ C 配列を渡すことができます。この逆も可能です。次に例を示します。

```
int f1(array int A[3][4]); // argument is computational array
int f2(int a[3][4]);       // argument is C array
f1(a); // OK
f1(A); // OK
f2(a); // OK
f2(A); // OK
```

C 配列の変数を配列のメモリのアドレスとして使用した場合は、計算配列のアドレスも同等のコードに使用する必要があります。たとえば、次のようにします。

```
int f3(int *a);              // argument is a pointer
f3(a);         // OK
f3(&A[0][0]); // OK
```

第17章　文字と文字列

Ch では、以下に示すように、**char** 型と **wchar_t** 型を使用して文字変数 とワイド文字変数を定義します。

```
char ch = 'a';
char ch2 = '\0';  /* null character */
wchar_t wch = L'a';
```

char 型の変数の値は、`'x'` のように一重引用符で囲まれた単一の文字またはエスケープシーケンスです。C では、文字定数の型は int です。C++と同じように、Ch での文字定数の型は **char** です。

ワイド文字の値は、`L'x'` のように文字 L が先行する以外は、文字と同じです。ワイド文字の型は **wchar_t** であり、これは stddef.h または stdlib.h ヘッダーファイルに定義された整数型です。拡張された実行文字セットのメンバに対応する単一のマルチバイト文字を格納するワイド文字の値は、そのマルチバイト文字に対応するワイド文字 (コード) であり、**mbtowc** 関数によって、プラットフォームに依存する現在のロケールを使用して定義されます。

`"xyz"`では、文字列は、"xyz"のように二重引用符で囲まれたマルチバイト文字のシーケンスです。C コンパイラと同様に、Ch では、すべての文字列の後ろに null 文字が自動的に付加されます。ワイド文字列リテラルは、文字 L が先行する以外は文字列と同じです。

Ch と C では、文字 (またはワイド文字) 配列を使用して文字列 (またはワイド文字列) 変数を定義します。次に例を示します。

```
char *str = "this is a string.";
char str2[] = "this is also a string.";
char str3[6] = "abcde"; /* the last one is '\0' */
wchar_t *wstr = L"this is a wide string.";
```

17.1　string.h ヘッダーファイル内の関数の使用

ヘッダー **string.h** は、文字型の配列および文字型の配列として扱われるその他のオブジェクトを処理するために役立つ、1 つの **size_t** 型と複数の関数を宣言します。配列の長さは、さまざまな方法で決定できますが、どの場合でも、char *または void *引数は、配列の最初 (最下位アドレス) の文字を指します。配列がオブジェクトの終わりを越えてアクセスされた場合は、最後の要素の値が使用されます。このセクションでは、**string.h** ヘッダーファイルで宣言されて一般的に使用される関数を分類し、説明します。

17.1.1 コピー関数

関数名	説明
memcpy()	オブジェクトの文字を別のオブジェクトにコピーします。
memmove()	オブジェクトの文字を別のオブジェクトに移動します。
strcpy()	文字列を別の文字列にコピーします。
strncpy()	文字列の指定された個数の文字を別の文字列にコピーします。

たとえば、次のコードがあるとします。

```
> char str1[80] = "abcdefghijk"
> strcpy(str1, "efghij")
efghij
> str1
efghij
> strncpy(str1, "klmnopqrs", 3)
klmhij
> str1
klmhij
> strncpy(str1, "tuv", 5)
tuv
> str1
tuv
>
```

以下の関数呼び出し

```
strcpy(str1, "efghij")
```

は、文字列 "`efghij`"を (終端の null 文字を含めて)、`str1` がポイントする配列にコピーします。この関数は、記憶域を割り当てません。`str1` がポイントするバッファが、文字列 s2 とその終端の null 文字を保持するのに十分な長さであることは、呼び出し元が保証する必要があります。同様に、以下の関数呼び出し

```
strncpy(str1, "klmnopqrs", 3)
```

は、文字列 "`klmnopqrs`"から最大 3 文字を `str1` がポイントするバッファにコピーします。**strncpy()** が 3 文字を `str1` にコピーした後、終端の null 文字は追加されません。従って、結果は "`klm`"ではなく、"`klmhij`"になります。以下の関数呼び出し

```
strncpy(str1, "tuv", 5)
```

は、終端の null 文字を含む最大 5 文字を、文字列 "`tuv`"から、`str1` がポイントするバッファにコピーします。文字列 "`tuv`"の長さが 5 未満なので、終端の null が追加されます。関数 **strncpy()** も、記憶域を割り当てません。`str1` がポイントするバッファが、そのバッファにコピーされる文字を保持するのに十分な長さであることは、呼び出し元が保証する必要があります。

17.1.2　連結関数

関数名	説明
strcat()	文字列のコピーを別の文字列の末尾に追加します。
strncat()	文字列の指定された個数の文字を別の文字列の末尾に追加します。

たとえば、次のコードがあるとします。

```
> char str1[80] = "abcd"
> strcat(str1, "efg")
abcdefg
> str1
abcdefg
>
```

strcat() 関数は、文字列 “`efg`”のコピーを (終端の null 文字を含めて)、`str1` がポイントする文字列の末尾に追加します。2 番目の引数の最初の文字 ‘`e`’ は、str1 の末尾にある null 文字を上書きします。この関数は、記憶域を割り当てません。`str1` がポイントするバッファが、2 番目の文字列とその終端の null 文字を追加するのに十分な長さであることは、呼び出し元が保証する必要があります。

17.1.3　比較関数

関数名	説明
memcmp()	オブジェクトの n 個の文字を別のオブジェクトの文字と比較します。
strcmp()	2 つの文字列を比較します。
strcoll()	文字列を別の文字列と比較します。
strncmp()	文字列の指定された個数の文字を別の文字列と比較します。
strxfrm()	文字列を別の文字列に変換します。

たとえば、次のコードがあるとします。

```
> char str1[80] = "abcd"
> strcmp(str1, "aacd")
1
> strcmp(str1, "abcd")
0
> strcmp(str1, "efg")
-1
>
```

strcmp() 関数は、`str1` がポイントする文字列を、文字列 “`aacd`”、“`abcd`”、および “`efg`”とそれぞれ比較します。文字列 str1 がレキシカルレベルで文字列 “`aacd`”よりも上位の場合は 1 を、文字列 str1 と “`abcd`”が同一の場合はゼロを、文字列 str1 が “`efg`”よりもレキシカルレベルが下位の場合は-1 を

返します。

17.1.4 検索関数

関数名	説明
memchr()	オブジェクト内での文字の最初の出現箇所を検索します。
strchr()	文字列内での文字の最初の出現箇所を検索します。
strcspn()	文字列の最大の初期セグメントの長さを計算します。
strpbrk()	文字列を別の文字列内で検索します。
strrchr()	文字列内での文字の最後の出現箇所を検索します。
strspn()	文字列の最大の初期セグメントの長さを計算します。
strstr()	文字列の最初の出現箇所を別の文字列内で検索します。
strtok()	文字列をトークンのシーケンスに分割します。

たとえば、次のコードがあるとします。

```
> char str1[80] = "abcdefgdef"
> strchr(str1, ’d’)
defgdef
> strchr(str1, ’w’)
00000000
> strstr(str1, "def")
defgdef
> strstr(str1, "dev")
00000000
> strtok(str1, "efg")
> char *str2 = "abcd;1234 ABCD"
> char *delimiter=" ;", *token
> token = strtok(str2, delimiter)
abcd
> token = strtok(NULL, delimiter)
1234
> token = strtok(NULL, delimiter)
ABCD
> token = strtok(NULL, delimiter)
(null)
>
```

以下の関数呼び出し

```
strchr(str1, ’d’)
```

は、str1 がポイントする文字列内で、文字 ‘d’ の最初の出現箇所を検索し、その場所のポインタを返します。文字列 str1 には文字 ‘w’ は出現しないので、以下の関数呼び出し

```
strchr(str1, 'w')
```

は、null ポインタを返します。同様に、以下の関数呼び出し

```
strstr(str1, "def")
```

は、文字列 str1 内で、終端の null 文字を除外したサブ文字列 “def”の最初の出現箇所を検索し、そのサブ文字列へのポインタを返します。文字列 str1 にはサブ文字列 “dev”がないので、以下の関数呼び出し

```
strstr(str1, "dev")
```

は null ポインタを返します。**strtok()** 関数は、文字列から次のトークンを取得します。トークンは、2 番目の引数で指定された文字によって分離された文字列です。文字列 str2 から最初のトークンを取得するには、以下の関数

```
token = strtok(str2, delimiter)
```

で str2 を最初のパラメータとして使用します。以下の関数呼び出し

```
token = strtok(NULL, delimiter)
```

は、最初のパラメータで null ポインタを使用し、それ以外のすべてのトークンを str1 から次々に返します。2 番目の引数は、区切り文字列であり、呼び出しごとに変更できます。セクション 17.3で、文字列からトークンを取得する **foreach** ループについて説明します。

17.1.5 その他の関数

関数名	説明
memset()	オブジェクトの最初に指定された個数の文字のそれぞれに値をコピーします。
strerror()	errnum 内の数値をメッセージ文字列にマップします。
strlen()	文字列の長さを計算します。

次に例を示します。

```
> strlen("abcde")
5
>
```

strlen() 関数は、文字列 “abcde”の長さを返します。**strlen()** 関数では、文字列の終端の null はカウントされないので、この例では結果として 6 ではなく 5 が返されます。動的に割り当てられた文字列のメモリサイズをこの関数を使用して計算する場合は、返された値に 1 を足す必要があります。

17.1.6 C 標準ライブラリでサポートされていない Ch の文字列関数

関数名	説明
strcasecmp()	2 つの文字列を、大文字と小文字を区別しないで比較します。
strconcat()	文字列を連結します。
strjoin()	文字列を、指定されたデリミタ文字列によって分離された文字列と結合します。
strncasecmp()	2 つの文字列の一部を、大文字と小文字を区別しないで比較します。

たとえば、次のコードがあるとします。

```
> char *buffer
> char test[90] = "abcd"
> buffer = strconcat(test, "efgh", "ijk")
abcdefghijk
> free(buffer)
> buffer = strjoin("+", test, "efgh", "ijk")
abcd+efgh+ijk
> free(buffer)
>
```

文字配列 `test` の値が文字列 `abcd` であると仮定すると、以下の関数呼び出し

```
buffer = strconcat(test, "efgh", "ijk")
```

は、これらの 3 つの文字列を連結し、動的にメモリが割り当てられる戻り値の文字列に結果を格納します。動的に割り当てられたメモリは、後でユーザーが解放する必要があります。以下の関数呼び出し

```
buffer = strjoin("+", test, "efgh", "ijk")
```

も、3 つの文字列を結合して、動的にメモリが割り当てられる戻り値の文字列に格納します。ただし、戻り値の文字列は、関数 **strjoin()** の最初の引数で指定した区切り文字列 “+”によって分離されます。

17.2 文字列型 string_t

C には文字列のデータ型はありません。前述したように、文字の配列は、C では文字列として処理されます。Ch には、文字列のデータ型 **string_t** が追加されています。それは、char へのポインタにシームレスに統合されます。標準 C ライブラリヘッダ string.h に定義されたすべての関数は、char へのポインタと文字列へのポインタの両方で有効です。Ch では、文字列型 **string_t** の文字列はファーストクラスオブジェクトです。たとえば、以下のコード例

```
string_t s, a[3];
s = "great string"
s = stradd("greater ", s)
strcpy(a[0], s);
printf("a[0] = %s\n", a[0]);
```

表 17.1: **string_t** 型用の関数

関数名	説明
str2ascii()	文字列の ASCII 値を取得します。
str2mat()	文字列を行列に変更します。
stradd()	2 番目の文字列を最初の文字列に追加します。
strgetc()	文字列から文字を取得します。
strputc()	文字列に文字を配置します。
strrep()	文字列内の任意の文字列を別の文字列に置き換えます。

は、`greater great string` を表示します。**string_t** は Ch のキーワードであり、stradd() 関数は組み込みの汎用関数です。書式指定子`"%s"`を使用して、以下のコマンドに示すように、文字列型変数への入力を取得できます。

```
> string_t s
> scanf("%s", &s)
123abc
> printf("%s", s)
123abc
>
```

文字列関数 **strcpy()**、**strncpy()**、**strcat()**、および **strncat()** では、最初の引数が **string_t** 型であれば、メモリは自動的に処理されます。次に例を示します。

```
> string_t s
> strcpy(s, "abcd")
abcd
> strcat(s, "ABCD")
abcdABCD
> s
abcdABCD
>
```

Ch では、ヘッダーファイル **string.h** に、表 17.1に示す **string_t** 型専用の関数が追加で宣言されています。主な追加関数として、**str2ascii()**、**str2mat()**、**strgetc()**、**strputc()**、および **strrep()** があります。

たとえば、次のコードがあるとします。

```
> str2ascii("a")
97
> str2ascii("b")
98
> str2ascii("ab")
```

```
195
> array char mat[3][10]
> str2mat(mat, "abcd", "0123456789")
0
> mat
a b c d
0 1 2 3 4 5 6 7 8 9
> str2mat(mat, "ABCD", "EFGH", "ab23456789", "too many strings")
-1
> mat
A B C D
E F G H
a b 2 3 4 5 6 7 8 9
> string_t s1 = "abcd"
> stradd(s1, "efg")          // add "efg" to s1
abcdefg
> strgetc(s1, 0)             // get the first character of s1
a
> strgetc(s1, 2)             // get the third character of s1
c
> strputc(s1, 2, 'z')        // change the third character to 'z'
0
>
s1
> abzdefg
> strrep(s1, "def", "xyz")  // replace "def" with "xyz" in s1
abzxyzg
>
```

以下の関数呼び出し

```
str2ascii("ab")
```

は、文字 “a”と “b”の ASCII 値を加算することで、文字列 “ab”の ASCII 値を計算します。以下の関数呼び出し

```
str2mat(mat, "abcd", "0123456789")
```

は、2 つの文字列 “abcd”と “0123456789”を、配列 mat の先頭の 2 行に割り当て、正常終了時に 0 を返します。mat の残りの行は null のままです。以下に示すように、引数リストの文字列が、配列 mat 内の行よりも多い場合、

```
str2mat(mat, "ABCD", "EFGH", "ab23456789", "too many strings")
```

関数は-1 を返し、文字列 “too many strings”を無視します。**stradd()** 関数は、ある文字列を別の文字列に追加するための汎用関数です。メモリは、Ch がユーザーに代わって処理します。値 “abcd”を持つ文字列 s1 の型が **string_t** と仮定すると、以下の関数

```
stradd(s1, "efg")
```

は、文字列 “efg”を s1 の末尾に追加した後、s1 を返します。以下の関数呼び出し

```
strgetc(s1, 0)
strgetc(s1, 2)
```

は、1 番目の文字と 3 番目の文字、つまり ‘a’ と ‘C’ を返します。関数 **strgetc()** と **strputc()** は、文字列の中の文字を操作するときに特に役立ちます。以下の関数呼び出し

```
strputc(s1, 2, 'z')
```

は、文字列 s1 の 3 番目の文字を ‘z’ に変更します。以下の関数呼び出し

```
strrep(s1, "def", "xyz")
```

は、s1 の文字列 “def”を文字列 “xyz”に置き換えて、s1 を返します。

前述したように、**string_t** 型の利点の 1 つは、Ch では、**string_t** 型の変数用のメモリを自動的に処理できることです。Ch では、このような変数に対するすべての演算で、必要なメモリのサイズが計算され、変数に対して十分なメモリが割り当てられます。Ch では、このような変数の存続期間の終わりに、割り当てられたメモリが自動的に解放されます。たとえば、以下のプログラムでは、関数 fun() の変数 s1 のメモリは、その関数の終了時に解放され、変数 s2 のメモリは、関数が戻るとき、または **main()** 関数内で変数 s が代入されるときに解放されます。

一方、s のメモリは、s の代入時に自動的に割り当てられます。

```
string_t fun() {
  string_t s1;
  string_t s2;
  ...
  return s2;
}
int main() {
  string_t s;
  fun();
  s = fun();
  ...
}
```

17.3　foreach ループを使用した文字列トークンの処理

while ループ、**do-while** ループおよび **for** ループに加え、セクション 8.4.4に示した **foreach** ループは、文字列を処理するときに特に便利です。foreach ループでは、テキストに対して実行される置換を毎回変更して、1 つのテキストを繰り返し使用します。これにより、文字列の処理や配列の反復を簡単に実行できます。

たとえば、関数 **strtok()** または **strtok_r()** を使用して、 null で終わる文字列のトークンを取得できます。以下のコード例があります。

```
char *s = "abcd;1234 ABCD;56;xyz";
char *delimiter=" ;", *token;
token = strtok(s, delimiter);
while(token) {
    printf("token = %s\n", token);
    token = strtok(NULL, delimiter);
}
```

次の出力が生成されます。

```
abcd
1234
ABCD
56
xyz
```

この例は、**foreach** ループを使用して以下のように書き換えることができます。

```
char *s = "abcd;1234 ABCD;56;xyz";
char *delimiter=" ;", *token;
foreach(token; s; NULL; delimiter)
    printf("token = %s\n", token);
```

上記のコードで、**foreach** ループの `cond` 値として **NULL** を文字列”ABCD”に置き換えた場合、コード例は次のようになります。

```
char *s = "abcd;1234 ABCD;56;xyz";
char *delimiter=" ;", *token;
foreach(token; s; "ABCD"; delimiter)
    printf("token = %s\n", token);
```

上記のコードの出力は次のとおりです。

```
abcd
1234
```

17.4 ワイド文字

ワイド文字定数の型は **wchar_t** であり、これは stddef.h ヘッダーに定義される整数型です。ワイド文字は、'x' や'ab' のように一重引用符で囲まれ、文字 L が先行する 1 つ以上のマルチバイト文字のシーケンスです。拡張実行文字セットのメンバに対応する単一のマルチバイト文字を格納するワイド文字定数の値は、そのマルチバイト文字に対応するワイド文字 (コード) であり、**mbtowc** 関数によって、プラットフォームに依存する現在のロケールを使用して定義されます。複数のマルチバイト文字を格納するワイド文字定数の値、または拡張実行文字セットで表現されない 1 つのマルチバイト文字またはエスケープシーケンスを格納するワイド文字定数の値は、プラットフォームに依存します。

たとえば、ワイド文字変数 wc の定義は以下のようになります。

```
wchar_t wc = L'a';
```

Ch コマンドシェルで、簡体字中国語、ロシア語、日本語などのマルチバイト言語のためにワイド文字とワイド文字列を効果的に使用するには、ヘッダーファイル **wchar.h** および **wctype.h** にある関数を使用し、特定の Unicode に対しては、以下のステートメント

```
#include <locale.h>
#pragma exec setlocale(LC_ALL, "Chinese-Simplified");
```

を、またはシステムの既定の Unicode に対しては、以下のステートメント

```
#include <locale.h>
#pragma exec setlocale(LC_ALL, "");
```

をプログラムの先頭に追加します。あるいは、特定の Unicode に対しては、以下のステートメント

```
setlocale(0, "Chinese-Simplified");
```

を、システムの既定の Unicode に対しては、

```
_setlocale = 1;
```

または

```
setlocale(LC_ALL, "");
```

をユーザーのホームディレクトリにある個別のユーザースタートアップファイル **_chrc** またはシステムのスタートアップファイル **CHHOME/config/chrc** に追加し、このセットアップがすべてのプログラムで有効になるようにします。

17.5 ワイド文字列

ワイド文字の文字列定数は、二重引用符で囲まれ、文字 L が先行するゼロ個以上のマルチバイト文字のシーケンスです。以下のコード

```
wchar_t *wstr = L"abcd";
```

は、Ch におけるワイド文字の文字列 wstr を定義します。ファイル **stdlib.h** に宣言された関数 **mbstowcs()** は、マルチバイト文字列をワイド文字の文字列に変換でき、**mbstowcs()** 関数は逆の操作を行います。ヘッダーファイル **wchar.h** は、ワイド文字用のデータ型、タグ、マクロ、および関数を宣言します。

第18章 構造体、共用体、ビットフィールド、および列挙体

18.1 構造体

Ch の構造体型は C++の構造体型と似ています。構造体型は、異なる型を格納できるメンバの集合です。たとえば、Ch の複素型は、次の構造体の定義と同等です。

```
struct Complex{
      float r;
      float m;
};
```

ここで、2 つのメンバ r と m は、複素数の実部と虚部の格納に使用されます。Complex は、構造体のタグと呼ばれます。次のコードで、Complex 型のオブジェクトを作成できます。

```
Complex z;
z.r = 10;
z.m = 5;
```

選択演算子 “.”(ドット演算子と呼ばれます) は、構造体のメンバへのアクセスに使用します。メンバ r は 10 に設定され、メンバ m は 5 に設定されます。変数が構造体へのポインタとして定義されている場合、そのメンバへのアクセスには、“->”(アロー演算子と呼ばれます) という演算子が使用されます。次に例を示します。

```
Complex *pz = &z;
pz->r = 10;
pz->m = 5;
```

C の構造体には 2 つの名前空間があります。1 つは構造体のタグの名前空間であり、1 つは変数の名前空間です。しかし、C++の構造体には 1 つと半分の名前空間があります。1 つはタグの名前空間であり、半分は変数の名前空間です。Ch では後者と同じ方法で構造体を処理します。タグと変数は同じ名前空間を共有します。変数として明示的に使用すると、そのタグ名は Ch では暗黙的に型指定された名前として処理されません。次に例を示します。

```
struct tag1_t {
    struct tag2_t;
    ....
```

```
};
tag1_t s;           // ok
int tag1_t;         // ok
struct tag1_t s2;   // ok
tag1_t s3;          // error, tag1_t is a variable of int
struct tag2_t s4;   // Not valid in Ch and C++, OK in C
```

18.2 共用体

共用体型は、重なっている空でないメンバオブジェクト集合を記述します。これらのそれぞれのオブジェクトには、任意に指定された名前があり、個別の型を持っている可能性があります。構造体のように、共用体は複数のメンバを持つことができます。構造体とは異なり、共用体は一度にメンバの 1 つのみを格納できます。概念的には、メンバは同一のメモリ内で重なります。共用体の各メンバは共用体の先頭に配置します。たとえば、次の共用体には 3 つのメンバがあります。

```
union U1{
     double d;
     char c[12];
     int i;
} obj, *P = &obj;
```

次に、次の等式が成り立ちます。

```
(union U1*)&(P->d) == (union U1*)(P->c) == (union U1*)&(P->i) == P
```

共用体のインスタンスのサイズは、最大メンバを表すのに必要なメモリの量に、長さを最適な整列境界まで延長するパディングを加えたサイズです。前の例では、次の等式が成り立ちます。

```
sizeof U1 == 16
```

ただし、最大メンバ c はメモリを 12 バイトしか使用しません。共用体は一度に 1 つのメンバのみを格納するため、複数のメンバをキャストせずに使用すると、未知の結果が得られる可能性があります。たとえば、次のコード例があります。

```
obj.i = 10;
printf("obj.d = %f\n", obj.d);
```

この例では、10 ではなくゼロまたは小さな値を出力します。これは、int 変数および float 変数の表現が異なるためです。Ch と C++の共用体には 1 つと半分の名前空間があります。1 つは構造体タグの名前空間であり、半分は変数の名前空間です。C++と同様に、Ch では、`U1` などの共用体タグは既定で型定義された名前空間に配置されます。

18.3 ビットフィールド

C と同様に、Ch では単語内で直接定義およびアクセスする機能を持つビットフィールドを提供します。次のコード例があります。

```
struct Bf1 {
    unsigned int a;
    unsigned int b;
    unsigned int c;
}  bf1 = {1, 1, 1};

struct Bf2 {
    unsigned int a : 4;
    unsigned int b : 4;
    unsigned int c : 4;
}  bf2 = {1, 2, 3};

bf2.c = 4;
printf("sizeof Bf1 is %d\n", sizeof(struct Bf1));
printf("sizeof Bf2 is %d\n", sizeof(struct Bf2));
```

構造体内部に 3 つの整数があるため、`Bf1` のサイズは 12 バイトです。しかし、`Bf2` のサイズは 4 バイトです。これは 3 つのメンバが 12 ビットのメモリを使用するだけでそれにパディングが追加されるためです。次のビットフィールドについて考えます。

```
struct eeh_type {
  uint16 u1: 10;   /* 10 bits */
  uint16 u2:  6;   /* 6 bits */
};
```

これは、実際は次のように実装される可能性があります。

```
<10-bits><6-bits>
```

または、次のように実装される可能性があります。

```
<6-bits><10-bits>
```

このどちらであるかは、マシンとオペレーティングシステムのエンディアンタイプにより異なります。選択演算子 “.”を使用すると、ビットフィールドのメンバにアクセスすることができます。たとえば、次の等式が成り立ちます。

```
bf2.a == 1;
bf2.b == 2;
bf2.c == 4;
```

18.4 列挙体

列挙型は列挙定数で表現される一連の整数値です。たとえば、次の宣言があるとします。

```
enum datatypes {
    inttype,      // 0
    floattype,    // 1
    doubletype,   // 2
} d1, d2;
```

この宣言は、値が inttype、floattype および doubletype である新規の列挙型 enum datatypes を作成します。また、列挙型の 2 つの変数 d1 と d2 を宣言します。これらの変数には、次の代入ステートメントを使用して列挙定数を割り当てることができます。

```
d1 = inttype;
d2 = doubletype;
```

最初の列挙定数は既定で値 0 を受け取ります。後続の列挙定数は前の列挙定数より 1 大きい整数値を受け取ります。d1 と d2 の値は、それぞれ 0 と 2 になります。明示的な整数値を定義内の列挙定数に関連付けることができます。たとえば、次の宣言があるとします。

```
enum datatypes {
    inttype,           // 0
    floattype = 10,    // 10
    doubletype         // 11
};
```

inttype、floattype、doubletype の値はそれぞれ 0、10、および 11 になります。一つの応用例として、**#define** ディレクティブの代わりに列挙型を使用することができます。次のコードは、**switch** ステートメントで列挙型の変数を使用しています。

```
enum datatypes {
    inttype,
    floattype,
    doubletype
};
enum datatype dt1;

...

switch(dt1) {
    case inttype:
        ...
        break;
```

```
        case floattype:
            ...
            break;

        case doubletype:
            ...
            break;
    }
    ...
```

第19章 クラスおよびオブジェクトベースのプログラミング

19.1　クラス定義とオブジェクト

C++および Ch のクラスは構造体が自然に進化したものです。クラスをユーザー定義の型の作成に使用できます。C ではクラスのメンバとして関数を使用できますが、構造体のメンバとして使用することはできません。C++と同様に、Ch ではクラスと構造体の両方がメンバ関数を持つことができます。既定では、クラスのメンバは private ですが、構造体のメンバは public です。クラスの定義例を次に示します。

```
class Student {
    int id;
    char *name;
};
```

クラス `Student` には 2 つのメンバがあります。`id` に学生の ID 番号が格納され、`name` が学生の名前であるとします。クラスを定義した後は、プログラム内で次のように使用できます。

```
int main() {
    class Student s1;
    ....
}
```

ここで、`s1` はクラス `Student` の**オブジェクト**または**インスタンス**と呼ばれます。

19.2　クラスのメンバ関数

前述したように、関数はクラスのメンバとして使用できます。ヘッダーファイル `student.h` で、`Student` クラスを次のように再定義できます。

```
/* Filename: student.h */
#ifndef STUDENT_H
#define STUDENT_H

class Student {
     int id;
```

```
      char *name;
    public:
      void setID(int i);
      void setName(const char *n);
      int getID();
};

#pragma importf <Student.cpp>

#endif
```

メンバ関数は、次に示す別のファイル Student.cpp で定義されます。システム変数_**fpath** で指定されたディレクトリ内にあるこのファイルは、

```
#pragma importf <Student.cpp>
```

ステートメントによって一度読み込まれます。

```
/* Filename: Student.cpp */
#include <string.h> /* for strdup() */
#include "student.h"

void Student::setID(int i) {
    id = i;
}

void Student::setName(const char *n) {
    if(n)
      name = strdup(n);
}

int Student::getID() {
    return id;
}
```

メンバ関数の定義では、関数名の前にクラス名およびスコープ解決演算子 '::' が付けられます。この演算子については後で説明します。関数 setID() は、学生の ID 番号を引数として受け取り、クラスメンバ id にその値を設定します。関数 setName() は、メンバ name に新しい名前を設定します。関数 getID() は学生の ID を取得します。Ch および C++では、メンバ setID()、setName()、および getID() は、**メンバ関数**または**メソッド**と呼ばれます。プログラム prog.cpp に示すように、構造体のメンバにアクセスするのと同じやり方で、メンバ演算子 '.' を使用してメンバ関数を呼び出すことができます。

```
/* Filename: prog.cpp */
```

```
#include <iostream.h>  /* for cout */
#include "student.h"

using namespace std;   /* for cout */
int main() {
    class Student s;
    s.setID(1);
    s.setName("Jason");
    cout << "id is " << s.getID() << endl;
    return 0;
}
```

name や id などのクラスの private メンバは、通常、クラスの外部からはアクセスできません。これは情報隠蔽と呼ばれます。クラスのメンバ関数の主な役割の 1 つは、クラスの private メンバにアクセスする手段を提供することです。

19.3 クラスの public メンバと private メンバ

すでに指摘したように、クラス Student のメンバである id および name にはクラスの外部からはアクセスできません。外部のコードはクラスの一部のメンバ関数を使用してのみ、これらのメンバにアクセスできます。これは、これらのメンバが private メンバであり、クラス Student 内で定義されるメンバ関数のすべてが public メンバであるためです。Ch には 2 つのメンバアクセス指定子 public:および private:があります。 これらの指定子はクラス定義内で、任意の順序で複数指定することができます。既定では、クラスのメンバは private ですが、構造体のメンバは public です。次のように、クラス Student の定義を記述することができます。

```
class Student {
  public:
    void setID(int i);
    void setName(const char *n);
  private:
    int id;
    char *name;
};
```

通常、クラスのデータメンバは private メンバとして定義され、メンバ関数は public メンバとして定義されます。クラスの public メンバ関数のセットは、インタフェースと呼ばれます。ただし、public データメンバまたは private メンバ関数を定義する必要がある場合もあります。public データメンバは、public メンバ関数と同様にメンバ演算子 ‘.’ によってクラス外部からアクセスできます。一方、private メンバ関数はクラスの他のメンバ関数によってのみ呼び出すことができます。

19.4 クラスのコンストラクタとデストラクタ

Ch のクラス定義ではクラスのデータメンバを初期化できません。コンストラクタで初期化を実行することができます。コンストラクタおよびデストラクタは、戻り値を指定させられないメンバ関数です。コンストラクタの名前はクラス名と同じです。コンストラクタはオブジェクトがインスタンス化されるたびに自動的に呼び出され、初期化を実行します。コンストラクタは、データメンバを初期化するために引数を受け取ることができますが、デストラクタは引数を受け取ることができません。たとえば、次のように、クラス `Student` のコンストラクタおよびデストラクタを追加することができます。

```
class Student {
  public:
    Student(int, const char *);  // constructor
    ~Student();            // destructor
    void setID(int i);
    void setName(const char *n);
  private:
    int id;
    char *name;
};

Student::Student(int i, const char *n) {
    id = i;            /* initialize id */
    name = strdup(n);  /* initialize name */
}

Student::~Student() {
    /* release the memory allocated in constructor */
    free(name);
}
```

ここで、コンストラクタは新しいオブジェクトが作成されるときにデータメンバ `id` と `name` を設定し、デストラクタはコンストラクタで割り当てられたメモリを解放します。以下に、`main` 関数内の初期化を含む宣言を示します。

```
int main() {
    class Student s1 = Student(1, "Jason");
    class Student s2 = Student(2, "Bob");
    ....
}
```

宣言時にコンストラクタが呼び出された後、`s1` および `s2` のデータメンバが設定されます。関数 **main()** が終了すると、デストラクタが呼び出されます。

19.5 演算子 new および delete

C では、動的メモリ割り当てと割り当て解除は、通常、関数 **malloc()** および **free()** によって実行されます。Ch と C++では、演算子 **new** と **delete** のペアが **malloc()** や **free()** と同じ処理を実行し、さらにその他のメリットも提供されます。

演算子 **new** は割り当てるメモリの正しいサイズを自動的に計算できますが、関数 **malloc()** ではメモリのサイズとして引数を受け取る必要があります。演算子 **new** は正しい型のポインタを返すことができますが、関数 **malloc()** では単に void へのポインタが返されます。最も重要なのは、演算子 **new** はクラスのコンストラクタを自動的に呼び出し、必要に応じて初期化を実行することができるのに対し、関数 **malloc()** は割り当てられたメモリの初期化を行わないという点です。対応するデストラクタは演算子 **delete** によって呼び出されます。

次のコードに、演算子 **new** と **delete** の使用方法を示します。

```
int main() {
    class Student *s1 = new Student (1, "Jason");
    class Student *s2 = new Student (2, "Bob");
    ...

    s1->setID((5); // change ID of s1 to 5
    ...

    delete s1;
    delete s2;
}
```

ユーザーにとって、この例は初期化を含む前の例と同じ処理を実行しますが、この例の方が柔軟で便利である場合があります。メモリ割り当ての試行が成功した場合、

演算子 `new` は割り当てられたメモリへのポインタを返します。その他の場合、この演算子は、**_new_handler** が NULL でない場合に **_new_handler** が指すハンドラ関数を呼び出した後、NULL ポインタを返します。プログラムは、プログラムまたはライブラリ内で定義されている関数へのポインタを関数 **set_new_handler()** の引数として指定して、演算子 **new** のさまざまなハンドラ関数を実行中にインストールすることができます。関数 **set_new_handler()** は、ヘッダーファイル **new.h** 内で次のように定義されます。

```
void (*set_new_handler (void(*)()))();
```

これは、現在の **_new_handler** の引数によって指定される関数を作成し、最初の呼び出しで NULL を返すか、または後続の呼び出しで前の **_new_handler** を返します。プログラム 19.1では関数 `newhandle()` を演算子 **new** のハンドラ関数として設定し、変数 `p` と `sp` のそれぞれにメモリを割り当てます。システムには `p` のメモリは十分ありますが、`sp` には十分なメモリがありません。演算子 **new** が `sp` のメモリの割り当てに失敗すると、関数 `newhandler()` が呼び出されます。プログラム 19.1を実行した出力結果はファイルの最後に追加されています。

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

struct tag{ int i; int j[900000];} s;
void newhandler(void);

int main() {
  set_new_handler(newhandler);

  int *p = new int[20];
  if(p==NULL)
    printf("not enough memory for p\n\n");
  else
    printf("enough memory for p\n\n");


  tag *sp = new tag[90];
  if(sp==NULL) {
    printf("not enough memory for sp\n");
    printf("sp = %p\n", sp);
  }
  else {
    printf("enough memory for sp\n");
    printf("sp = %p\n", sp);
  }
}

void newhandler(void) {
   printf("message from newhandler\n");
}

/**** result of the program
newhandler.ch
enough memory for p

message from newhandler
not enough memory for sp
sp = 00000000
****/
```

プログラム 19.1: 演算子 **new** のハンドラ関数の設定

クラスへのポインタの変数では、クラスのメンバにアクセスするには演算子 ‘–>’ を使用する必要があります。

演算子 **new** および **delete** は単一の値だけでなく、配列も処理できます。たとえば、次のコードがあります。

```
class Employee {
    char *name;
};
int main() {
    class Employee *e = new Employee[10];
```

```
        ....

        delete [10] e;
    }
```

このコードでは、クラス Employee の 10 個の新しいオブジェクトをインスタンス化します。プログラムの最後で、これらの 10 個のオブジェクトすべてが削除されます。

19.6 クラスの静的メンバ

通常、クラスの各オブジェクトはメモリ内にそのオブジェクト固有のデータメンバのコピーがあります。しかし、あるクラスの異なるオブジェクトが “クラス全体” の情報を使用する必要がある場合があります。つまり、それらのオブジェクトが 1 つの変数の同一のコピーを共有する必要があります。静的クラス変数では、このメカニズムを提供できます。クラスのすべてのオブジェクト内の静的メンバの値は同じです。値の変化はすべてのオブジェクトに影響します。クラスのオブジェクトが存在しない場合でも、静的メンバは依然として存在し、操作することができます。静的メンバの宣言では、先頭にキーワード **static** を付けます。たとえば、次のように、クラス Student の定義に静的メンバ count を追加することができます。

```
class Student {
    // number of objects instantiated
    static int count;
    int id;
    char *name;
  public:
    Student(int, char *);
    ~Student();
    void setID(int i);
    void setName(const char *n);
};
```

ここで、メンバ count はクラス Student のオブジェクト数を維持します。他のメンバ関数の定義と共に、次のステートメントを使用して静的データメンバを初期化することができます。静的データメンバはファイルスコープで一度初期化する必要があります。次に例を示します。

```
int Student::count = 0;
```

メンバ count は Student オブジェクトの任意のメンバ関数を使用して参照できます。この例では、コンストラクタは count に 1 を加算し、デストラクタは count から 1 を減算します。コンストラクタとデストラクタを次のように書き直すことができます。

```
Student::Student(int i, char *n) {
    id = i;             /* initialize id */
```

```
        name = strdup(n);  /* initialize name */
        count++;
    }

    Student::~Student() {
        /* release the memory allocated in constructor */
        free(name);
        count--;
    }
```

C++と同様に、Ch には単純なデータ型の静的メンバだけでなく、以下に定義するメンバ関数 getCount() などの静的メンバ関数もあります。

```
   class Student {
        // number of objects instantiated
        static int count;
        int id;
        char *name;
      public:
        Student(int, char *);
        ~Student();
        void setID(int i);
        void setName(const char *n);
        static int getCount();
    };

    int Student::getCount() {
        return count;
    }
```

この関数を使用すると、次のように、現在インスタンス化されているオブジェクト数を取得することができます。

```
     int main() {
         class Student s1 = Student(1, "Jason");
         ....

         cout << "Number of student is "
              << s1.getCount() << endl;
         ....
     }
```

オブジェクトがインスタンス化されていなくても、静的メンバ関数を呼び出すことができます。つまり、s1 がインスタンス化される前に、次の例のように getCount() を呼び出すことができます。

```
int main() {
    ....
    cout << "Number of student is "
         << Student::getCount() << endl;
    ....
}
```

19.7 スコープ解決演算子 ::

Ch および C++では、同じ名前のローカル変数がスコープ内にあるときにグローバル変数にアクセスするための単項スコープ解決演算子 '::' が提供されています。次に例を示します。

```
#include <stdlib.h>
    int num;
    int main() {
      int num;
      num = 10;          // use local num
      ::num = ::num+2    // use global num

      ...

      ::exit(0);         // use C function exit()
    }
```

また、この演算子はクラスでよく使用します。前の例で既にスコープ解決演算子 '::' を使用しています。この演算子は主に次のような場合に使用します。
1. メンバ関数の定義。メンバ関数がクラス定義の後に定義されている場合は、関数名の前にクラス名とスコープ解決演算子 '::' が付けられます。異なるクラスが同じ名前のメンバを含む可能性があるため、スコープ解決により混乱を防ぐことができます。たとえば、前の例のメンバ関数 getCount() は、次のように定義されます。

```
... /* definition of the class Student */

int Student::getCount() {
    return count;
}
...
```

2. 静的メンバへのアクセス。前述したように、クラスのオブジェクトが存在しない場合でも、静的データメンバ名の前にクラス名とスコープ解決演算子 '::' を追加することによって、そのクラスの静的メンバにアクセスできます。たとえば、次のコードがあります。

```
int main() {
```

```
        ....
        cout << "Number of student is "
             << Student::getCount() << endl;
        ....
    }
```

このコードではオブジェクトが宣言されていない場合でも、クラス Student のオブジェクト数を出力することができます。

19.8 暗黙のポインタ this

Ch および C++では、すべてのオブジェクトにオブジェクト自身のアドレスを指す this という暗黙のポインタがあります。ポインタ this はオブジェクトの一部とは見なされません。つまり、このポインタは **sizeof()** 演算には反映されません。しかし、実際には暗黙的に、オブジェクトのデータメンバおよびメンバ関数の参照に使用されます。次のようなメンバ関数の定義

```
void setID(int i) {
    id = i;
}

void setName(const char *n) {
    if(n)
      name = strdup(n);
}
```

は以下と同等です。

```
void Student::setID(int i) {
    this->id = i;
}

void Student::setName(const char *n) {
    if(n)
      this->name = strdup(n);
}
```

ここで、this ポインタは明示的に使用されています。静的メンバ関数には this ポインタがありません。これは、この関数がクラスのどのオブジェクトからも独立して存在しているためです。

19.9 ポリモーフィズム

Ch はユーザーレベルでの演算子のオーバーロードをサポートしていませんが、一般的に使用される算術演算子が内部的にオーバーロードされ、さまざまなデータ型のオペランドを処理します。たと

えば、演算子”+”は、整数、浮動小数点数、複素数、および異なるデータ型の計算配列の加算に使用できます。

Ch ではポリモーフィズム、つまり、同じ関数呼び出しに対して異なる処理を実行する通常の関数およびクラスのメンバ関数の機能をサポートしていますが、異なる引数の個数や型を使用します。Ch はユーザーレベルでの C++関数のオーバーロード、つまり、異なるデータ型や引数を使用して複数の関数を定義できる機能をサポートしていません。これは、C++コンパイラ内で内部的に関数名を分割することによって実現します。この名前の分割は、オーバーヘッドのため単一パスでのインタープリタ実装に適していません。Ch のポリモーフィズムは主に関数の再読み込みによって実装されます。このセクションでは、Ch でのポリモーフィズム関数の処理方法の概要を説明します。

19.9.1 ポリモーフィックな汎用数学関数

よく使用される sin() などの汎用数学関数はポリモーフィックです。汎用数学関数は、整数値、浮動小数点数値、複素数値、および異なるデータ型とサイズを持つ計算配列の引数を処理できます。実数、複素数、および異なるデータ型とサイズを持つ計算配列の汎用数学関数については、それぞれセクション 12.1、 13.5、および 16.10で説明しました。次の例では、Ch シェルで異なる引数を使用して汎用数学関数 **sin()** を呼び出します。

```
> float f = 1.0
> sin(f)        // call with a float
0.84
> double d = 1.0
> sin(d)        // call with a double
0.8415
> complex float zf = 1
> sin(zf)       // call with float complex
complex(0.84,0.00)
> complex double zd = 1
> sin(zd)       // call with doulbe complex
complex(0.8415,0.0000)
> array float af1[2] = {1.0, 2.0}
> sin(af1)      // call with a one-dimensional array
0.84 0.91
> array float af2[2][3] = {1.0, 2.0, 3.0, 4.0, 5.0, 6.0}
> sin(af2)      // call with a two-dimensional array
0.84 0.91 0.14
-0.76 -0.96 -0.28
> array double ad1[2] = {1.0, 2.0}
> sin(ad1)      // call with a double array
0.8415 0.9093
> array complex double az1[2] = {1.0, 2.0}
> sin(az1)      // call with a complex array
```

```
complex(0.8415,0.0000) complex(0.9093,-0.0000)
>
```

19.9.2 参照配列型のパラメータを含む関数

参照配列のパラメータを含む関数は、次元およびデータ型が異なる配列引数の処理に使用できます。たとえば、次のプロトタイプを含む関数 func() があります。

```
int func(double a[&][&], array double &b);
```

のプロトタイプを含む関数 func() は、次のように異なる次元および型の引数を受け取ることができます。

```
int func(double a[&][&], array double &b);
array double A1[2][3], B1[10];
array float  A2[4][5], B2[3][4];
int func(A1, B1);
int func(A2, B2);
```

参照配列のパラメータ型を含む関数を使用して異なる次元および型の引数を受け渡す方法の詳細については、セクション 16.7.5で説明しました。

19.9.3 ポリモーフィックな関数

Ch ではポリモーフィックな汎用数学関数は組み込みの関数として実装されます。セクション 10.7で説明したヘッダーファイル **stdarg.h** 内で定義される標準ライブラリの機能を使用すると、ポリモーフィックな関数を記述することができます。ヘッダーファイル **stdarg.h** には、可変長の引数を処理するための表 10.1の一覧にある関数プロトタイプとマクロが含まれています。このセクションでは、いくつかのサンプルコードを使用して、ユーザープログラムでポリモーフィックな関数を実装する **VA_NOARG**、**va_count**、**va_datatype**、**va_arraydim**、**va_arrayextent**、**va_arraynum**、および **va_arraytype** の各マクロの使用方法を説明します。

プログラム 19.2の関数 func() は、異なるデータ型の可変長の引数の値を出力します。関数内では、マクロ **va_count()** が引数リスト内の残りの引数の数を返します。したがって、while ループを使用します。

```
while(va_count(ap)) {
  ...
}
```

このループで、すべての引数を 1 つずつ抽出できます。図 19.1にプログラム 19.2の出力を示します。

```
#include<stdarg.h>

void func(...) {
  va_list ap;
  int arg_num = 0;
  int i;
  float f;
  double d;

  va_start(ap, VA_NOARG);
  while(va_count(ap)) {
    if(va_datatype(ap)  == elementtype(int)) {
      i = va_arg(ap, int);
      printf("the %d argument is int %d\n", ++arg_num, i);
    }
    else if(va_datatype(ap)  == elementtype(float)) {
      f = va_arg(ap, float);
      printf("the %d argument is float %f\n", ++arg_num, f);
    }
    else if(va_datatype(ap)  == elementtype(double)) {
      d = va_arg(ap, double);
      printf("the %d argument is double %f\n", ++arg_num, d);
    }
  }
  va_end(ap);
  return;
}

int main(){
  int i = 10;
  float f = 2.0;
  double d = 3.0;

  func(i, f);
  func(f, i, d);  // different types and different order
  return 0;
}
```

プログラム 19.2: 可変長引数を処理するポリモーフィックな関数

```
the 1 argument is int 10
the 2 argument is float 2.000000
the 1 argument is float 2.000000
the 2 argument is int 10
the 3 argument is double 3.000000
```

図 19.1: プログラム 19.2の出力結果

正規のデータ型のほかに、可変長の引数リスト内で異なるデータ型の配列をポインタとして渡すことができます。**va_arraytype**()、**va_datatype**()、**va_arraydim**()、**va_arrayextent**()、および **va_arraynum**()の各マクロを使用すると、計算配列またはC配列のデータ型、次元、エクステントおよび要素数を取

得することができます。これらのマクロは、マクロ **va_arg** が呼び出される前に呼び出す必要があります。たとえば、プログラム 19.3を考えてみます。

```
#include <array.h>
#include <stdarg.h>
int fun(...) {
   int *ptr, count, dim,num, m, n;
   va_list ap;

   va_start(ap, noarg);
   count = va_count(ap);
   printf("count = %d\n", count);
   if(count >= 1) {
     if(va_arraytype(ap) == CH_UNDEFINETYPE) { // check if it is not an array
       printf("the argument is not an array\n");
       return -1;
     }
     dim = va_arraydim(ap);     // get the dimension
     printf("dim = %d\n",dim);
     num = va_arraynum(ap);     // get the number of element
     printf("num= %d\n",num);
     if(va_datatype(ap)  == elementtype(int)) // get the type
     // or if(va_datatype(ap)  == CH_INTTYPE)
       printf("elementtype = int\n");
     else
       printf("elementtype = other types\n");
     m = va_arrayextent(ap, 0);  // get the extent
     if(dim == 1) {
       printf("extent1 = %d\n",m);
       array int *p;
       ptr = va_arg(ap, int*);
       p =  (array int [:])(int [m])ptr;
       printf("p =\n%d\n", p);
     }
     if(dim == 2) {
       array int (*p)[:];
       n = va_arrayextent(ap, 1);
       printf("extent1 = %d, extent2 = %d\n", m, n);
       ptr = va_arg(ap, int*);
       p =  (array int [:][:])(int [m][n])ptr;
       printf("p =\n%d\n", p);
     }
   }
   va_end(ap);
   return 0;
}

int main() {
     array int A1[2][3];
     array int (*A2)[:];
     array int A3[3] = {1, 2, 3};

     A1 = (array int [2][3])50;
     fun(A1);

     A1 = (array int [2][3])80;
     A2 = (array int [:][:])A1;
     fun(A2);
     fun(A3);
}
```

プログラム 19.3: 関数への計算配列の受け渡し

この関数 fun() にあるステートメント

```
 if(va_arraytype(ap) == CH_UNDEFINETYPE) { // check if it is not an array
   printf("the argument is not an array\n");
   return -1;
 }
```

では引数が配列でない場合、エラーメッセージを出力します。次のステートメント

```
if(va_datatype(ap) == elementtype(int)) {
/* if(va_datatype(ap) == CH_INTTYPE) {
  printf("elementtype = int\n");
}
else {
  printf("elementtype = other types\n");
}
```

ではマクロ **va_datatype**() および汎用関数 **elementtype()** を呼び出すことにより、引数の型が **int** であるかどうかを判定し、対応するメッセージを出力します。汎用関数 **elementtype()** はヘッダーファイル **stdarg.h** 内に定義されたデータ型を指定します。Ch では **char***および **string_t** は両方とも文字列を表すため、次のステートメントを文字列型の判定に使用できる場合があります。

```
if(va_datatype(ap)==elementtype(char*)
   || va_datatype(ap)==elementtype(string_t)) {
  printf("element type is string\n");
}
else {
  printf("element type is not string\n");
}
```

19.3の例で、ステートメント

```
dim = va_arraydim(ap);
num = va_arraynum(ap);
```

は、配列引数の次元と要素数を取得します。次のステートメント

```
m = va_arrayextent(ap, 0);
```

は、配列の最初の次元内の要素数を取得します。配列が一次元である場合、ステートメント

```
array int *p;
ptr = va_arg(ap, int*);
p = (array int [:])(int [m])ptr;
```

は、p が、main() 関数内の A3 などの一次元配列とメモリを共有するようにします。配列が 2 次元である場合、次のステートメント

```
array int (*p)[:];
n = va_arrayextent(ap, 1);
ptr = va_arg(ap, int*);
p =  (array int [:][:])(int [m][n])ptr;
```

は、p が、main() 関数内の A1 および A2 などの 2 次元配列とメモリを共有するようにします。A1 は形状が完全に指定された計算配列であり、A2 は形状引継ぎの計算配列です。A1 のように、これらは int へのポインタとして関数 fun() に渡されます。計算配列の詳細については、第 16章を参照してください。図 19.2にプログラム 19.3の出力を示します。

```
count = 1
dim = 2
num= 6
elementtype = int
extent1 = 2, extent2 = 3
p =
50 50 50
50 50 50

count = 1
dim = 2
num= 6
elementtype = int
extent1 = 2, extent2 = 3
p =
80 80 80
80 80 80

count = 1
dim = 1
num= 3
elementtype = int
extent1 = 3
p =
1 2 3
```

図 19.2: プログラム 19.3の出力結果

Ch でのポリモーフィックな関数に対する 1 つの制限は、異なるデータ型の値を返すことができないということです。関数から異なるデータ型の結果を取得する場合は、引数としてポインタを渡し、異なる型の値を取得することによって実装できます。たとえば、プログラム 19.4の関数 func() では、ポインタである最初の引数を使用し、戻り値として int または float の値を取得できます。

```
#include<stdarg.h>

void func(...) {
  va_list ap;
  int *pi, flag;
  float *pf;

  va_start(ap, VA_NOARG);
  if(va_count(ap) != 2) {
    printf("need 2 arguments\n");
    return;
  }

  if(va_datatype(ap)  == elementtype(int *)) { // get the 1st argument
    pi = va_arg(ap, int*);
    flag = 1;
  }
  else if(va_datatype(ap)  == elementtype(float *)) {
    pf = va_arg(ap, float*);
    flag = 2;
  }

  if(va_datatype(ap)  == elementtype(int)) {// get the 2nd argument
    if(flag == 1)
      *pi = va_arg(ap, int);
  }
  else if(va_datatype(ap)  == elementtype(float)) {
    if(flag == 2)
      *pf = va_arg(ap, float);
  }
  va_end(ap);
  return;
}

int main(){
  int ret_i, i = 10;
  float ret_f, f = 1.0;

  func(&ret_i, i);
  printf("ret_i = %d\n", ret_i);
  func(&ret_f, f);
  printf("ret_f = %f\n", ret_f);
  return 0;
}
```

プログラム 19.4: 異なるデータ型を返す関数の多態性

図 19.3にプログラム 19.4の出力を示します。

```
ret_i = 10
ret_f = 1.000000
```

図 19.3: プログラム 19.4の出力結果

19.9.4 クラスのポリモーフィックなメンバ関数

Ch では、関数だけでなくクラスのコンストラクタやメンバ関数も、可変長の引数を処理するためのヘッダーファイル **stdarg.h** の機能を使用して、ポリモーフィックにすることができます。

プログラム 19.5では、クラス `C1` のコンストラクタ `C1()` とメンバ関数 `memfunc()` の両方が、int 型である可変個の引数を取ることができます。

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

class C1 {
    double m_d;
  public:
    C1(...);      // constructor taking variable length arguments
    void memfunc(...); // member function taking variable length arguments
};

C1::C1(...) {
  va_list ap;
  int vacount;

  va_start(ap, VA_NOARG);
  vacount = va_count(ap);
  m_d = 0;
  if(vacount == 1 || vacount == 2) { /* integral value for 1st arg */
    if(va_datatype(ap)  <= elementtype(int)) {
      m_d += va_arg(ap, int);
    }
    else {
      printf("Error: wrong data type\n");
    }
  }
  else {
      printf("Error: wrong number of arguments\n");
  }

  if(vacount == 2) { /* floating-point number for 2nd arg */
    if(va_datatype(ap)  == elementtype(float)) {
      m_d += va_arg(ap, float);
    }
    else if(va_datatype(ap)  == elementtype(double)) {
      m_d += va_arg(ap, double);
    }
    else {
      printf("Error: wrong data type\n");
    }
  }
  va_end(ap);
}
```

プログラム 19.5: 可変長の引数リストを使用するメンバ関数

```
void C1::memfunc(...) {
  va_list ap;
  int vacount;
  int i, num = 0;

  printf("m_d = %f\n",  m_d);
  va_start(ap, VA_NOARG);
  vacount = va_count(ap);
  printf("vacount = %d\n", vacount);
  while(num++, vacount--) {
    i = va_arg(ap, int);
    printf("argument %d = %d, ", num, i);
  }
  printf("\n\n");
  va_end(ap);
  return;
}

int main() {
  class C1 c1 = C1(3);
  class C1 c2 = C1(3, 6.5);
  c1.memfunc(1);
  c2.memfunc(1, 2, 3);

  return 0;
}
```

プログラム 19.5: 可変長の引数リストを使用するメンバ関数 (続き)

オブジェクト c1 および c2 は、それぞれ 1 つおよび 2 つの引数を取るコンストラクタを使用してインスタンス化されます。図 19.4にプログラム 19.5の出力を示します。

```
m_d = 3.000000
vacount = 1
argument 1 = 1,

m_d = 9.500000
vacount = 3
argument 1 = 1, argument 2 = 2, argument 3 = 3,
```

図 19.4: プログラム 19.5の出力結果

19.10 入れ子にされたクラス

入れ子にされたクラスとは、他のクラスのスコープ内を含むクラスです。入れ子にされたクラスが定義されるクラスは、**包含クラス**または**包囲クラス**と呼ばれます。Ch では入れ子にされたクラスがサポートされています。

包含クラスのすべての部分でクラスを入れ子にすることができます。入れ子にされたクラスは、実際は、包含クラスのメンバとして見なされます。そのため、入れ子にされたクラスには、クラスのアクセスおよび可視性に関する通常の規則が適用されます。

クラスの public セクションで入れ子にされているクラスの場合、包含クラスの外部から参照可能です。private セクションで入れ子にされている場合は、包含クラスのメンバのみが参照可能です。

入れ子にされたクラスは包含クラスのメンバとして見なされますが、そのメンバは包含クラスのメンバではありません。そのため、包含クラスのメンバ関数には入れ子にされたクラスのメンバに対する特定のアクセス権がありません。その一方で、入れ子にされたクラスのメンバ関数も通常のアクセス規則に従っており、包含クラスのメンバに対する特定のアクセス権限がありません。

プログラム 19.6に、包含クラス `Encl` 内で入れ子にされたクラスを定義する方法を示します。

```
/* Nested classes */

#include <iostream.h>

class Encl {
    public:
        Encl(int); /* constructor */
        int getVar();

        class nestPub {
            public:
                int getVar();
            private:
                int variable;
        };

    private:
        class nestPrv{
            public:
                int getVar();
            private:
                int variable;
        }nPr;
        int variable;
};

Encl::Encl(int var) {
    variable = var;
}

int Encl::getVar() {
    return variable;
}

int Encl::nestPub::getVar() {
    return variable;
}

int Encl::nestPrv::getVar() {
    return variable;
}

int main() {
   Encl e1 = Encl(5);
   cout << "variable = " << e1.getVar() << endl;

   return 0;
}
```

プログラム 19.6: 入れ子にされたクラス

この例では、メンバへのアクセスは次のように定義されます。

1. 入れ子にされた public クラス `nestPub` は、包含クラス `Encl` の内部および外部の両方から参照可能です。

2. クラス `nestPub` の public メンバ関数 `getVar()` もグローバルに参照可能です。
3. クラス `nestPub` の private データメンバ `variable` は、クラス `nestPub` のメンバのみアクセス可能です。
4. private クラス `nestPrv` は包含クラス `Encl` 内でのみ参照可能です。
5. クラス `nestPrv` の public メンバは、入れ子にされた public クラス `nestPub` のメンバで使用することができます。
6. クラス `nestPrv` の public メンバ関数 `getVar()` は、包含クラス `Encl` のメンバとその入れ子にされたクラスのメンバによってのみアクセス可能です。
7. クラス `nestPrv()` の private データメンバ `variable` は、クラス `nestPrv()` のメンバのみ参照可能です。

プログラム 19.6では、入れ子にされたクラスの定義のほか、そのメンバ関数も定義されています。

入れ子にされたクラスのメンバ関数の定義は、通常クラスのメンバ関数の定義と同様です。関数名の先頭には、包含クラス名と入れ子にされたクラス名の両方が付けられます。`nestPub` および `nestPrv` の両方に `getVar()` という名前のメンバ関数があるため、2 つのスコープ解決演算子 '`::`' が使用されます。スコープ解決によりこの混乱を防ぐことができます。

19.11 メンバ関数内のクラス

C++への拡張機能として、Ch にはメンバ関数内のクラスが用意されています。Ch では、他のクラスのメンバ関数で定義されるクラスは、メンバ関数内のクラスと呼ばれます。プログラム 19.7に、メンバ関数 `C1::func()` にクラス `C2` を定義する方法を示します。

```
/* Classes inside member functions */

#include <iostream.h>

int main () {
  int t;

  class C1 {
      int v1;
  public:
      int func();
  };

  int C1::func() {
      class C2 {
          int v2;
      public:
          int func2();
          int v3;
      };
      int C2::func2() {
          class C1 c;
          class C2 c2;
          v2 = 10;
          c.v1 = 20;
          c2.v2 = 30;
          return 10;
      }
      C2 c2;
      /* c2.v2 = 30; is wrong */
      c2.func2();
      c2.v3 = 50;

      v1 = 30;
      return v1;
  }

  C1 s;
  /* C2 s2; is wrong */
  cout << s.func() << endl;

  return 0;
}
```

プログラム 19.7: メンバ関数内のクラス

この例では、メンバへのアクセスは次のように定義されます。

1. メンバ関数内のクラスは、そのメンバ関数が public であるか private であるかにかかわらず、定義しているメンバ関数内でのみ参照可能です。この例で、メンバ関数 `C1::func()` の外部でクラス `C2` 型の変数を宣言すると、Ch では構文エラーとなります。
2. クラス `C2` を入れ子にしているメンバ関数 `C1::func()` には、`C2` のメンバに対する特定のアクセス権限はありません。

3. `C2` の public メンバには、`C1::func()` 内からアクセス可能です。
4. `C2` の private メンバには、自身のメンバのみアクセス可能です。

19.12　関数の引数としてのメンバ関数の受け渡し

C++でサポートされておらず、Ch ではサポートされているもう 1 つの機能として、メンバ関数を関数へのポインタである引数として関数に渡すことがあります。プログラム 19.8で、メンバ関数`C1::f5()`と `C2::f()`、および通常関数 `func()` は、関数へのポインタである引数を受け取ります。

```
#include <stdio.h>
/* pass member function to a function */
/* normal function with argument of pointer to function */
int func(void (*fp)()) {
   printf("func() called\n");
   fp();
   return 0;
}

class C1 {
   int i;
   void f1(); // private member function access i
public:
   C1();
   void f2(); // access member i
   void f3(); // does not access any member
   void f4(); // call func()
   /* function with argument of pointer to function */
   void f5(void (*fp)());
};
C1::C1() {
   i = 5;
}
void C1::f1() { // private member function
   printf("C1::f1() called, i = %d\n", i);
}
void C1::f2() {
   printf("C1::f2() called, i = %d\n", i);
}
void C1::f3() {
   printf("C1::f3() called\n");
}
/* member function with argument of pointer to function */
void C1::f4() {
   func(f1); /* pass private function, ok in Ch and bad in C++ */
   func(f2); /* pass public function,  ok in Ch and bad in C++ */
}
/* member function with argument of pointer to function */
void C1::f5(void (*fp)()) {
   printf("C1::f5() called \n");
   fp(); /* function as argument */
}

class C2 {
   int d;
public:
   C2();
   /* function with argument of pointer to function */
   void f(void (*fp)());
};
C2::C2() {
   d = 10;
}
/* member function with argument of pointer to function */
void C2::f(void (*fp)()) {
   fp(); /* function as argument */
}
```

プログラム 19.8: メンバ関数を引数として関数に受け渡す

```
int main() {
   class C1 s;
   class C2 s2;

   printf("(1) passed member func to regular func\n");
   func(s.f2);  // OK in Ch, bad in C++
   func(s.f3);  // OK in Ch, bad in C++
   printf("(2) passed member func to regular func inside member func\n");
   s.f4();
   printf("(3) passed member func to member func of the same class \n");
   s.f5(s.f2);  // OK in Ch, bad in C++
   s.f5(s.f3);  // OK in Ch, bad in C++
   printf("(4) passed member func, without accessing member field,\n");
   printf("    to member func of a diff class.\n");
   s2.f(s.f3);  // Ok in Ch, bad in C++

   printf("\n(5) Error: passed member func, with accessing member field,\n");
   printf("    to member func of a diff class.\n");
   s2.f(s.f2);  // bad in Ch and C++
   return 0;
}
```

プログラム 19.8: メンバ関数を引数として関数に受け渡す (続き)

プログラム 19.8を実行した出力結果は次のとおりです。

```
(1) passed member func to regular func
func() called
C1::f2() called, i = 5
func() called
C1::f3() called
(2) passed member func to regular func inside member func
func() called
C1::f1() called, i = 5
func() called
C1::f2() called, i = 5
(3) passed member func to member func of the same class
C1::f5() called
C1::f2() called, i = 5
C1::f5() called
C1::f3() called
(4) passed member func, without accessing member field,
    to member func of a diff class.
C1::f3() called

(5) Error: passed member func, with accessing member field,
    to member func of a diff class.
C1::f2() called, i = 10
```

Ch でメンバ関数を関数の引数に渡す場合は、クラスメンバの private メンバおよび public メンバにアクセスする通常の規則に従います。たとえば、private メンバ関数は、クラスのメンバのみが関数の引数として使用することができます。ただし、メンバ関数をポインタの引数として関数に渡すには、次のようないくつかの追加の制約があります。

1. プログラム 19.8の次のステートメントのように、メンバ関数を関数のポインタの引数として通常関数に渡すことができます。

   ```
   func(s.f2);  // OK in Ch, bad in C++
   func(s.f3);  // OK in Ch, bad in C++
   ```

   この場合、C1::f2() などの渡されたメンバ関数は、クラスのメンバにアクセスできます。

2. メンバ関数内で、メンバ関数を関数へのポインタの引数として通常関数に渡すことができます。この場合、プログラム 19.8の関数呼び出し `s.f4()` に示すように、渡されたメンバ関数はそのメンバにアクセスできます。

3. メンバ関数を、クラスの同じインスタンスのメンバ関数に関数へのポインタの引数として渡すことができます。この場合、プログラム 19.8の次のステートメントのように、渡されたメンバ関数はそのメンバにアクセスできます。

   ```
   s.f5(s.f2);  // OK in Ch, bad in C++
   s.f5(s.f3);  // OK in Ch, bad in C++
   ```

4. メンバ関数を、異なるクラスのメンバ関数に関数へのポインタの引数として渡すことができます。この場合、プログラム 19.8の次のステートメントのように、渡されたメンバ関数はそのメンバにアクセスできません。

   ```
   s2.f(s.f3); // Ok in Ch, bad in C++
   ```

   `s` および `s2` は、異なるクラスのインスタンスです。ただし、メンバ関数 `C1::f3()` は、メンバフィールドにアクセスしないため、異なるクラスのメンバ関数に引数として渡すことができます。

5. メンバにアクセスするメンバ関数は、プログラム 19.8の次のステートメントのように、異なるクラスのメンバ関数に関数へのポインタの引数として渡すことができません。

   ```
   s2.f(s.f2); // bad in Ch, bad in C++
   ```

   `s` と `s2` は、異なるクラスのインスタンスです。メンバ関数 `C1::f2()` はメンバフィールドにアクセスするので、異なるクラスのメンバ関数に引数として渡すことができません。この場合、`s` のメモリは `s2` のメモリ範囲に制約されます。

19.13 事前定義済みの識別子 __class__ と __class_func__

事前定義識別子**__func__**と同様に、事前定義識別子**__class__**および**__class_func__**を使用すると、クラス名およびメンバ関数内のクラス名と関数名の両方を取得できます。識別子**__class__**および**__class_func__**は、各メンバ関数定義の左かっこの直後に指定するかのように暗黙的に宣言されます。

```
static const char __class__[] = "class-name";
static const char __class_func_[] = "class-name:function-name";
```

次の宣言で、class-name はクラス名であり、function-name はクラスのメンバ関数を語彙的に包含する名前です。

たとえば、プログラム 19.9は、これらの事前定義済みの識別子を使用して、クラスとメンバ関数の名前を出力します。

```
/* Filename: classname.ch */
#include <iostream>
class tag{
    public:
      tag(int i1); /* only one argument */
      ~tag();
      void func(void);
    private:
      int m_i;
};

tag::tag(int i) {
    cout << "__func__ in tag::tag() = " << __func__ << endl;
    cout << "__class_ in tag::tag() = " << __class__ << endl;
    cout << "__class_func__ in tag::tag() = " << __class_func__ << endl;
    m_i = i;
}

tag::~tag() {
    cout << "__func__ in tag::~tag() = " << __func__ << endl;
    cout << "__class_ in tag::~tag() = " << __class__ << endl;
    cout << "__class_func__ in tag::~tag() = " << __class_func__ << endl;
}

void tag::func(void) {
    cout << "__func__ in tag::func() = " << __func__ << endl;
    cout << "__class_ in tag::func() = " << __class__ << endl;
    cout << "__class_func__ in tag::func() = " << __class_func__ << endl;
}

int main() {
    struct tag s = tag(10);
    s.func();
    return 0;
}
```

プログラム 19.9: __class__ および __class_func__ を使用したクラス名および関数名の取得

出力は次のとおりです。

```
__func__ in tag::tag() = tag
__class_ in tag::tag() = tag
__class_func__ in tag::tag() = tag::tag
__func__ in tag::func() = func
__class_ in tag::func() = tag
__class_func__ in tag::func() = tag::func
__func__ in tag::~tag() = ~tag
__class_ in tag::~tag() = tag
__class_func__ in tag::~tag() = tag::~tag
```

第20章　入出力

20.1　ストリーム

Ch では、端末やテープドライブなどの物理デバイスからの入出力や構造化された記憶装置に格納されたファイルからの入出力など、すべての入出力は論理データストリームにマッピングされます。このストリームの特性は、さまざまな種類の入力や出力にかかわらず一貫しています。テキストストリームとバイナリストリームという 2 つのマッピング形式がサポートされます。テキストストリームは、行を構成する順序付けられた文字のシーケンスです。各行はゼロ個以上の文字と行末の改行文字で構成されています。バイナリストリームも順序付けられているバイトのシーケンスですが、内部データを透過的に記録しています。

各ストリームには指向性があります。ストリームを外部ファイルに関連付けた後、ストリーム上で操作を実行するまでは、ストリームは無指向の状態です。ワイド文字の入出力関数を無指向ストリームに適用すると、ストリームはワイド指向ストリームになります。同様に、バイトの入出力関数を無指向ストリームに適用すると、ストリームはバイト指向ストリームになります。ストリームの指向性を変更できるその他の方法は、**freopen()** または **fwide()** への関数呼び出しだけです。**freopen()** への関数呼び出しに成功すると、指向性はなくなります。バイトの入出力関数はワイド指向ストリームに適用するべきではありません。また、ワイド文字の入出力関数をバイト指向ストリームに適用するべきではありません。

ワイド指向の各ストリームには関連付けられた **mbstate_t** オブジェクトがあり、そこにストリームの現在の解析状態が格納されています。**fgetpos()** への関数呼び出しに成功した場合は、この **mbstate_t** オブジェクトの値を **fpos_t** オブジェクトの値の一部として示す表現が格納されます。格納されている **fpos_t** の値と同じ値を使用して **fsetpos()** への関数呼び出しをもう一度行い、 その呼び出しに成功すると、関連付けられている **mbstate_t** オブジェクトの値と制御ストリーム内での位置が復元されます。

Ch プログラムが実行を開始すると、事前に定義された 3 つのテキストストリームが開きます。**stdin** ストリームは標準入力、**stdout** ストリームは標準出力、および **stderr** ストリームは標準エラー出力を表します。これらはヘッダーファイル **stdio.h** で定義されます。

20.2　入出力のバッファリングとノンバッファリング

Ch では、ストリームをバッファリングすることも、バッファリングしない *(アンバッファリング)* こともできます。ストリームをバッファリングしない場合、文字は可能な限り迅速にソースから取り出されるか、ターゲットに置かれます。そうでない場合は、文字を蓄積し、ブロックにまとめてホスト環境との間で送受信します。

また、バッファリングするストリームはフルバッファリングまたはラインバッファリングすることができます。ストリームをフルバッファリングする場合、バッファがいっぱいになったときに文字を

ブロックにまとめてホスト環境との間で送受信します。ストリームをラインバッファリングする場合は、改行文字が出現するたびに、ホスト環境との間で文字をブロックにまとめて送受信します。

Ch の既定では、端末への出力はラインバッファリング、その他すべての入出力はフルバッファリングですが、ストリーム **stderr** はバッファリングされません。

関数 **setbuf()** と **setvbuf()** を使用してストリームファイルに バッファリングを割り当てることができます。この 2 つの関数のプロトタイプは次のとおりです。

```
void setbuf(FILE *stream, char *buf);
int setvbuf(FILE *stream, char *buf, int type, size_t size);
```

関数 **setbuf()** を使用できるのは、ストリームを開いてから、ストリームの読み込みまたは書き込みを行う前までの間です。これにより、自動割り当てされるバッファではなく、引数 `buf` が指すバッファが使用されます。`buf` の値が **NULL** でない場合、入出力はフルバッファリングされます。NULL の場合は、完全にノンバッファリングです。バッファのサイズは、ヘッダーファイル **stdio.h** で定義されている定数 **BUFSIZ** によって指定します。**Ch** では、この値はプラットフォームよって異なります。次に例を示します。

```
char buf[BUFSIZ];
FILE *ptf = fopen("example", "a+");
if(ptf)
  setbuf(ptf, buf);  // set user-defined buffer
```

関数 **setvbuf()** を使用すると、ストリームファイルへのバッファリングの割り当てをより柔軟に行うことができます。関数 **setbuf()** と同様に、この関数を使用できるのは、`stream` が指すストリームを開いた状態のファイルに関連付けた後、ストリーム上で他の操作を実行するまでの間です。引数 `type` で、`stream` のバッファリング方法が決まります。ヘッダーファイル **stdio.h** で定義されている型の有効な値を表 20.1に示します。

表 20.1: 関数 **setvbuf()** のバッファリングの型

型	説明
_IOFBF	入出力をフルバッファリングします。
_IOLBF	入出力をラインバッファリングします。
_IONBF	入出力をバッファリングしません。

`buf` が指すバッファのサイズは、**BUFSIZ** ではなく、`size` で指定されます。**setvbuf()** 関数は、成功すると 0 を返します。引数 `type` の値が無効か、または要求を満たすことができない場合は、0 以外の値を返します。次の式

```
setbuf(stream, buf);
```

は、次の条件式と同じものを表します。

```
((buf == NULL) ?
 (void) setvbuf(stream, NULL, _IONBF, 0) :
 (void) setvbuf(stream, buf, _IOFBF, BUFSIZ))
```

関数 **fflush()** を使用して、ファイルストリームのバッファを明示的に解放することもできます。ファイルを閉じると、ファイルと制御ストリームの関連付けが解除されます。ストリームとファイルの関連付けが解除される前に、出力ストリームがフラッシュされます。関連付けられたファイルを閉じると、ファイルオブジェクトへのポインタの値は **NULL** になります。

20.3 入出力形式

C では、整数および浮動小数点数の入力は、**scanf**() や **fscanf**() などの標準の入力関数で行い、出力には **fprintf**() や **printf**() などの出力関数を使用します。 これらの関数は Ch でも使用でき、C の標準に完全に準拠しています。ただし、 Ch のこれらの関数のいくつかの実装は、コンパイラを使用する C の従来の実装とは異なります。このセクションでは、これらの関数の Ch と C での相違や Ch での機能拡張について詳しく説明します。Ch と C でのこれらの関数の主な違いは、Ch ではこれらの関数のいくつかは組み込みの内部関数になっていますが、C では外部関数であることです。したがって、これらの関数は Ch の内部で調整でき、柔軟性と機能が向上します。アプリケーションの観点からは、Ch と C におけるこれらの関数の違いにプログラマが気付くことはありません。

20.3.1 fprintf 出力関数ファミリの出力形式

このセクションでは、**fprintf()** 出力関数ファミリの形式について詳しく説明します。特に、 Ch で拡張された C の **fprintf()** の出力形式について取り上げます。基本原理は、その他の出力関数にも適用できます。Ch の関数 **fprintf**() の出力形式は次のとおりです。

```
int fprintf(FILE *stream, char *format, arg1, arg2, ...);
```

関数 **fprintf**() は、`format` が指す文字列の制御下で、引数 `stream` が参照するストリームへ出力を行い、出力された文字の数を返します。通常の文字と変換指定 (文字 ‘%’ で開始し変換文字で終了する) の 2 種類のオブジェクトが出力形式の文字列に含まれる場合、以下の C の規則が **fprintf**() 関数ファミリに適用されます。%の後に、以下の情報を順番に指定します。

– 変換指定の意味を変更する 0 個以上のフラグ (順不同)。
– (オプション) 最小フィールド幅。変換後の値に含まれる文字数がフィールド幅に満たない場合、既定では、フィールド幅まで左側に (または後述する左揃えフラグが指定されている場合は右側に) 空白が埋め込まれます。フィールド幅はアスタリスク(後述) または 10 進整数の形をとります。

– (オプション) **d**、**i**、**o**、**u**、**x**、および **X** 変換で表示される最小桁数、**a**、**A**、**e**、**E**、**f**、および **F** の変換で小数点以下に表示される桁数、**g** および **G** 変換での最大有効桁数、または **s** 変換で文字列から書き込まれる最大文字数などの数値を指定する精度。精度は、アスタリスク(後述) またはオプションの 10 進整数のいずれかの前にピリオド (.) を置く形で表されます。ピリオド

しか指定されていない場合、精度はゼロとして扱われます。上記以外の変換指定子で精度を指定した場合、動作は不定です。

– (オプション) 引数のサイズを指定する長さ修飾子。

– 適用する変換の種類を指定する変換指定子文字。

上記の説明にあるように、フィールド幅や精度は、アスタリスクで示すことができます。この場合、フィールド幅または精度は **int** 型の引数で指定します。フィールド幅や精度を指定する引数は、変換される引数 (もしあれば) の前に (この順番で) 置く必要があります。フィールド幅の引数を負の値で指定すると、正のフィールド幅の前に"-" (マイナス) フラグが付けられます。精度の引数が負の場合、精度が省略されていると見なされます。
各フラグ文字とその意味は次のとおりです。

- 変換結果をフィールド内に左揃えで表示します (このフラグを指定しない場合は右揃えになります)。

+ 符号付き変換の結果は、先頭が常に"+" 符号または"-"符号になります (このフラグを指定しない場合、負の値が変換される場合のみ先頭が符号になります)。

space 符号付き変換の最初の文字に符号が付いていない場合、または、符号付き変換の結果に文字がない場合は、結果の前に空白が付加されます。*space* フラグと**+**フラグの両方を指定すると、space フラグが無視されます。

結果は"代替形式"に変換されます。**o** 変換では、必要な場合にのみ精度を上げ、結果の最初の桁を強制的に 0 にします (値と精度の両方が 0 の場合は、0 が 1 つ出力されます)。**x** (または **X**) 変換では、結果がゼロ以外の場合、前に **0x** (または **0X**) が付加されます。**a**、**A**、**e**、**E**、**f**、**F**、**g**、および **G** 変換では、浮動小数点数の変換結果には、小数点以下の数字がない場合でも常に小数点文字が表示されます (通常、変換結果に小数点が表示されるのは、小数点以下の数字がある場合のみです)。**g** および **G** の変換では、末尾のゼロは結果からは削除されません。それ以外の変換では、動作は不定です。

0 **d**、**i**、**o**、**u**、**x**、**X**、**a**、**A**、**e**、**E**、**f**、**F**、**g**、および **G** 変換の場合、フィールド幅を埋めるために空白を埋め込むのではなくゼロを先頭に (符号または基数が指定されている場合はその後に続けて) 付けます。ただし、無限または NaN を変換する場合を除きます。**0** フラグと**-**フラグの両方を指定した場合、0 フラグが無視されます。**d**、**i**、**o**、**u**、**x**、および **X** 変換では、精度を指定した場合、**0** フラグが無視されます。その他の変換では、このフラグの動作は不定です。

長さ修飾子とそれぞれの意味は以下のとおりです。

hh 後続の変換指定子が **d**、**i**、**o**、**u**、**x**、または **X** の場合は、その指定子が **signed char** 型または **unsigned char** 型の引数に適用されることを指定します (引数は整数の上位変換に応じて上位変換されますが、値は **signed char** 型または **unsigned char** 型に上位変換してから出力する必要があります)。または、**n** の場合は、それが **signed char** 型の引数へのポインタに適用されることを指定します。

h 後続の変換指定子が **d**、**i**、**o**、**u**、**x**、または **X** の場合は、その指定子が **short int** 型または **unsigned short int** 型の引数に適用されることを指定します (引数は整数の上位変換に応じて上位変換されますが、値は **short int** 型または **unsigned short int** 型に上位変換してから出力する必要があります)。または、**n** の場合は、それが **short int** 型の引数へのポインタに適用されることを指定します。

l 後続の変換指定子が **d**、**i**、**o**、**u**、**x**、または **X** の場合は、その指定子が **long int** 型または **unsigned long int** 型の引数に適用されることを指定します。**n** の場合は **long int** 型の引数へのポインタに、**c** の場合は **wint_t** 型の引数に、**s** の場合は **wchar_t** 型の引数へのポインタにそれぞれ適用されることを指定します。または、後続の **a**、**A**、**e**、**E**、**f**、**F**、**g**、または **G** 変換指定子に影響を与えないことを指定します。

ll 後続の変換指定子が **d**、**i**、**o**、**u**、**x**、または **X** の場合は、その指定子が **long long int** 型または **unsigned long long int** 型の引数に適用されることを指定します。または、**n** の場合は、

j 後続の変換指定子が **d**、**i**、**o**、**u**、**x**、または **X** の場合は、その指定子が **intmax_t** 型または **uintmax_t** 型の引数に適用されることを指定します。または、**n** の場合は、**intmax_t** 型の引数へのポインタに適用されることを指定します。

z 後続の変換指定子が **d**、**i**、**o**、**u**、**x**、または **X** の場合は、その指定子が **size_t** 型の引数または対応する符号付き整数型引数に適用されることを指定します。または、**n** の場合は、それが **size_t** 型の引数に対応する符号付き整数型引数に適用されることを指定します。

t 後続の変換指定子が **d**、**i**、**o**、**u**、**x**、または **X** の場合は、その指定子が **ptrdiff_t** 型の引数または対応する符号なし整数型引数に適用されることを指定します。または、**n** の場合は、それが **ptrdiff_t** 型の引数へのポインタに適用されることを指定します。

L 後続の変換指定子が **a**、**A**、**e**、**E**、**f**、**F**、**g**、または **G** の場合は、その指定子が **long double** 型の引数に適用されることを指定します。長さ修飾子を上記の変換指定子以外の変換指定子と一緒に指定した場合、動作は不定です。変換指定子とそれぞれの意味は次のとおりです。

長さ修飾子を上記の変換指定子以外の変換指定子と一緒に指定した場合、動作は不定です。

変換指定子とそれぞれの意味は次のとおりです。

d,i **int** 型の引数を *[-]dddd* という形式の符号付き 10 進数に変換します。精度は表示する最小桁数を指定します。変換対象の値を少ない桁数で表現できる場合は、先頭にゼロが埋め込まれます。既定の精度は 1 です。精度 0 でゼロの値を変換した結果は、文字にはなりません。

o,u,x,X **unsigned int** 型の引数を *dddd* という形式の符号なし 8 進数 (**o**)、符号なし 10 進数 (**u**)、または符号なし 16 進数 (**x** または **X**) の表記に変換します。**x** 変換には **abcdef** の文字が、**X** 変換には **ABCDEF** の文字が使用されます。精度は表示する最小桁数を指定します。変換対象の値を少ない桁数で表現できる場合は、先頭にゼロが埋め込まれます。既定の精度は 1 です。精度 0 でゼロの値を変換した結果は、文字数ゼロです。

f,F 浮動小数点数を表す **double** 型引数を *[-]ddd.ddd* という形式の 10 進数表記に変換します。小数点以下の桁数は精度の指定に等しくなります。精度を指定しない場合、精度は 6 と見なされます。精度を 0 に指定して**#**フラグを指定しない場合、小数点文字は表示されません。小数点文字を表示する場合は、小数点の前に少なくとも 1 桁が表示されます。値は適切な桁数に丸められます。

無限を表す **double** 型引数は、*[-]***inf** または *[-]***infinity** のいずれかの形式で変換されます。どちらの形式になるかは実装依存です。NaN を表す **double** 型引数は、*[-]***nan** または *[-]***nan** *(n-char-sequence)* のいずれかの形式で変換されます。どちらの形式になるか、および n-char シーケンスの意味は、実装で依存です。F 変換指定子は、**inf**、**infinity**、**nan** の代わりに、それぞれ **INF**、**INFINITY**、**NAN** を生成します。

e,E 浮動小数点数を表す double 型引数を *[-]d.ddd* **e**±*dd* という形式に変換します。小数点の前に 1 桁 (引数がゼロ以外の場合はゼロ以外) が表され、小数点以下の桁数は精度の指定に等しくなります。精度を指定しない場合は、6 として扱われます。精度 0 を指定し、**#**フラグを指定しない場合、小数点は表示されません。値は、適切な桁数に丸められます。**E** 変換指定子は、指数を示す **e** の代わりに、**E** の付いた数字を出力します。指数には常に 2 桁以上が含まれ、指数を表すのに必要な分だけ桁が増やされます。値がゼロであるなら、指数もゼロです。
無限または NaN を表す **double** 型引数は、**f** または **F** の変換指定子の形式に変換されます。

g,G

浮動小数点数を表す **double** 型引数を、有効桁数を示す精度を付けて、f または **e**(または **G** 変換指定子の場合は、**F** または **E**) という形式に変換します。精度 0 の場合は、1 として扱われます。使用される形式は、変換される値によって決まります。変換結果の指数が-4 よりも小さいか、精度に等しいまたはそれよりも大きい場合にのみ、**e** (または **E**) が使用されます。末尾のゼロは、**#**フラグが指定されていない限り変換結果の小数部から削除されます。小数点文字が表示されるのは、小数点以下の桁が存在する場合のみです。
無限または NaN を表す double 型引数は、**f** または **F** の変換指定子の形式に変換されます。

a,A

浮動小数点数を表す **double** 型引数を、*[-]***0x***h.hhhh***p**±*d* という形式に変換します。小数点の前に 16 進数が 1 桁 (引数が正規化された浮動小数点数の場合はゼロ以外の値になり、それ以外の場合は未指定) が表され、小数点以下の桁数は精度の指定に等しくなります。精度が指定されておらず、**FLT_RADIX** が 2 の累乗である場合、精度は値を正確に表現できます。精度が指定されておらず、**FLT_RADIX** が 2 の累乗でない場合、精度は **double** 型の値を判別します。ただし、末尾のゼロは削除されます。精度に 0 を指定し、**#**フラグを指定しない場合、小数点は表示されません。**a** 変換には **abcdef** の文字が、A 変換には **ABCDEF** の文字が使用されます。**A** 変換指定子は、**x** および **p** ではなく、**X** および **P** を付けた数字を生成します。指数には常に 1 桁以上が含まれ、2 の 10 進指数を表現するのに必要な分だけ桁数が増やされます。値がゼロなら、指数もゼロです。
無限または NaN を表す double 型引数は、**f** または **F** の変換指定子の形式に変換されます。

c 長さ修飾子 **l** を指定しない場合、int 型引数は **unsigned char** 型に変換され、その結果の文字が書き出されます。長さ修飾子 **l** を指定した場合、**wint_t** 引数は、精度の指定を付けずに、**wchar_t** の 2 つの要素配列 (最初の要素には **lc** 変換指定への **wint_t** 引数が含まれ、2 番目の要素には null ワイド文字が含むまれる) の最初の要素を参照する引数を持つ ls 変換指定で変換された場合と同様に変更されます。

s 長さ修飾子 **l** を指定しない場合、引数は文字型配列の最初の要素へのポインタでなければなりません。配列からの文字は、終了 null 文字まで (null 文字は含まない) 書き出されます。精度を指定した場合は、指定以上の文字は書き出されません。精度を指定しない、または配列サイズより大きい精度を指定した場合は、null 文字まで配列に含める必要があります。
長さ修飾子 **l** を指定した場合、引数は **wchar_t** 型配列の最初の要素へのポインタでなければなりません。この配列からのワイド文字は、終端の null ワイド文字 (その null 文字を含む) までマルチバイト文字へ変換されます。各文字は **wcrtomb** への関数呼び出しで変換された場合と同様に変換され、**mbstate_t** オブジェクトで表される変換状態は、最初のワイド文字が変換される前にゼロに初期化されます。変換結果のマルチバイト文字は、終端の null 文字 (バイト) まで書き込まれます (その終端文字は含まない)。精度を指定しない場合、配列に null ワイド文字が含まれている必要があります。精度を指定した場合、指定した以上の文字数 (バイト数) は書き出されません (シフトシーケンスが含まれている場合は、シフトシーケンスが含まれます)。また、関数で配列の終わりより後にあるワイド文字にアクセスする必要がある場合、配列には、精度で指定されているマルチバイト文字のシーケンスと等しい長さの null ワイド文字が含まれている必要があります。マルチバイト文字が部分的に書き込まれることはありません。

p 引数は **void** へのポインタでなければなりません。ポインタの値は、実装によって定義された方法で、出力文字シーケンスに変換されます。

n この引数は符号付き整数へのポインタでなければなりません。その整数に書き込まれるのは、**printf** へのこの呼び出しでこれまでに出力ストリームに書き込まれた文字の数です。引数は変換されませんが、使用されます。変換指定になんらかのフラグ、フィールド幅、または精度が含まれている場合、動作は不定です。

% %文字を書き込みます。引数は変換されません。完全な形で変換指定する場合は、%%と指定する必要があります。

変換指定が正しくない場合、動作は不定です。引数の型と対応する変換指定が一致しない場合、動作は不定です。フィールド幅を指定しない場合、または指定したフィールド幅が小さい場合でも、フィールドが切り詰められることはありません。変換結果がフィールド幅より大きくなった場合は、変換結果が表示されるようにフィールド幅が広げられます。**a** 変換および **A** 変換では、**FLT_RADIX** が 2 の累乗である場合、値は精度に指定した 16 進浮動小数点数に正しく丸められます。**FLT_RADIX** が 2 の累乗でない場合、結果は 2 つの隣接する数字のいずれかになり、精度に指定した 16 進浮動小数点形式で表されます。ただし、誤差を表す符号は、現在の丸め方向に一致します。**e**、**E**、**f**、**F**、**g**、および **G** 変換では、10 進有効桁数が最大 **DECIMAL_DIG** である場合、結果は正しく丸められます。10 進有効桁数が **DECIMAL_DIG** を超え、ソースの値が **DECIMAL_DIG** の桁数で正確に表現できる場合、結果は末尾にゼロを含む形で正確に表現されます。そうでない場合、ソースの値は L < U の関係を持つ隣接する 2 つの 10 進数文字列 (どちらも **DECIMAL_DIG** 桁の有効桁数を持つ) で結合されま

す。その結果の 10 進数文字列 D の値は、$L \le D \le U$ を満たします。ただし、誤差を表す符号は、現在の丸め方向に一致します。次のコマンドは、異なる形式による関数 **printf** の使用例です。

```
> double d = 123.45678
> float f = 123.45678
> char *str = "123456789"
> printf("d = %1.3f", d)
d = 123.457
> printf("f = %5.10f", f)
f = 123.4567794800
> printf("%-15s", str)
123456789
> printf("%15s", str)
      123456789
```

C 規格で指定されている制御文字以外に、 Ch では、実数をバイナリ形式で出力するための変換文字として、さらに ‘b’ が用意されています。シンボル%と文字”b”で囲んだ整数で、出力するビット数をビット 0 からのビット数で指定します。シンボル%と文字 ‘b’ の間に整数を指定しない場合、既定の出力形式は、先頭にゼロを埋め込まない int 型データ、32 ビットの float 型データ、および 64 ビットの double 型データになります。このバイナリ形式は、メタ数値のビットパターンを調べるのに非常に便利です。次に例を示します。

```
> int i = 5
> float f = 1.234
> printf("binary of i = %b, f = %b", i, f)
binary of i = 101, f = 00111111100111011111001110110110
```

Ch では、関数 **printf**() 内に形式の文字列が指定されていない場合、または通常の文字しか含まない場合、以降の数値の定数または変数は、既定のプリセット形式で出力されます。int 型、float 型、および double 型の既定の出力形式は、それぞれ%d、%.2f、および%.4lf です。次に例を示します。

```
> float f = 1.234
> printf(f)
1.23
```

fprintf() 関数ファミリの既定の出力形式については、セクション 20.4で詳しく説明します。

20.3.2 fscanf 入力関数ファミリの入力形式

このセクションでは、 **scanf()**、**fscanf()**、**sscanf()** などの **fscanf()** 入力関数ファミリの入力形式について詳しく説明します。特に、Ch で拡張された C の **fscanf()** の入力形式について説明します。Ch の関数 **fscanf**() の入力形式は次のとおりです。

```
int fscanf(FILE *stream, char *format, arg1, arg2, ...);
```

関数 **fscanf**() は、引数 `format` が指す文字列の制御下で、引数 `stream` が指すストリームから入力を読み込み、成功すると入力項目の数を返します。入力形式は、初期シフト状態の開始と終了を持つ、マルチバイト文字のシーケンスです。入力形式は、1 つ以上の空白文字、通常のマルチバイト文字 (%でも空白文字でもない)、または変換指定など、ゼロ個以上のディレクティブで構成されます。各変換指定は、文字%で開始されます。
%の後に、以下の情報を順番に指定します。

– (オプション) 代入抑止文字 “*”
– (オプション) フィールドの最大幅 (文字数) を指定するゼロ以外の小数整数。

– (オプション) 受信オブジェクトのサイズを指定する長さ修飾子。

– 適用する変換の種類を指定する変換指定子の文字。

fscanf 関数は入力形式の各ディレクティブを順番に実行します。この後で詳しく説明するように、ディレクティブにエラーがある場合、関数は終了します。エラーは、入力エラー (コーディングエラーの発生または使用できない入力文字が原因)、またはマッチングエラー (不適切な入力が原因) です。

空白文字で構成されるディレクティブは、読み込まれないで残っている最初の空白以外の文字まで、または読み込む文字がなくなるまで、入力を読み込んで実行されます。

通常のマルチバイト文字で構成されるディレクティブは、ストリームの次の文字を読み込むことによって実行されます。これらの文字のいずれかがディレクティブを構成する文字と異なる場合、ディレクティブの実行は失敗し、異なる文字以降の文字は読み込まれずに残ります。

変換指定のディレクティブは、以下の各指定子の説明にあるように、一連のマッチング入力シーケンスを定義します。変換指定は次のステップで実行されます。

[、**c**、**n** などの変換指定子が変換指定に含まれている場合を除き、入力した空白文字 (**isspace** 関数で指定される) はスキップされます。

n 変換指定子が変換指定に含まれている場合を除き、入力項目はストリームから読み込まれます。入力項目は、入力された最も長い文字シーケンス (指定したフィールド幅を超えず、マッチング入力シーケンスまたはその接頭語である) として定義されます。入力項目の後の最初の文字 (もしあれば) は読み込まれないで残ります。入力項目の長さが 0 の場合、ディレクティブの実行は失敗します。この状態はマッチングエラーです。ただし、ファイルの終わり (EOF)、エンコードディングエラー、またはストリームからの読み込みエラー (この場合は入力エラー) の場合を除きます。

変換指定子が%の場合を除き、入力項目 (%**n** ディレクティブの場合は、入力文字数) は、変換指定子に適した型に変換されます。入力項目がマッチングシーケンスではない場合、ディレクティブの実行は失敗します。この状態はマッチングエラーです。代入抑止が “*”で表された場合を除き、変換結果は、まだ変換結果を受け取っていない引数 *format* の後で最初に出現する引数が参照するオブジェクトに出力されます。このオブジェクトが適切な型でない場合、または変換結果をこのオブジェクトで

表すことができない場合、動作は不定です。

長さ修飾子とそれぞれの意味は以下のとおりです。

hh 後続の変換指定子が **d**、**i**、**o**、**u**、**x**、**X**、または **n** の場合、その指定子が **signed char** または **unsigned char** へのポインタ型を持つ引数に適用されることを指定します。

h 後続の変換指定子が **d**、**i**、**o**、**u**、**x**、**X**、または **n** の場合、その指定子が **short int** または **unsigned short int** へのポインタ型を持つ引数に適用されることを指定します。

l 後続の変換指定子が **d**、**i**、**o**、**u**、**x**、**X**、または **n** の場合、その指定子が **long int** または **unsigned long int** へのポインタ型を持つ引数に適用されることを指定します。**a**、**A**、**e**、**E**、**f**、**F**、**g**、または、**G** の場合は、 **double** へのポインタ型を持つ引数に、**c**、**s**、または **[**の場合は、**wchar_t** へのポインタ型を持つ引数に適用されることを指定します。

ll 後続の変換指定子が **d**、**i**、**o**、**u**、**x**、**X**、または **n** の場合、その指定子が **long long int** または **unsigned long long int** へのポインタ型を持つ引数に適用されることを指定します。

j 後続の変換指定子が **d**、**i**、**o**、**u**、**x**、**X**、または **n** の場合、その指定子が **intmax_t** または **uintmax_t** へのポインタ型を持つ引数に適用されることを指定します。

z 後続の変換指定子が **d**、**i**、**o**、**u**、**x**、**X**、または **n** の場合、その指定子が **size_t** または対応する符号付き整数型へのポインタ型を持つ引数に適用されることを指定します。

t 後続の変換指定子が **d**、**i**、**o**、**u**、**x**、**X**、または **n** の場合、その指定子が **ptrdiff_t** または対応する符号なし整数型へのポインタ型を持つ引数に適用されることを指定します。

L 後続の変換指定子が **a**、**A**、**e**、**E**、**f**、**F**、**g**、または **G** の場合、その指定子が **long double** へのポインタ型を含む引数に適用されることを指定します。type pointer to **long double**.

長さ修飾子を上記の変換指定子以外の変換指定子と一緒に指定した場合、動作は不定です。

変換指定子とそれぞれの意味は次のとおりです。

d 10 進数の整数 (符号は任意) と一致します。入力形式は、引数 **base** に値 10 を指定した関数 **strtol** の変換対象の文字列と同じ文字列である必要があります。対応する引数は、符号付き整数へのポインタ型でなければなりません。

i 整数 (符号は任意) と一致します。入力形式は、引数 **base** に値 0 を指定した関数 **strtol** の変換対象の文字列と同じ文字列である必要があります。対応する引数は、符号付き整数へのポインタ型でなければなりません。

o 8 進数の整数 (符号は任意) と一致します。入力形式は、引数 **base** に値 8 を指定した関数 **strtoul** の変換対象の文字列と同じ文字列である必要があります。対応する引数は、符号なし整数へのポインタ型でなければなりません。

u 10 進数の整数 (符号は任意) と一致します。入力形式は、引数 **base** に値 10 を指定した関数 **strtoul** の変換対象の文字列と同じ文字列である必要があります。対応する引数は、符号なし整数へのポインタ型でなければなりません。

x 16 進数の整数 (符号は任意) と一致します。入力形式は、引数 **base** に値 16 を指定した関数 **strtoul** の変換対象の文字列と同じ文字列である必要があります。対応する引数は、符号なし整数へのポインタ型でなければなりません。

a,e,f,g 浮動小数点数、無限、または NaN と一致します (符号は任意)。入力形式は、関数 **strtod** の変換対象の文字列と同じ文字列である必要があります。対応する引数は、float へのポインタ型でなければなりません。

c フィールド幅に指定された数 (ディレクティブにフィールド幅の指定がない場合は 1) の文字列と一致します。
長さ修飾子 **l** を指定しない場合、対応する引数は、その文字列を受け入れるだけの十分な大きさを持つ文字配列の先頭要素へのポインタでなければなりません。null 文字は追加されません。
長さ修飾子 **l** を指定した場合、入力は、初期シフト状態で始まるマルチバイト文字でなければなりません。文字列内の各マルチバイト文字は、**mbrtowc** への関数呼び出しを行った場合と同様にワイド文字へ変換され、変換状態は、最初のマルチバイト文字が変換される前にゼロに初期化された **mbstate_t** オブジェクトによって記述されます。対応する引数は、結果のワイド文字の文字列を受け入れるだけの十分な大きさを持つ **wchar_t** 配列の先頭要素へのポインタでなければなりません。null ワイド文字は追加されません。

s 空白文字でない文字の文字列に一致します。
長さ修飾子 **l** を指定しない場合、対応する引数は、文字列と終端 null 文字 (自動的に含まれる) を受け入れるだけの十分な大きさを持つ文字配列の先頭要素へのポインタでなければなりません。

長さ修飾子 **l** を指定した場合、入力は、初期シフト状態で始まるマルチバイト文字の文字列でなければなりません。各マルチバイト文字は、**mbrtowc** への関数呼び出しを行った場合と同じ様に、ワイド文字へ変換され、変換状態は、最初のマルチバイト文字が変換される前にゼロに初期化された **mbstate_t** オブジェクトによって記述されます。対応する引数は、文字列と終端 null 文字 (自動的に含まれる) を受け入れるだけの十分な大きさを持つ **wchar_t** 配列の先頭要素へのポインタでなければなりません。

[期待される文字セット (scanset) からの空でない文字列と一致します。
長さ修飾子 **l** を指定しない場合、対応する引数は、文字列と終端 null 文字 (自動的に含まれる) を受け入れるだけの十分な大きさを持つ文字配列の先頭要素へのポインタでなければなりません。
長さ修飾子 l を指定した場合、入力は、初期シフト状態で始まるマルチバイト文字の文字列でなければなりません。各マルチバイト文字は、**mbrtowc** への関数呼び出しを行った場合と同じ様に、ワイド文字へ変換され、変換状態は、最初のマルチバイト文字が変換される前にゼロに初期化された **mbstate_t** オブジェクトによって記述されます。対応する引数は、文字列と終端 null 文字 (自動的に含まれる) を受け入れるだけの十分な大きさを持つ **wchar_t** 配列の先頭要素

へのポインタでなければなりません。

変換指定子には、**format** 文字列内の後続のすべての文字を、対応する右角かっこ**]** まで含みます (角かっこも含む)。角かっこに囲まれた文字 (*scanlist*) は scanset を構成します。ただし、左角かっこの後にカレット (^) がある場合は、カレットと右角かっこ挟まれた scanlist に入っていないすべての文字が scanset を構成します。変換指定子が **[]** または **[^]** で始まる場合、右角かっこは scanlist 内に含まれ、この後に出現する右角かっこが、指定の終わりを示す対応する右角かっこになります。2 番目の右角かっこがない場合は、最初の右角かっこが、変換指定の終わりを示す右かっこと見なされます。scanlist に含まれる "**-**"の文字が 1 文字目ではなく、2 文字目 ("^" を 1 文字目として) でもなく、かつ末尾の文字でない場合、動作は実装依存です。

p 実装依存の文字列に一致します。これにより、関数 **fprintf** の**%p** 変換によって出力される文字列と同じになります。対応する引数は、 **void** へのポインタのポインタでなければなりません。入力項目は、実装依存の方法でポインタ値に変換されます。入力項目が、同じプログラムの実行中で以前に変換された値の場合、結果となるポインタはその値と同じでなければなりません。そうでない場合、**%p** 変換の動作は不定です。

n 入力は使用されません。対応する引数は、**fscanf** 関数へのこの呼び出しでこれまでに入力ストリームから読み取った文字の数を示す、符号付き整数へのポインタでなければなりません。**%n** ディレクティブを実行しても、**fscanf** 関数の実行完了時に返される代入回数は増えません。引数は変換されませんが、使用されます。変換指定に代入抑止文字またはフィールド幅が含まれている場合、動作は不定です。

% 単一の%文字と一致します。変換または代入は行われません。完全な形で変換指定する場合は、%%と指定する必要があります。

変換指定が正しくない場合、動作は不定です。**A**、**E**、**F**、**G**、および **X** の変換指定子も有効です。それぞれ、**a**、**e**、**f**、**g**、および **x** と同じ動作をします。入力中にファイルの終わりに達すると、変換は終了します。現在のディレクティブに一致する文字 (許可されている場合は先行の空白を除く) が読み込まれる前にファイルの終わりに達した場合、現在のディレクティブの実行は入力エラーで終了します。それ以外の場合、現在のディレクティブの実行がマッチングエラーで終了しなければ、以降のディレクティブ (**%n** を除く) の実行が入力エラーで終了します。ディレクティブと一致しない場合、末尾の空白 (改行文字も含む) は読み込まれずに残ります。リテラルの一致と抑止された代入が成功したかどうかを直接確認するには、**%n** ディレクティブを使用する以外にありません。入力文字が競合して変換が終了した場合、入力ストリーム内の競合する入力文字は読み込まれないままになります。さまざまな入力形式で関数 **scanf()** を使用したコマンド例を次に示します。

```
> int i
> float x
> char name[50]
> scanf("%2d%f%*d %[01234567890]", &i, &x, name)
56789 0123 34a72
> i
56
```

```
> x
789.00
> name
34
```

Ch では、**fscanf()** 関数ファミリの入力形式の文字列がない場合、int 型では%d、float 型では設定された既定の入力形式が使用されます。次に例を示します。

```
> float f;
> scanf(&f); // <==> scanf("%f", &f);
10
> f
10.00
```

fscanf() の既定の入力形式については、セクション 20.4で詳しく説明します。

20.4 既定の入出力形式

20.4.1 fprintf 出力関数ファミリの既定の形式

Ch では、出力関数の形式を指定しない場合、既定の形式が使用されます。表 20.2に、 **printf()**、**fprintf()**、**vprintf()**、**vfprintf()**、および **vsprintf()** の関数で扱われるそれぞれのデータ型の既定の出力形式を示します。

表 20.2: **fprintf** 出力関数ファミリの既定の形式

データ型	形式
char	"%c"
unsigned char	"%c"
short	"%hd"
unsigned short	"%hu"
int	"%d"
unsigned int	"%u"
long	"%ld"
unsigned long	"%lu"
long long	"%lld"
unsigned long long	"%llu"
float	"%.2f"
double	"%.4lf"
char *	"%s"
unsigned char *	"%s"
string_t	"%s"
pointer type	"%p"

次に例を示します。

```
> int i = 8
> float f = 8.0
> double d = 8.0
> printf(i) /* <==> printf("%d", i); */
8
> printf(f) /* <==> printf("%.2f", f); */
8.00
> printf(d) /* <==> printf("%.4f", d); */
8.0000
> f
8.00
> d
8.0000
```

char へのポインタと unsigned char へのポインタの既定の形式は`%s` です。`%s` は出力が文字列であることを示します。このようなポインタが指すアドレスを表示するには、まず、値を void へのポインタとして型指定します。そうすると、次のように出力されます。

```
> char *p = "abc"
> p
abc
> (void*)p
004a3418
```

20.4.2 fscanf 入力関数ファミリの既定の形式

同様に、入力引数を指定しない場合のために、既定の入力関数形式があります。表 20.3に、**scanf()**、**fscanf()**、 および **sscanf()** の関数で扱われるそれぞれのデータ型の既定の入力形式を示します。

表 20.3: **fscanf** 入力関数ファミリの既定の形式

データ型	形式
char	`"%c"`
unsigned char	`"%c"`
short	`"%hd"`
unsigned short	`"%hu"`
int	`"%d"`
unsigned int	`"%u"`
long	`"%ld"`
unsigned long	`"%lu"`
long long	`"%lld"`
unsigned long long	`"%llu"`
float	`"%f"`
double	`"%lf"`
char *	`"%s"`
unsigned char *	`"%s"`
string_t	`"%s"`
pointer type	`"%p"`

次に例を示します。

```
> int i;
> float f;
> scanf(&i); /* <==> scanf("%d", &i); */
10
> i
10
> scanf(&f); /* <==> scanf("%f", &f); */
10
> f
10.00
```

20.4.3 cout、cin、cerr、および endl を使用する入出力

Ch には、プログラムの入出力用に C++スタイルの 3 つのストリームがあります。通常のキーボードに接続される入力ストリーム **cin**、通常のコンピュータ画面に接続される入力ストリーム **cout**、および通常のコンピュータ画面に接続されるエラーストリーム **cerr** の 3 つのストリームが標準で用意されています。これらの 3 つのストリームはすべて、既定のデバイス以外のデバイスに割り当てることが可能です。

また、Ch では、ストリーム **cout** への出力を実行するストリーム挿入演算子 “<<”、ストリーム **cin** からの入力を実行するストリーム抽出演算子 “>>”、および改行文字を発行し、出力バッファを解放するストリームマニピュレータ endl が用意されています。演算子 “<<”および “>>”は左から右への向きに結合して、カスケード式に使用できます。次に例を示します。

```
> int i, j
> cin >> i          // <==> scanf(&i)
10
> cout << i         // <==> printf(i)
10
> cerr << i
10
> cin >> i >> j     // <==> scanf(&i, &j)
20 30
> cout << i << j    // <==> printf(i, j)
2030
```

ストリーム **cout**、**cin**、および **cerr** は、C++のクラス **istream** と **ostream** のオブジェクトです。Ch では、それぞれ、関数 **printf()**、**scanf()**、および **fprintf(stderr,)** のショートカットです。セクション 20.4で説明されている既定の入力出形式が使用されます。これらは、システム全体のスタートアップファイル **chrc** で別名として定義され、ヘッダーファイル **iostream.h** ではマクロとして定義されます。したがって、 Ch プログラムで **cin**、**cout**、および **cerr** のストリームを使用するには、次に示すようにヘッダーファイル **iostream.h** を組み込む必要があります。

```
#include <iostream.h>  // for cout/cin/cerr/endl
int main() {
  int i;

  cout << "Type a number :" << endl;
  cin >> i;
  cout << "The input number is " << i << endl;
}
```

ストリームマニピュレータ **endl** は、プログラムモードとコマンドモードの両方で使用できます。

新しい C++規格と互換性を持たせるため、入出力ストリームでの名前空間用の **using** ディレクティブは以下の形式でサポートされています。

```
using std::cout;
using std::cin;
using std::cerr;
using std::endl;
using std::ends;
```

または

```
using namespace std;
```

たとえば、上記のプログラムを次のように書き換えて、C++規格に準拠させることができます。

```
#include <iostream.h>  // for cout/cin/cerr/endl
using std::cout;
using std::cin;
using std::endl;
/* or using namespace std; */
int main() {
  int i;

  cout << "Type a number :" << endl;
  cin >> i;
  cout << "The input number is " << i << endl;
}
```

20.5 メタ数値の入出力

メタ数値 Inf と NaN は、入出力関数では通常の数値として扱われます。 これらの数値の既定のデータ型は float です。次の Ch プログラムの例は、b 形式とメタ数値を入出力関数 **printf**() と **scanf**() で処理する方法を示します。

```
float fInf, fNaN;
double dInf, dNaN;
printf("Please type ’Inf NaN Inf NaN’ \n");
scanf(&fInf, &fNaN, &dInf, &dNaN);
printf("The float   Inf = %f\n", fInf);
printf("The float  -Inf = ", -fInf, "\n");
printf("The float   NaN = %f\n", fNaN);
printf("The float   Inf = %b \n", fInf);
printf("The float  -Inf = %b \n", -fInf);
printf("The float   NaN = %b \n", fNaN);
printf("The double  Inf = %lf\n", dInf);
printf("The double -Inf = ", -dInf, "\n");
printf("The double  NaN = %lf\n", dNaN);
printf("The double  Inf = %b \n", dInf);
printf("The double -Inf = %b \n", -dInf);
printf("The double  NaN = %b \n", dNaN);
printf("The int     2   = %b \n",  2);
printf("The int     2   = %32b \n",  2);
printf("The int    -2   = %b \n", -2);
printf("The float   0.0 = %b \n", 0.0);
```

```
printf("The float  -0.0 = %b \n", -0.0);
printf("The float   1.0 = %b \n",  1.0);
printf("The float  -1.0 = %b \n", -1.0);
printf("The float   2.0 = %b \n",  2.0);
printf("The float  -2.0 = %b \n", -2.0);
```

プログラムの先頭の 2 行で、fInf および fNaN という 2 つの float 型変数と、dInf および dNaN という 2 つの double 型変数を宣言しています。関数 **scanf()** は、宣言した変数に Inf と NaN を標準の入力デバイス (この例では、キーボード端末) から取得します。これらのメタ数値は、float の場合は%.2f、double の場合は%0.4lf という既定の形式で出力されます。また、これらの数値は、バイナリ形式%b を使用しても出力されます。比較のため、±2 の整数と ±0.0、±1.0、±2.0 の浮動小数点数のメモリ記憶域を出力してみます。上記のプログラムを対話的に実行した結果を以下に示します。

```
Please type 'Inf NaN Inf NaN'
```

Inf NaN Inf NaN

```
The float   Inf = Inf
The float  -Inf = -Inf
The float   NaN = NaN
The float   Inf = 01111111100000000000000000000000
The float  -Inf = 11111111100000000000000000000000
The float   NaN = 01111111111111111111111111111111
The double  Inf = Inf
The double -Inf = -Inf
The double  NaN = NaN
The double  Inf = 01111111111100000000000000000000\
                  00000000000000000000000000000000
The double -Inf = 11111111111100000000000000000000\
                  00000000000000000000000000000000
The double  NaN = 01111111111111111111111111111111\
                  11111111111111111111111111111111
The int     2   = 10
The int     2   = 00000000000000000000000000000010
The int    -2   = 11111111111111111111111111111110
The float   0.0 = 00000000000000000000000000000000
The float  -0.0 = 10000000000000000000000000000000
The float   1.0 = 00111111100000000000000000000000
The float  -1.0 = 10111111100000000000000000000000
The float   2.0 = 01000000000000000000000000000000
The float  -2.0 = 11000000000000000000000000000000
```

ここで、斜体になっている 2 行目は入力を示し、プログラムの残りの行は出力を示します。これからわかることは、メタ数値 Inf、−Inf、および NaN に関して、ユーザーにとっては float 型と double 型

の間に違いはないということです。メモリ内のこれらの数値のビットマッピングが、セクション 6.1で説明されているデータ表現と一致することは簡単に確かめることができます。

20.6 集合体データ型の入出力形式

Ch では、複素数、計算配列、構造体、クラス、共用体などの集合体データ型の入力を要素ごとに処理する必要があります。次に例を示します。

```
> array int  A[2]
> scanf("%d", &A[0])
10
> A
10 0
>
```

Ch では、**fprintf()** や **printf()** などの出力関数ファミリを使用して、集合体データ型の変数および定数のすべての要素を出力することができます。複素数と計算配列では、出力形式の指定子は各要素に適用されます。構造体、クラス、および共用体では、既定の出力形式が各メンバに対して使用されます。次に例を示します。

```
> printf("%.2f", complex(1.0, 2.0))
complex(1.00,2.00)
> array int  A[2][3], B[2][2]  // array
> A[0][0] = 1; B[1][1] = 6
1
> printf("A = \n%d\nB = \n%d\n", A, B);
A =
1 0 0
0 0 0

B =
0 0
0 6

> struct tag1{int i; float f;} s  //struct
> s.i = 10
10
> printf(s)
.i = 10
.f = 0.00
>
```

20.7　fprintf による逐次出力ブロック

関数 **fprintf** の機能を使用して、逐次出力ブロックを作成できます。逐次出力ブロックの構文は次のとおりです。

fprintf *stream* << *TERMINATOR*
...
TERMINATOR

または

fprintf *stream* << "*TERMINATOR*"
...
TERMINATOR

ここで、*stream* には有効なファイルストリームを指定します。ターミネータ *TERMINATOR* には、プログラム内でキーワードや変数名として使用されていない有効な識別子を指定します。ターミネータとして "END"などのマクロ名を使用できます。この場合、マクロは展開されずに逐次処理されます。識別子にはすべて大文字を使用することが推奨されます。**fprintf** を使用する逐次ブロック出力には、以下の制限があります。

- 空白とコメントは最初のターミネータの後に配置できる。
- 空白は 2 番目のターミネータの前に配置できる。
- 2 番目のターミネータは改行文字で終了しなければならない。2 番目のターミネータの後には文字を配置できない (空白も不可)。
- 最初の行と最終行の間にあるすべての文字 (空白文字およびコメントを含む) が逐次処理される。
- 最初のターミネータは二重引用符で囲むことができるが、2 番目のターミネータは囲むことができない。最初のターミネータを二重引用符で囲むと、囲みブロック内のドル記号 '$' は逐次的に扱われます。二重引用符で囲まれていない場合、変数または式の代用としてドル記号 '$' を 1 つ使用します。変数の代わりに、

  $*var* と ${*var*}

  という 2 つの構文を使用できます。展開される変数名またはシンボルは中かっこで囲むことができます。この中かっこは省略もすることも可能ですが、挿入すると変数が名前の一部として解釈され、その直後の文字から展開されるのを防ぐことができます。

  変数置換における変数には、事前に定義された識別子、ユーザー定義の変数 (文字列、char へのポインタ、整数、浮動小数点、複素数型などのデータ型)、環境変数、または定義されていないシンボルを指定できます。変数置換では、 Ch シェルはスコープ規則に従ってまず Ch の名前空間で変数名を検索します。変数を定義していない場合は、現在のプロセスの環境変数を検索します。指定した名前の変数が Ch の名前空間にも環境空間にも見つからない場合、変数置換は行われず、変数は無視されます。

- 形式

```
$(expression)
```

  で式の置換を行うと、有効な Ch 式をその結果で置換できます。*expression* には、文字列型、char へのポインタ型、整数型、浮動小数点型、または複素数型などのデータ型の式を指定する必要があります。

- 変数または式の置換は、ドル記号 '$' の前に円記号 '\' を置くことで抑止できます。後続に空白、タブ、または行末がある場合、'$' はそのまま渡されます。

- 変数置換または式置換された値は、そのデータ型の既定の形式制御文字列を使用して出力されます。

たとえば、プログラミングステートメント

```
#include <stdio.h>
int sum = 2
fprintf stdout << END /* This is a comment */
  /* this is verbatim output */
  sum = \$$sum
  sum + 1 = \$$(sum+1)
 END
```

で構成されるプログラム `verbat.ch` の場合、その実行結果は次のようになります。

```
> verbat.ch
    /* this is verbatim output */
    sum = $2
    sum + 1 = $3
>
```

コマンド

```
sum = \$$sum
```

の場合、エスケープ文字 '\' によってドル記号として '`$`' が 1 つ出力され、シンボル`$sum` は `sum` の値 (たとえば、2 など) で置き換えられます。次のようなコマンドがあるとします。

```
sum + 1 = \$$(sum+1)
```

このコマンドでシンボル`$(sum+1)` は式置換を示します。つまり、式 `sum+1` の結果 (たとえば 3 など) で置き換えられます。最初の終了文字 `END` の後のコメントおよび 2 番目の `END` の前の空白は無視されます。ただし、ブロック内のコメントは逐次出力されます。

既定で、double 型の変数は小数点以下 4 桁まで出力されますが、float 型の変数の出力は小数点以下 2 桁までです。double 型の変数を出力する場合、その値が float 型で表現可能な範囲の中にあれば、出

力する前に float 型に型指定することができます。たとえば、$((float)d) は小数点以下 2 桁で出力することができ、$((int)d) は整数部のみです。

CGI プログラムでは、多くの場合、HTML コードのブロックを標準の出力ストリームとして送信することが必要になります。たとえば、プログラム 20.1は次のコードを生成します。

```
Content-Type: text/html

<HTML>
<HEAD>
<Title> Hello, world </Title>
</Head><BODY>
<h4> Hello, world </h4>
</BODY>
</HTML>
```

```
/* File: genereatehtml.c */
#include <stdio.h>
int main() {
    char hello[] = "Hello, world";

    printf("Content-Type: text/html\n\n");
    printf("<HTML>\n");
    printf("<HEAD>\n");
    printf("<Title> Hello, world </Title>\n");
    printf("</Head>");
    printf("<BODY>\n");
    printf("<h4> %s </h4>\n", hello);
    printf("</BODY>\n");
    printf("</HTML>\n");
    return 0;
}
```

プログラム 20.1: HTML ファイルの生成

ここで、“`Hello, world`”は Web ブラウザに表示される文字です。HTTP プロトコルに従って、次の行

```
Content-Type: text/html
```

は空白なしで開始する必要があり、その後には空白のない空行のみを続ける必要があります。逐次出力機能を使用して、Ch で記述された上記の CGI プログラムは、プログラム 20.2にように簡単に書き換えることができます。`hello` の値は、逐次出力ブロック内のドル記号 “$”を使用して取得されることに注意してください。

```
#!/bin/ch
/* File: genereatehtml.ch */
#include <stdio.h>
int main() {
    char hello[] = "Hello, world";

    printf("Content-Type: text/html\n\n");
    fprintf stdout << ENDPRINT
        <HTML>
        <HEAD>
        <Title> Hello, world </Title>
        </Head>
        <BODY>
        <h4> $hello </h4>
        </BODY>
        </HTML>
    ENDPRINT
    return 0;
}
```

プログラム 20.2: fprintf を使用したブロック出力

もう 1 つの例として、次の関数 `sendApplet()` は C プログラムを生成します。

```
void sendApplet(char *x, char *y, char *expr) {
    fprintf(stdout, "#include<stdio.h>\n");
    fprintf(stdout, "int main() {\n");
    fprintf(stdout, "  double x = %s;\n", x);
    fprintf(stdout, "  double y = %s;\n", y);
    fprintf(stdout, "  printf(\"x = %%f, \", x);\n");
    fprintf(stdout, "  printf(\"y = %%f \\n\", y);\n");
    fprintf(stdout, "  printf(\"%s = %%f\\n\", %s);\n", expr, expr);
    fprintf(stdout, "}\n");
}
```

この関数 `sendApplet()` は、Ch で次のように書き直すことができます。

```
void sendApplet(char *x, char *y, char *expr) {
    fprintf stdout << ENDFILE
      #include<stdio.h>
      int main() {
        double x = $x;
        double y = $y;
        printf("x = %f", x);
        printf("y = %f\n", y);
        printf("$expr = %f\n", $expr);
      }
```

```
    ENDFILE
}
```

ここで、変数 x、y、および $expr$ の値は、演算子$によって取得されます。

20.8 ファイル操作

20.8.1 ファイルを開く/閉じる

ファイルは、Ch では、入出力ストリームとして最もよく使用するオブジェクトです。ヘッダーファイル **stdio.h** で定義されているデータ型 **FILE** には、ストリームの情報が保持されます。**fopen()** などのいくつかの関数を呼び出して作成した **FILE** *型のオブジェクトを使用し、**fscanf()** などの他のファイル操作関数でファイルにアクセスできます。関数 **fopen()** は、ファイルを開くために一般的に使用される関数です。プロトタイプは次のとおりです。

```
FILE *fopen(const char *filename, const char *mode);
```

成功すると、ストリームを制御するオブジェクトへのポインタを返します。ファイルを開く操作に失敗すると、**NULL** を返します。ファイルを開いてストリームに関連付けるには、そのファイルの名前を先頭の引数 **filename** に指定します。mode という別の引数でもファイルを開く目的を指定できます。表 20.4に **fopen()** の有効値を示します。

表 20.4: 関数 **fopen()** のファイルを開くモード

モード	意味
r	テキストファイルを開いて読み込みます。
w	長さゼロに切り捨てるか、またはテキストファイルを作成して書き込みます。
a	追加。テキストファイルを開くか、または作成してファイルの終わりに書き込みます。
rb	バイナリファイルを開いて読み込みます。
wb	長さゼロに切り捨てるか、またはバイナリファイルを作成して書き込みます。
ab	追加。バイナリファイルを開くか、または作成してファイルの終わりに書き込みます。
r+	テキストファイルを開いて更新 (読み込み/書き込み) します
w+	長さゼロに切り捨てるか、またはテキストファイルを作成して更新します
a+	追加。テキストファイルを開くか、または作成して更新 (ファイルの終わりに書き込み) します。
r+b または **rb+**	バイナリファイルを開いて更新 (読み込み/書き込み) します。
w+b または **wb+**	長さゼロに切り捨てるか、またはバイナリファイルを作成して更新します。
a+b または **ab+**	追加。バイナリファイルを開くか、または作成して更新 (ファイルの終わりに書き込み) します。

引数 mode の先頭に読み込みモードを示す '**r**' の文字が付いているファイルを開くと、ファイルは読み取り専用で開かれます。引数 mode の先頭に書き込みモードを示す '**w**' の文字が付ているファイルを開くと、ファイルは書き込み専用で開かれます。引数 *mode* の先頭に追加書き込みモードを示す '**a**' の文字が付いているファイルを開くと、以降のファイルへのすべての書き込みは、強制的に現在のファイルの終わりに書き込まれます。

上記のリストにある引数 *mode* の値の 2 番目または 3 番目の文字として更新モードを示す '**+**' が使われているファイルを開くと、入力、出力のどちらの操作も関連付けられたストリーム上で実行されます。ただし、関数 **fflush()** またはファイル位置決め関数 (**fseek**、**fsetpos**、または **rewind**) への呼び出しが介在しない場合、 出力直後に入力操作を行ってはなりません。また、入力操作がファイルの終わりに達した場合を除き、ファイル位置決め関数への呼び出しが介在しない場合、入力直後に出力操作を行ってはなりません。代わりに、一部のプラットフォームでは、テキストファイルを更新モードで開く (または作成する) ことで、バイナリストリームを開く (または作成する) ことができます。

ファイルが開かれると、対話型デバイスを参照しないことが確認できた場合に限り、ストリームはフルバッファリングされます。ストリームのエラーインジケータとファイルの終わりのインジケータがクリアされます。

開いてストリームに関連付けたすべてのファイルをプログラムが終了する前に閉じる必要があります。**fclose()** は、ファイルを閉じるためによく使用する関数です。プロトタイプは次のとおりです。

```
int fclose(FILE *stream);
```

ストリームが正常に閉じた場合、関数 **fclose()** はゼロを返します。エラーが検出された場合は、マクロ **EOF** の値が返されます。関数 **fclose** は、引数 *stream* が指すストリームを解放し、ストリームに関連付けられているファイルを閉じます。ストリームに書き込まれないバッファリングデータは、ホスト環境へ配信されてファイルに書き込まれ、バッファリングデータは破棄されます。ストリームとファイルの関連付けが解除されます。関連付けられたバッファが自動的に割り当てられていた場合は、その割り当ても解除されます。

関数の **fopen()** と **fclose()** の使用方法を次のコード例に示します。

```
FILE *fpt1, *fpt2;
/* create file named "testfile1" or append to it if exists */
if((fpt1 = fopen("testfile1","a+")) == NULL) {
   printf("Cannot create or open the file\n");
   exit(1);
}
/* create file named "testfile2 for both reading and writing,
   starting at the beginning. */
if((fpt2 = fopen("testfile2","r+")) == NULL) {
   printf("Cannot open the file\n");
   exit(1);
}
...
fclose(fpt1);
fclose(fpt2);
```

20.8.2 ファイルの読み込み/書き込み

ファイルを開いてストリームに関連付けた後は、開くときのモードに従って読み込みまたは書き込みの操作が実行されます。ファイルの読み込みによく使用する関数は **getc()** と **fgetc()** です。 これらは入力ストリームから次の文字を読み込みます。関数 **gets()** と **fgets()** は、入力ストリームから指定した数の文字を読み込みます。関数 fscanf() はある入力形式の制御下でストリームからの入力を読み込みます (セクション 20.3を参照)。関数 **fread()** は、指定したサイズを持つある集合体データ型などのデータブロックを読み込むのに有効です。バッファのオーバーフローによってセキュリティの問題が起こる可能性があるため、C では関数 **gets()** は使用するべきではありません。これらの入力関数の使用例を次のプログラムに示します。

```
#include <stdio.h>

FILE *fpt;
char c;
char s[100];

if((fpt = fopen("testfile","r")) == NULL) {
   printf("Cannot create or open the file\n");
   exit(1);
}

/* read a character from file testfile */
if((c = fgetc(fpt)) != EOF)
   printf("c = %c", c);

/* read up to 99 characters from file testfile
   to string s which ends with \0 */
if(fgets(s, 100, fpt)
   printf("s = %s", s);

fclose(fpt);
```

Ch では、次のコマンドモードでファイルを対話的に操作できます。

```
> FILE *fp
> fp = fopen("testfile", "w")
> fprintf(fp, "This is output to testfile\n");
> fclose(fp)
> more testfile
This is output to testfile
>
```

この例では、コマンド **more** を使用して画面上にファイルを表示できます。上記のファイル読み込み用の入力関数にはそれぞれ、対応するファイル書き込み用の出力関数があります。ファイルの書き

込みによく使用する関数は **putc()** と **fputc()** です。 これらは出力ストリームに文字を書き込みます。関数 **puts()** と **fputs()** は、出力ストリームに文字列を書き込みます。 関数 **fprintf()** はある出力形式の制御下でストリームに出力を書き込みます (セクション 20.3を参照)。関数 **fwrite()** は、指定したサイズを持つ集合体データ型などのデータブロックを書き込むのに有効です。次に例を示します。

```
#include <stdio.h>

FILE *fpt;

if((fpt = fopen("testfile","w")) == NULL) {
   printf("Cannot create or open the file\n");
   exit(1);
}

/* write a character 'a' to file testfile */
fputc('a', fpt);
/* write a character 'b' to file testfile */
putc('b', fpt);

/* write string "this is a test" to file testfile */
fputs("this is a test", fpt);

fclose(fpt);
```

関数の **fread()** と **fwrite()** を使ったバイナリファイルの 読み込み/書き込み例を次のプログラムに示します。

```
#include <stdio.h>

FILE *fpt;
struct tag {int i; float f;} s[2];
char buf[20*sizeof(tag)];

if((fpt = fopen("testfile","rb+")) == NULL) {
   printf("Cannot create or open the file\n");
   exit(1);
}

/* read 20 elements of struct tag to buf */
if(fread(buf, sizeof(tag), 20, fpt) != 10) {
   if(feof(fpt))
      printf("End of file.");
   else
```

```
        printf("File read error.");
    }

    /* write 2 elements of struct tag to file testfile */
    s[0].i = 10; s[0].f = 1.2;
    s[1].i = 20; s[1].f = 3.4;
    fwrite(&s, sizeof(tag), 2, fpt);

    fclose(fpt);
```

20.8.3 ランダムアクセス

ハードドライブ上のファイルのようにランダムアクセスをサポートするファイルもありますが、コンソールに接続される stdout や stdin のように、ランダムアクセスをサポートしないファイルもあります。ファイルがランダムアクセスをサポートする場合は、**ファイル位置インジケータ**を使用して次の項目の読み込みや書き込みを行う位置を決定できます。既定では、ファイル位置インジケータは、開くファイルの開始位置を指します。前のセクションで説明した読み込み関数または書き込み関数によって、ファイル位置インジケータが指す位置から項目の読み込みまたは書き込みが行われると、ファイル位置インジケータの値が適切に増分され、ファイル位置インジケータが次の読み込みまたは書き込み位置を指せるようになります。たとえば、読み込まれた項目または書き込まれた項目が文字の場合、ファイル位置インジケータは 1 だけ増分されます。また、ランダムアクセスをサポートするファイルでは、関数 **fseek()** でファイル位置インジケータを設定できます。プロトタイプは次のとおりです。

```
int fseek(FILE *stream, long int offset, int whence);
```

関数は満たすことができない要求に対してのみゼロ以外の値を返します。これにより、引数 `stream` が指すストリームのファイル位置インジケータが設定されます。バイナリストリームの場合、`whence` に指定した位置に `offset` 値を追加することにより、ファイルの先頭からの文字数で測定された新しい位置を取得します。`whence` が **SEEK_SET** の場合は、指定位置はファイルの先頭になります。**SEEK_CUR** の場合はファイル位置インジケータの現在の値、**SEEK_END** の場合はファイルの終わりになります。

テキストストリームの場合、`offset` は、ゼロか、または同じファイルに関連付けられているストリームで以前に成功した **ftell()** への関数呼び出しで返された値のどちらかにならなければなりません。`whence` は **SEEK_SET** でなければなりません。

fseek の呼び出しに成功すると、更新ストリームの次の操作は、入力または出力のいずれかになります。たとえば、次のコードは、ファイル `testfile` の構造体 `S1` の 6 番目の要素を読み込みます。

```
struct S1 {
  int i;
  float f;
} s;
```

```
FILE *ftp;
int num = 6; /* the 6th element */

if((fpt = fopen("testfile","rb")) == NULL) {
   printf("Cannot create or open the file\n");
   exit(1);
}

/* set the indicator to the 6th element */
fseek(fpt, (num-1) * sizeof(S1), SEEK_SET);

/* read 6th element of struct S1 from testfile */
if(fread(&s, sizeof(S1), 1, fpt) != 1) {
   if(feof(fpt))
      printf("End of file.");
   else
      printf("File read error.");
}

fclose(fpt);
```

関数 **fseek()** のほかに、ファイル位置関数として関数 **fgetpos()** があります。これにはファイル位置インジケータの現在の値が格納されます。関数 **fsetpos()** は **fpos_t** 型のオブジェクトの値に応じてファイル位置インジケータを設定します。関数 **ftell()** は、ストリームからファイル位置インジケータの現在の値を取得します。関数 **rewind()** は、ファイルの先頭を指すようにファイル位置インジケータを設定します。

20.9 ディレクトリ操作

オペレーティングシステムによってファイルシステムが異なり、ディレクトリの内部処理の方法もシステムによって異なります。Ch では、POSIX 規格で定義されている関数 **opendir()**、**closedir()**、**readdir()**、および **rewinddir()** を使って、 システムに依存しない方法でさまざまなオペレーティングシステムのディレクトリを開き、閉じ、読み込むことができます。次に示すこれらの関数のプロトタイプは、ヘッダーファイル **dirent.h** で定義されています。

```
DIR            *opendir (const char *dirname);
struct dirent  *readdir (DIR *dirp);
void           rewinddir(DIR *dirp);
int            closedir (DIR *dirp);
```

これらの関数に加え、ディレクトリエントリ構造体 **dirent** とディレクトリストリーム構造体 **DIR** という 2 つの構造体もこのヘッダーファイルで定義されています。構造体 **dirent** は、指定したディレクトリのファイルやサブディレクトリなどのディレクトリエントリの情報を格納するのに使用します。

構造体 **dirent** でよく使用するメンバは `d_name` (ディレクトリエントリの名前) です。メンバ `d_name` は、`_MAXNAMLEN + 1` のサイズ (終端 null 文字を含む) を持つ char 型の配列です。システムに依存する値を含むマクロ `_MAXNAMLEN` は、ヘッダーファイル **dirent.h** で定義されています。構造体 **DIR** はディレクトリストリームを表します。このストリームは、特定のディレクトリにあるすべてのディレクトリエントリの順序付きシーケンスです。

20.9.1 ディレクトリを開く処理と閉じる処理

関数呼び出し *dirp* **= opendir(** *dirname* **)** で、文字列 *dirname* の引数で指定されたディレクトリに対応するディレクトリストリームを開きます。ディレクトリストリーム *dirp* が返されます。これは、ディレクトリストリームの型 DIR のオブジェクトへのポインタです。ディレクトリストリームは先頭のディレクトリエントリに置かれます。変数 *dirp* を **readdir()** などの他の関数で使用して、ディレクトリを操作することができます。関数呼び出し **closedir(** *dirp* **)** は、*dirp* が参照するディレクトリストリームを閉じ、成功するとゼロを返します。関数 **opendir()** と **closedir()** を使用したプログラムの典型的な構造を次に示します。

```
#include <stdio.h>
#include <dirent.h>
int main(void) {
  DIR *dirp;

  dirp = opendir("."); // open the current directory
  ...   // manipulating current directory by dirp
  closedir(dirp);
  return 0;
}
```

関数 **stat()** を使用して、ディレクトリを開く前に、関数 **opendir()** の引数 *dirname* がそのディレクトリの名前であることを確認することができます。関数 **stat()** の最初の引数はファイル名です。2 番目の引数は構造体 **stat** にあるオブジェクト内のそのファイルに関するすべての情報を返します。正常に完了した場合は、ゼロの値が返らなければなりません。失敗した場合は`-1` の値が返り、エラーを示す **errno** が設定されなければなりません。関数 **stat()** の次のプロトタイプは、ヘッダーファイル **sys/stat.h** で定義されています。

```
int stat(const char * name, struct stat *stbuf);
```

関数 **stat()** の 2 番目の引数で渡す値を記述する構造体は、通常、次のように定義されます。

```
struct stat {
    dev_t   st_dev;     /* block device inode is on */
    ino_t   st_ino;     /* inode number */
    mode_t  st_mode;    /* protection and file type */
    nlink_t st_nlink;   /* hard link count */
```

```
    uid_t   st_uid;    /* user id */
    gid_t   st_gid;    /* group id */
    dev_t   st_rdev;   /* the device number for a special file */
    off_t   st_size;   /* number of bytes in a file */
    time_t  st_atime;  /* time of last access */
    time_t  st_mtime;  /* time of last modify */
    time_t  st_ctime;  /* time of last status change */
}
```

構造体のこれらのメンバの意味をコメントフィールドで説明しています。**dev_t** や **mode_t** などの型定義された型はすべて、ヘッダーファイル **sys/types.h** で定義されています。次に示すマクロでメンバ **st_mode** を使用すると、ファイルが指定された型かどうかをテストできます。

```
S_ISDIR(m)    // Test macro for a directory file.
S_ISCHR(m)    // Test macro for a character special file.
S_ISBLK(m)    // Test macro for a block special file.
S_ISREG(m)    // Test macro for a regular file.
S_ISFIFO(m)   // Test macro for a pipe or a FIFO special file.
```

マクロに渡される値 *m* は、**stat** 構造体 **stbuf** の **st_mode** の値です。マクロは、テスト結果が true ならばゼロ以外の値、false ならばゼロに評価されます。次にコード例を示します。

```
#include <dirent.h>
#include <sys/stat.h>
...

struct stat stbuf;
DIR *dirp;
char *dirname = "/home/myname";
...

if(!stat(dirname, &stbuf) && S_ISDIR(stbuf.st_mode)) {
  dirp = opendir(dirname); // open the directory
}
...   // manipulating the directory by dirp

closedir(dirp);
...
```

関数 **opendir()** を使用してディレクトリを開く前に、`/home/myname` がディレクトリの名前であるかどうかをチェックできます。

関数 **exec()** のいずれかのファミリへの呼び出しが成功すると、呼び出しプロセスで開いたディレクトリストリームが閉じます。

20.9.2 ディレクトリの読み込み

dirp が参照している、現在開いているディレクトリは、関数呼び出し *direntp*= **readdir(***dirp***)**. で読み込むことができます。関数 **readdir()** は、構造体 **DIR** へのポインタ *dirp*(関数 *direntp* によって返される) 引数として受け取ります。関数 **readdir()** の戻り値 (direntp) は構造体 **dirent** へのポインタで、*dirp* が参照するディレクトリストリームの現在の位置にあるディレクトリエントリを表します。この関数により、ディレクトリストリームは次のエントリに置かれます。現在のディレクトリエントリの名前は、`direntp->d_name` で表すことができます。ディレクトリストリームの終わりに達すると、**NULL** が返されます。たとえば、プログラム 20.3は、現在のディレクトリを開いてすべてのエントリを読み込み、エントリ名を出力します。

```
#include <dirent.h>
#include <sys/stat.h>

int main() {
    struct stat stbuf;
    DIR *dirp;
    struct dirent *direntp;
    char *dirname = ".";  /* current directory */

    if(!stat(dirname, &stbuf) && S_ISDIR(stbuf.st_mode)) {
      dirp = opendir(dirname);
    }

    printf("List of files in directory %s :\n", dirname);
    while(direntp = readdir(dirp)) {
      printf("%s\n", direntp->d_name);
    }

    closedir(dirp);
    return 0;
}
```

プログラム 20.3: 検索パスの出力コマンド

別の例として、プログラム 20.4はシステム変数**_path** で各ディレクトリを検索し、実行可能なファイルのリストを stdout に出力します。ファイルが実行可能ファイルであるかどうかは、セクション 4.16で説明されている `access(file, X_OK)` の関数呼び出しでチェックできます。

```
#!/bin/ch
/*----------- printexec --------------------
This program searches through _path and
prints all the names of the executable files.
------------------------------------------*/
#include<unistd.h>
#include<sys/stat.h>
#include<dirent.h>
string_t s, filename;
struct stat sbuf;
struct dirent *direntp;
DIR * dirp;

foreach(s; _path) { //or foreach(s; _path; NULL; ";")
  dirp = opendir(s);
  if(dirp != NULL) {
    while(direntp = readdir(dirp)) {
      /* or filename = stradd(s, "/", direntp->d_name); */
      sprintf(filename, "%s/%s", s, direntp->d_name);
      stat(filename, &sbuf);
      if(S_ISREG(sbuf.st_mode) && access(filename, X_OK) == 0)
        printf("%s\n", filename);
    }
    closedir(dirp);
  }
}
```

プログラム 20.4: 検索パスの出力コマンド

プログラム 20.3に基づくプログラム 20.5は、現在のディレクトリを読み込むだけではなく、サブディレクトリを再帰的に検索します。

```
/* File: rec.ch */
#include <dirent.h>
#include <sys/stat.h>

void dirwalk(char *dirname);

int main() {
    char *dirname = ".";

    dirwalk(dirname);
    return 0;
}

/* open, read a directory, and go into its subdirectories recursively */
void dirwalk(char *dirname) {
    struct stat stbuf;
    DIR *dirp;
    struct dirent *direntp;
    char filename[1024];

    /* open the directory */
    if(!stat(dirname, &stbuf) && S_ISDIR(stbuf.st_mode)) {
      dirp = opendir(dirname);
      if (dirp == NULL) return;
    }

    /* read the directory and go into its subdirectories recursively */
    while(direntp = readdir(dirp)) {
      sprintf(filename, "%s/%s", dirname, direntp->d_name);
      stat(filename, &stbuf);
      printf("size of %s is %d\n", filename, stbuf.st_size);

      /* if the file is a directory, except for "." and ".." */
      if((strcmp(".", direntp->d_name) != 0) \
          && (strcmp("..", direntp->d_name) != 0) \
          && S_ISDIR(stbuf.st_mode))
      {
        dirwalk(filename);  /* recursive calling this function */
      }
    }

    closedir(dirp);
}
```

プログラム 20.5: ディレクトリの再帰的検索

関数 `dirwalk` は再帰関数です。ディレクトリ名を引数として扱い、値を返しません。引数で指定したディレクトリのすべてのエントリの名前とサイズを出力します。エントリがディレクトリの場合は、“`.`”と “`..`”の場合を除き、関数 `dirwalk()` によって関数自体が呼び出され、エントリの名前が引数として渡されます。

このため、このプログラムは、現在のディレクトリのすべてのサブディレクトリをループ処理できます。各ディレクトリには、ディレクトリ自体を表す “`.`”と親ディレクトリを表す “`..`”が含まれます。これらはスキップされる必要があります。そうでない場合、プログラムは無限ループに入ります。前

のセクションで説明したように、構造体 **stat** のメンバ **st_size** を使用してファイルのバイト数を示すことができます。

関数 **rewinddir()** によってディレクトリストリームの位置がリセットされ、*dirp* はディレクトリの先頭を参照します。また、**opendir()** への関数呼び出しが行われるので、ディレクトリストリームが対応するディレクトリの現在の状態を参照するようになります。値は返されません。たとえば、プログラム 20.6は、現在のディレクトリを開き、最初の 4 つのエントリを読み込んでエントリ名を出力します。次に、関数 **rewinddir()** が呼び出され、再度、先頭からディレクトリが読み込まれます。

```
#include <dirent.h>
#include <sys/stat.h>

int main() {
    int i;
    struct stat stbuf;
    DIR *dirp;
    struct dirent *direntp;
    char *dirname = ".";

    if(!stat(dirname, &stbuf) && S_ISDIR(stbuf.st_mode)) {
      dirp = opendir(dirname);
    }

    printf("The first four entries in directory %s are :\n", dirname);
    for(i = 0; i < 4; i++) {
      direntp = readdir( dirp );
      printf( "%s\n", direntp->d_name );
    }

    /* reset the position of the directory stream to which
       dirp refers to the beginning of the directory. */
    rewinddir(dirp);

    printf("\nAfter calling function rewinddir(), files in "
           "directory %s are :\n", dirname);
    while(direntp = readdir(dirp)) {
      printf("%s\n", direntp->d_name);
    }

    closedir(dirp);
    return 0;
}
```

プログラム 20.6: ディレクトリの再読み込み

第21章　セーフCh

Chは使いやすさとセキュリティを考慮して設計されました。ポインタとメモリ割り当て/割り当て解除機能によりC/C++は強力になりますが、経験の浅いプログラマはこれらの機能を簡単には使いこなすことができません。ポインタやメモリの不適切な処理により、バッファオーバーフローが発生する可能性があります。クラッシュするプログラムは極めて高い割合で、文字列の処理の誤りに起因していることがわかっています。

Chはこの欠点を認識しており、この問題を解決するための自動メモリ管理機能を持つ組み込みの文字列型を備えています。この型はchar*型およびchar[]型とシームレスに機能します。メモリ処理やポインタに関する懸念なく迅速なアプリケーション開発を行うために、この機能を使用することをお勧めします。さらに、Chはメモリの破損を避けるために自動的に配列の境界を確認します。

21.1　セーフChシェル

セーフChはCベースのアプレットがインターネットを介して実行される場合のセキュリティ上の懸案事項に対応するため、あるいは制限付きのシェルとして導入されました。セーフChではCポインタを使用できないため、装置のリアルタイム制御およびデータ取得などの他のアプリケーションでポインタの利点を活用する際の潜在的なセキュリティ上のリスクを軽減します。セーフChにはサンドボックスがあり、悪意のあるアプレットがコンピュータを完全に制御する権限を獲得しないように制限します。

21.1.1　Windowsでの起動

Chソフトウェアをダウンロードし、インストールした後は、Unixではコマンド`chs`を入力するとセーフChを起動できます。Windowsでは、[**スタート**]->[**ファイル名を指定して実行**]の順にメニューを使用して、次に、chsまたはchs.exeと入力します。セーフChシェルは、UnixとWindowsの両方で、`ch -S` というオプションを指定したコマンドを使用することで起動できます。

21.2　サンドボックス内の無効な機能

プログラムchsはセーフChシェルです。Ch言語環境が呼び出されるときに`ch -S` として`-S`フラグが指定されている場合も、Chシェルはセーフシェルとして呼び出されます。マクロ`_SCH_`はセーフシェルでは値1で事前定義されています。セーフシェルの実行環境は通常のシェルの実行環境より制御されています。`ch -S` の動作はchの動作と同じです。ただし、次の機能は無効です。

- cdとchdirによるディレクトリの変更。

- 入力コマンド、コマンドステートメント、およびコマンド置換での文字 '/' を含むパスまたはコマンド名の指定。
- ドットコマンドステートメントでの文字 '/' または '\' を含むパスまたはコマンド名の指定。
- `#include<file>`の `file` での先頭文字が '/' または '\' であるパスの指定。
- `#include "file"` は`#include<file>`として処理される。
- 入出力のリダイレクト (`<`、`<<`、`>`、`>|`、および`>>`)。
- システム変数`_path`、`_fpath`、`_lpath`、`_ppath`、`_ipath`、 `_user`、`_home`、`_cwd`、`_cwdn`、`_shell`、`_host` lvalue としての使用 (.chsrc の実行後これらのシステム変数の値は内部で使用されるために保持されます。セーフシェルユーザーの場合、これらのシステム変数は NULL 値になります。設定を確認するために、Unix では.chsrc ファイル、Windows では_chsrc ファイルのこれらのシステム変数の値を出力できます)。
- 対話型モードでのシェルコマンド `chparse` および `chrun` の使用。
- ; を含むパスまたはコマンド名の指定 (将来的に使用が可能)。
- `|` を含むパスまたはコマンド名の指定 (将来的に使用が可能)。
- ポインタ型の宣言が許可される。ただし、lvalue をポインタ型にすることはできません。次に例を示します。

  ```
  char *p, **p2;    // ok
  p = malloc(90);   // bad
  p2 = &p;          // bad
  ```

  一方、配列型を lvalue として使用することができます。次に例を示します。

  ```
  int a[2]={1,2};
  a[1] =90;
  ```

- ポインタへのポインタ以外の値のキャスト。次に例を示します。

  ```
  char *p;
  p = (char *)16; // bad
  ```

- ポインタ演算。次に例を示します。

  ```
  char *p;
  int a[15];
  p = p+128;     // bad
  *(a+10)=12;    // bad
  ```

- 汎用関数_execv()、_execvp()、_fopen()、_fork()、_fstat()、_lstat()、_pipe()、_popen()、_remove()、_rename()、_socket()、_socketpair()、_stat()、_utime()、_system()、access()、getenv()、open()、putenv()、setrlimit()、umask()。

- 型修飾子 **restrict** によって修飾されている戻り値の型を含む関数または関数ファイルは制限付き関数と呼ばれます。次の制限付き関数は、セーフ Ch プログラムでは呼び出すことができません。

  accept()、chdir()、chown()、chroot()、creat()、execl()、execv()、execle()、execve()、execlp()、execvp()、fchdir()、fchown()、fchroot()、fdopen()、fopen()、fstat()、gethostname()、kill()、lchown()、link()、lstat()、mkdir()、pipe()、popen()、remove()、rename()、rmdir()、socket()、socketpair()、stat()、system()、unlink()、

- マクロ offsetof()。

- コマンドシェルではコマンドおよびファイル名補完を使用できません。

ホームディレクトリで Unix の.chsrc および Windows の_chsrc が解釈された後に上記の制限が強制されます。セキュリティを最大限保護するため、システム管理者は Unix の.chsrc および Windows の_chsrc の所有権を取得し、このファイルのモードを読み取り専用に変更することができます。また、各プロセスのセーフシェルでの CPU リソースはファイル.chsrc を変更することによっても制限できます。-S オプションを指定して Ch を呼び出す場合、-f オプションは無視されます。これらの追加の制限は、クライアント上にある関数ファイルでは緩和されます。したがって、安全のガイドラインとして重要なのは、関数ファイルの引数を制限付き関数への直接の入力として使用しないということです。関数ファイルは、制限付き関数を呼び出すことができます。

fgets()、fread()、gets()、memcpy()、memmove()、memset()、read()、recv()、sprintf()、strcat()、strcpy()、strncat()、strncpy() の各関数内で void へのポインタまたは char へのポインタとして配列を使用する場合は、配列境界外のメモリは汚染されないことが保証されます。関数 fscanf()、scanf()、および sscanf() によって汚染されたメモリには null 文字が設定され、潜在的なセキュリティホールが閉じられます。プログラムが配列境界外のメモリに書き込もうとすると、エラーメッセージが生成されます。セーフシェルではポインタ型の変数を呼び出すことができないため、これらの関数を安全に使用できます。

実行されるコマンドが Ch プログラムであることがわかった場合、セーフシェルはプログラムを実行するために ch -S を呼び出します。

シェルを識別する#!/bin/ch をがオプション-S の指定なしで使用して Ch プログラムが呼び出された場合、セーフシェルはプログラムを実行するために ch を呼び出します。したがって、エンドユーザーに通常のシェルの完全な機能にアクセスできる Ch プログラムを提供すると同時に、限られた数のコマンド数を提供できます。このしくみは、コマンドが格納されているディレクトリの書き込み許可と実行許可をエンドユーザーが持たないことを前提としています。したがって、保証されている設定動作を実行し、ユーザーを適切なディレクトリ (おそらくログインディレクトリ以外) に配置することによって、.chsrc の作成者はユーザーの動作を完全にコントロールできます。既定のディレクトリ CHHOME/sbin は、セーフ Ch シェルによって安全に呼び出すことのできるバイナリおよび Ch コマンドを配置するために用意されているディレクトリです。

Windows では、Windows エクスプローラまたは [**スタート**]–>[**ファイル名を指定して実行**] のメニューからセーフ Ch シェルを起動できます。コマンド `chs` および `ch -S` は、コマンドプロンプトウィンドウまたは Ch ウィンドウで実行する場合、Windows NT/2000/XP では機能しません。

21.3 制限付き関数

戻り値の型を指定する型指定子 `restrict` によって宣言された関数は制限付き関数と呼ばれます。これらの関数をセーフ Ch プログラムで呼び出すことはできません。次のようになります。

```
#!/bin/ch -S
restrict void funct(void) {
   printf("This function cannot be called by Safe Ch program.\n");
}
funct(); // Error: call restricted functions
```

21.4 セーフ Ch プログラム

`CHHOME/sbin` ディレクトリおよび `CHHOME/toolkit/sbin` ディレクトリのプログラムは、通常のシェルおよびセーフ Ch シェルからアクセスできます。たとえば、`CHHOME/sbin` にあるプロット用のバイナリ実行可能プログラム **gnuplot** やライセンス情報を示す Ch プログラム **license.ch** は、セーフ Ch シェルおよびセーフ Ch スクリプトでアクセスできます。

21.5 アプレットとネットワークコンピューティング

Ch のクロスプラットフォームネットワークコンピューティングは、セーフシェルを使用して実現されます。WWW を介して転送される MIME タイプの拡張子.chs を持つデータストリームは、Ch アプレットとして処理され、セーフ Ch シェルで実行されます。ネットワークコンピューティングを実行するには、Web サーバーと Web ブラウザの両方が Ch アプレットをそれぞれ生成、認識するように設定されている必要があります。

設計および製造での利用を目的にした Ch のクロスプラットフォームネットワークコンピューティングのオンラインドキュメントとデモンストレーションについては、Web サイト
`http://www.softintegration.com` を参照してください。

第22章　ライブラリ、ツールキット、およびパッケージ

セクション 3.4.5では、複数のファイルを使用してプログラムを実行する方法を説明しました。本章では、Ch で実行するライブラリとソフトウェアパッケージを作成する方法を説明します。

22.1　ライブラリ

関数は Ch プログラムの構成要素です。関数はユーザーが作成するか、またはいくつかのソフトウェアパッケージによって提供されたライブラリに含まれています。文字や文字列に対する操作、入出力操作、数学関数、日時の変換、動的メモリ割り当てなど一般的に使用される多数の関数が、標準ライブラリに含まれています。これらの関数を再作成する必要はなく、標準ライブラリからこれらの関数を呼び出すだけで済みます。ライブラリの関数を使用するには、ライブラリの正しいヘッダーファイルを組み込むためのプリプロセッサディレクティブ`#include` がプログラムに必要です。ヘッダーファイルはアプリケーションプログラムとライブラリ間のインタフェースです。ヘッダーファイルで定義されているすべての関数プロトタイプ、データ型、およびマクロは、アプリケーションプログラムを効率的に作成するためのより多くの機能を提供します。たとえば、C の標準ライブラリのヘッダーファイル `math.h` を組み込むことにより、プログラムはこのヘッダーファイルで宣言される **sin()** や **cos()** などのすべての算術関数を使用できます。

C にはホストオペレーティングシステムから独立している多数の標準ライブラリが含まれています。このライブラリは、`float.h` で定義されている浮動小数点環境、`math.h` の算術、`stdio.h` の入出力、`string.h` の文字列処理、`time.h` の日時、`wchar.h` の拡張マルチバイトおよびワイド文字ユーティリティなどに対する演算をサポートしています。C の標準ライブラリのほか、IEEE Portable Operating System Interface (POSIX) 標準には、`dirent.h` で定義されているファイルシステムディレクトリエントリに対する操作、`fcntl.h` のプロセス間通信、`semaphore.h` のセマフォメカニズムなど、Unix システム環境間のアプリケーションプログラムの移植性を向上させる多数のライブラリが用意されています。

Ch では C および POSIX 標準で定義されているほとんどの関数をサポートしています。また、Ch では `chplot.h` で定義されている 2D および 3D プロット関数や、`numeric.h` で定義されている高度な数値分析関数など、多数の高度な関数をサポートしています。表 22.1に、Ch でサポートされているライブラリの概要を示します。Windows の場合、Ch ではライブラリ **windows.h** および **windows/winsock.h** もサポートしています。Ch がサポートするライブラリのすべてのヘッダーファイルは、`CHHOME/include` ディレクトリおよびそのサブディレクトリにあります。

Ch ではユーザー独自のライブラリを追加することができます。関数ファイル、ヘッダーファイル、動的に読み込まれるライブラリ、ライブラリのコマンドなどのコンポーネントをファイルシステム内の任意の場所に配置できます。そのため、必ずこれらのコンポーネントの場所が Ch で認識されてい

表 22.1: ライブラリの概要

ヘッダーファイル	説明	C	POSIX	Ch
aio.h	非同期入出力		X	X
arpa/inet.h	インターネット操作			X
array.h	計算配列			X
assert.h	診断	X	X	X
chplot.h	2D および 3D のプロット			X
chshell.h	Ch シェル関数			X
complex.h	複素数	X		X
cpio.h	Cpio アーカイブ値			X
crypt.h	暗号化関数			X
ctype.h	文字の処理	X	X	X
dirent.h	ディレクトリエントリの形式		X	X
dlfcn.h	動的に読み込まれる関数			X
errno.h	エラー番号	X	X	X
fcntl.h	プロセス間通信関数		X	X
fenv.h	浮動小数点の環境	X		X
float.h	プラットフォーム依存の浮動小数点の制限	X	X	X
glob.h	パス名の一致するパターンの種類			X
grp.h	グループ構造		X	X
inttypes.h	固定サイズの整数型	X		X
iostream.h	C++スタイルでの入出力ストリーム			X
iso646.h	代替スペル	X		X
libintl.h	国際化対応用のメッセージカタログ			X
limits.h	プラットフォーム依存の整数の制限	X	X	X
locale.h	ローカル関数	X	X	X
malloc.h	動的メモリ管理関数			X
math.h	数学関数	X	X	X
mqueue.h	メッセージキュー		X	X
netconfig.h	ネットワーク構成データベース			X
netdb.h	ネットワークデータベース操作			X
netdir.h	転送プロトコルの名前とアドレスのマッピング			X
netinet/in.h	インターネットプロトコルファミリ			X
new.h	C++形式でのメモリ割り当てエラー処理			X
numeric.h	数値分析			X
poll.h	poll() 関数の定義			X
pthread.h	スレッド		X	
pwd.h	パスワード構造		X	X
re_comp.h	re_comp() の正規表現検索関数			X
readline.h	Readline 関数			X
regex.h	正規表現検索の種類			X

表 22.1: ライブラリの概要 (続き)

ヘッダーファイル	説明	C	POSIX	Ch
sched.h	実行スケジューリング		X	X
semaphore.h	セマフォ関数		X	X
setjmp.h	ローカル以外のジャンプ	X	X	X
signal.h	シグナル処理	X	X	X
stdarg.h	可変長引数リスト	X	X	X
stdbool.h	ブール数	X		X
stddef.h	その他の関数とマクロ	X	X	X
stdint.h	整数型	X	X	X
stdio.h	入出力	X	X	X
stdlib.h	ユーティリティ関数	X	X	X
string.h	文字列関数	X	X	X
stropts.h	ストリームインタフェース			X
sys/acct.h	プロセスアカウンティング			X
sys/fcntl.h	ファイル制御			X
sys/file.h	ファイル構造配列へのアクセス			X
sys/ioctl.h	デバイス制御			X
sys/ipc.h	プロセス間通信アクセス構造			X
sys/lock.h	プロセスのロック			X
sys/mman.h	メモリ管理宣言		X	X
sys/msg.h	メッセージキュー構造			X
sys/procset.h	プロセスの設定			X
sys/resource.h	XSI リソース操作			X
sys/sem.h	セマフォ機能			X
sys/shm.h	共有メモリ機能			X
sys/socket.h	インターネットプロトコルファミリ			X
sys/stat.h	ファイル構造関数		X	X
sys/time.h	時間型			X
sys/times.h	ファイルのアクセスおよび修正時の構造		X	X
sys/types.h	データ型		X	X
sys/uio.h	ベクトル入出力操作			X
sys/un.h	Unix ドメインソケット			X
sys/utsname.h	システム名構造		X	X
sys/wait.h	終了ステータスの評価		X	X
syslog.h	システムエラーロギング			X
tar.h	拡張 tar 定義			X
termios.h	termios の値の定義		X	X
tgmath.h	型汎用数学関数	X		X

表 22.1: ライブラリの概要 (続き)

ヘッダーファイル	説明	C	POSIX	Ch
time.h	時間および日付関数	X	X	X
tiuser.h	トランスポート層インタフェース			X
unistd.h	システム関数およびプロセス関数		X	X
utime.h	アクセスおよび修正時の構造		X	X
wait.h	子プロセスの停止または終了を待機			X
wchar.h	マルチバイト入出力関数および文字列関数	X		X
wctype.h	マルチバイト文字クラスのテスト	X		X

注: 記号 'X' はライブラリが標準でサポートされていることを示します。

ることが重要です。前述したように、 Ch ではシステム変数 **_fpath**、**_ipath**、 **_lpath** および **_path** で指定されたディレクトリで関数ファイル、ヘッダーファイル、動的に読み込まれるライブラリおよびコマンドがそれぞれ検索されます。システム変数 **_path**、**_ipath**、**_fpath**、および **_lpath** の説明と既定値については、セクション 2.3.1を参照してください。これらのシステム変数の現在の値を確認するには、コマンドモードに変数名を直接入力してください。たとえば、**_ipath** の現在の値を確認するには、次のコマンドを入力してください。

```
> _ipath
/usr/ch/include;/usr/ch/toolkit/include;
>
```

その結果、 Ch はまずディレクトリ `/usr/ch/include` でヘッダーファイルを検索し、見つからない場合は、ディレクトリ `/usr/ch/toolkit/include` を検索することになります。

ソフトウェアパッケージに含まれる Ch プログラム `pack1.ch` で次のステートメントを記述しているとします。

```
#include <stdio.h>
#include "pack1.h"
int main() {
  /* ... */
  myfunc1(10);
  /* ... */
  return 0;
}
```

このプログラムを実行するには、 Ch がヘッダーファイル **stdio.h**、`pack1.h` および関数ファイル `myfunc1.chf` を検索する場所を認識している必要があります。既定では、 Ch はシステム変数 **_ipath** で指定されたディレクトリでヘッダーファイル **stdio.h** を検索し、現在のディレクトリ、システム変数 **_ipath** で指定されたディレクトリの順でヘッダーファイル `pack1.h` を検索し、システム変数 **_fpath** で指定されたディレクトリで関数ファイル `myfunc1.chf` を検索します。

Ch が正しいディレクトリで正しいファイルを確実に検索できるようにするために、該当するファイルのあるディレクトリを対応するシステム変数の値に追加するか、対応するシステム変数に既に含まれているディレクトリにこれらのファイルをコピーすることができます。しかし、ほとんどの場合、ユーザーにはこれらのシステム変数の既定値に含まれるディレクトリの書き込み許可がありません。汎用関数 **stradd()** を使用すると、対応するシステム変数の値にパスを追加できます。関数ファイル myfunc1.chf がディレクトリ /home/mydir/pack1/lib にあり、ヘッダーファイル pack1.h がディレクトリ /home/mydir/pack1/include にあるとします。次のコマンドでは、これらの 2 つのディレクトリをシステム変数 **_fpath** および **_ipath** の末尾にそれぞれ追加します。

```
> _fpath = stradd(_fpath, "/home/mydir/pack1/lib;")
/usr/ch/lib/libc;/usr/ch/lib/libch;/usr/ch/lib/libopt;
/usr/ch/lib/libch/numeric;/home/mydir/pack1/lib;
> _ipath = stradd(_ipath, "/home/mydir/pack1/include;")
/usr/ch/include;/usr/ch/toolkit/include;/home/mydir/pack1/include;
>
```

ユーザーが Ch を起動するたびにこれらの 2 つのパスを自動的に追加するようにする場合は、

```
_fpath = stradd(_fpath, "/home/mydir/pack1/lib;");
_ipath = stradd(_ipath, "/home/mydir/pack1/include;");
```

Unix のユーザーのホームディレクトリ内の **.chrc** などのスタートアップファイルに、次のコマンドを追加する必要があります。スタートアップファイルをカスタマイズする方法の詳細については、セクション 3.2を参照してください。その後、プログラム pack1.ch が実行されると、Ch は /usr/ch/include、/usr/ch/toolkit/include、および /home/mydir/pack1/include の各ディレクトリで順にヘッダーファイル pack1.h を検索します。同様に、関数ファイルの既定パスを検索した後に、Ch はディレクトリ /home/mydir/pack1/lib で関数ファイル myfunc1.chf を検索します。

22.2　ツールキット

汎用ユーティリティを提供する C および POSIX の標準ライブラリとは異なり、Windows、X-Windows、Motif、OpenGL、ODBC などのいくつかの自己完結型システムおよびソフトウェア標準では、特定の目的または特定の分野の関数のみが提供されます。たとえば、X-Windows は、ネットワークベースのグラフィックウィンドウシステムである X-Windows に Xlib と呼ばれる低レベルのプログラミングインタフェースを提供します。Motif は X-Windows に高度なプログラミングインタフェースを提供します。OpenGL は 3D グラフィックス用のプログラミングインタフェースを提供します。ODBC はデータベースアクセス用のプログラミングインタフェースを提供します。これらのソフトウェア標準は、Ch ではツールキットとして扱われ、CHHOME/toolkit に置かれています。

これらのツールキットを整理するために、すべてのヘッダーファイルはディレクトリ CHHOME/toolkit/include およびそのサブディレクトリに配置され、動的に読み込まれるファイルはディレクトリ CHHOME/toolkit/dl に配置されます。既定では、これらの 2 つのシステムディレクトリは既にシステム変数 **_ipath** および **_lpath** に含まれています。ただし、Ch 関数ファイルが配置

されるディレクトリは数が多すぎるためシステム変数 **_fpath** に前もって追加することができません。Ch での解決策は、プリプロセッサディレクティブ`#pragma _fpath`を対応するヘッダーファイルに追加することです。表 5.2で説明しているとおり、プラグマディレクティブ

```
#pragma _fpath <fpathname>
```

は、 Ch プログラムを実行するサブシェルのシステム変数 **_fpath** に、ディレクトリ`CHHOME/toolkit/lib/fpathname`を追加します。たとえば、GTK ツールキットのヘッダーファイル`gtk/gtk.h`に次の行が含まれているとします。ヘッダーファイル`gtk/gtk.h`内のディレクティブ

```
#ifndef __GTK_H__
#define __GTK_H__
#pragma _fpath <GTK/gtk>
...

#endif /* __GTK_H__ */
```

`#pragma _fpath <GTK/gtk>`によって、GTK を使用する Ch プログラムはディレクトリ`CHHOME/toolkit/lib/GTK/gtk`で関数ファイルを検索します。プラグマディレクティブにはもう 1 つの形式があります。

```
#pragma _fpath /dir1/dir2/fpathname
```

この形式では、サブシェルのシステム変数 **_fpath** に絶対パス名`/dir1/dir2/fpathname`を追加できます。同様に、ディレクティブ

```
#pragma _ipath /dir1/dir2/ipathname
#pragma _lpath /dir1/dir2/lpathname
#pragma _path  /dir1/dir2/pathname
```

は、これらの絶対パス名をシステム変数 **_ipath**、**_lpath**、および **_path** にそれぞれ追加します。プラグマディレクティブ

```
#pragma exec expr
```

は、式の解析時に式を評価します。このディレクティブを使用すると、システムパス変数にパスを追加できます。たとえば、関数呼び出し`getenv("HOME")`で取得されたホームディレクトリが`/home/myname`であるとします。

```
#pragma exec _fpath=stradd(_fpath, getenv("HOME"), "/chfunc;");
```

このディレクティブは、実行時に関数ファイルのシステム変数 **_fpath** にディレクトリ`/home/myname/chfunc`を追加します。

22.3 パッケージ

セクション 3.4.5では、前処理ディレクティブ pragma を import および importf と併用して複数のファイルを含むプログラムを実行する方法を説明しました。このセクションでは、Ch でのパッケージの処理について説明します。

Ch に付属のライブラリとツールキットのほかに、Ch ではユーザーが開発したソフトウェアパッケージもサポートできます。ソフトウェアパッケージはさまざまなディレクトリ内にある多数のファイルから構成される場合があり、これらのファイルはヘッダーファイル、関数ファイル、動的に読み込まれるライブラリ、インポートされたファイル、コマンドなどのパッケージ内部または外部の他のコンポーネントやファイルに関連している可能性があるため、ソフトウェアパッケージのプログラムを正しく実行するには、Ch がこれらの関連ファイルを特定できる場所を認識している必要があります。Ch の規則に従って、ソフトウェアパッケージ用のヘッダーファイルは include サブディレクトリ、関数ファイルは lib ディレクトリ、動的に読み込まれるライブラリは dl サブディレクトリ、コマンドファイルは bin サブディレクトリに置かれます。

ソフトウェアパッケージに含まれる Ch プログラム pack1.ch で次のステートメントを記述しているとします。

```
#include <stdio.h>
#include "pack1.h"
int main() {
  /* ... */
  myfunc1(10);
  /* ... */
  return 0;
}
#pragma importf "module2.ch"
#pragma import  <module3.ch>
```

このプログラムを実行するには、 Ch は解析フェーズで、いくつかのパスでヘッダーファイル **stdio.h** と pack1.h、関数ファイル myfunc1.chf、およびインポートされたファイル module2.ch と module3.ch を検索する必要があります。既定では、Ch はシステム変数 **_ipath** で指定されたディレクトリでヘッダーファイル **stdio.h** を検索します。現在のディレクトリ、システム変数 **_ipath** で指定されたディレクトリの順でヘッダーファイル pack1.h を検索します。システム変数 **_fpath** で指定されたディレクトリで関数ファイル myfunc1.chf を検索します。現在のディレクトリ、システム変数 **_fpath** で指定されたディレクトリの順でファイル module2.ch を検索します。システム変数 **_path** で指定されたディレクトリで module3.ch を検索します。関数ファイルで使用する場合は、動的に読み込まれるファイルをシステム変数 **_lpath** で指定されたディレクトリで検索します。

多くのソフトウェアパッケージでは、これらのソフトウェアパッケージのすべてのディレクトリをシステム変数に格納した場合、システム変数の値のサイズが大きくなりすぎて、効率的な管理や検索が困難になります。Ch ではソフトウェアパッケージの整理に役立つよう、ディレクティブ #pragma package を取り入れています。

前のセクションで説明したプリプロセッサディレクティブ #pragma _fpath はシステム変数 _fpath にユーザー指定の単一のディレクトリを追加しました。これと同様に、プリプロセッサディ

レクティブ`#pragma package`は`_fpath`、`_ipath`、`_lpath`、および`_path`などの該当するシステム変数に複数のユーザー指定ディレクトリを自動的に追加します。これらのすべてのシステム変数は、Ch プログラムを実行するサブシェル内で更新されます。このディレクティブは親シェルのシステム変数には影響しません。表 5.2で説明しているように、ディレクティブ`#pragma package`には 3 つの形式があります。最初の形式は次のとおりです。

```
#pragma package <packagename>
```

これは、指定されたディレクトリ`_ppath/packagename`内のサブディレクトリ
`bin`、`lib`、`include`および`dl`をシステム変数**_path**、**_fpath**、**_ipath**、および**_lpath**にそれぞれ追加します。システム変数**_ppath**の説明と既定値については、セクション 2.3.1を参照してください。システム変数**_ppath**の値が`/usr/ch/package`であるとすると、ディレクティブ`#pragma package <pack1>`は、パス名`/usr/ch/package/pack1/bin`、`/usr/ch/package/pack1/lib`、`/usr/ch/package/pack1/include`、および`/usr/ch/package/pack1/dl`を対応するシステム変数に追加します。他の 2 つの形式は、次のとおりです。

```
#pragma package "/home/mydir/packagename"
```

および、次のとおりです。

```
#pragma package </home/mydir/packagename>
```

これらは、指定されたディレクトリ`/home/mydir/packagename`内のサブディレクトリ`bin`、`lib`、`include`および`dl`をシステム変数**_path**、**_fpath**、**_ipath**、および**_lpath**にそれぞれ追加します。絶対パス名がこれら 2 つの場合に使用されます。

次の Ch ソフトウェアパッケージの例で、ディレクティブ`#pragma package`のしくみを説明します。このパッケージのファイルが次に示すようにディレクトリ`/home/mydir/pack1`のサブディレクトリにあるとします。

```
pack1.ch     --  /home/mydir/pack1/bin
pack1.h      --  /home/mydir/pack1/include
module1.ch   --  /home/mydir/pack1/bin
myfunc1.chf  --  /home/mydir/pack1/lib
```

リスト 1 - ファイル`/home/mydir/pack1/bin/pack1.ch`

```
#!/bin/ch
#pragma package </home/mydir/pack1>
#pragma import <module1.ch>

#include <stdio.h>
#include "pack1.h"

int main() {
  printf("Output from the main function\n");
```

```
  myfunc1(10);
  return 0;
}
```

このChプログラムを実行すると、ディレクティブ#pragma package </home/mydir/pack1>は、ディレクトリ/home/mydir/pack1のサブディレクトリbin、lib、include、およびdlをシステム変数**_path**、**_fpath**、**_ipath**、および**_lpath**の末尾にそれぞれ追加して、Chが解析フェーズで関連するすべてのコンポーネントを検索できるようにします。

リスト2 - ファイル/home/mydir/pack1/include/pack1.h

```
#ifndef PACK1_H
#define PACK1_H
int myfunc1(int i);
#endif // PACK1_H
```

ユーザ定義関数myfunc1()のプロトタイプは、このヘッダーファイル内にあります。

リスト3 - 1つのステートメントを含むファイル/home/mydir/pack1/bin/module1.ch

```
printf("Output from module1.ch in the subdirectory bin\n");
```

リスト4 - ファイル/home/mydir/pack1/lib/myfunc1.chf

```
int myfunc1(int i) {
  printf("Output from the function file myfunc1.ch in\
          subdirectory lib, i = %d\n", i);
  return 0;
}
```

関数myfunc1()はこの関数ファイル内で定義されます。

リスト5 - Chプログラムpack1.chの実行結果

```
> cd /home/mydir/pack1/bin
> ./pack1.ch
Output from module1.ch in the subdirectory bin
Output from the main function
Output from the function file myfunc1.ch in subdirectory lib, i = 10
>
```

パス名/home/mydirがシステム変数_ppathに追加された場合、またはパッケージがCHHOME/package/pack1 (CHHOMEはChのホームディレクトリ) に移動された場合は、プログラムpack1.chでディレクティブ#pragma package </home/mydir/pack1>を#pragma package

<pack1>に変更できます。実行結果は同じです。また、プログラム pack1.ch をコマンドの検索パスに含まれる任意のディレクトリに移動できます。

Ch 内で C プログラムを実行するときに、元の C のコードを一切変更しないことが望ましい場合があります。これは、ヘッダーファイルとスタートアップファイルを変更することによって実現できます。たとえば、パッケージ pack1 が CHHOME/package/pack1 にインストールされ、ヘッダーファイル pack1.h に次のコードが含まれているとします。

```
#ifndef PACK1_H
#define PACK1_H
#pragma package pack1
int myfunc1(int i);
#endif // PACK1_H
```

システムのスタートアップファイル chrc または個々のユーザーのスタートアップファイル (Unix の.chrc または Winodws の_chrc) で次のステートメントを使用して、ヘッダーファイル用のシステム変数 **_ipath** の新しいパスを追加できます。

```
_ipath = stradd(_ipath, "<CHHOME>/package/pack1/include;");
```

ここで、<CHHOME>は Ch ホームディレクトリに置換される必要があります。これで、プログラム pack1.ch または pack1.c を修正することなく実行できます。

Ch シェルのコマンド crteatepkg.ch を使用して Ch パッケージをパッケージ化できます。たとえば、ディレクトリ sample にヘッダーファイル、関数ファイル、動的に読み込まれるライブラリ、およびデモ用の個別のサブディレクトリを持つパッケージが含まれているとします。次のコマンド

```
> ch createpkg.ch sample 1.0
```

は、sample-1.0.tar.gz というパッケージファイルを作成します。このファイルは、解凍した後に Ch の **installpkg.ch** コマンドでインストールできます。

第II部

科学計算用ライブラリ

第23章　2次元プロットと3次元プロット

Ch または C++では、SoftIntegration Graphical Library を使用することで 2 次元と 3 次元のプロットを容易に作成できます。SoftIntegration Graphical Library を使用するプログラムは、Ch のインタープリタを介するか、または C++コンパイラによるコンパイルで、コードを移植しながら実行されます。プロットは、データ配列またはファイルから生成することができ、画面に表示するか、または各種のファイル形式で画像ファイルとして保存するか、Web サーバーを介して Web ブラウザに表示するために png また gif ファイル形式で stdout ストリームとして出力できます。この章では、2 次元空間および 3 次元空間でプロットを生成するプログラムを記述する方法を説明します。Mac OS X でプロットを行うには、http://aquaterm.soureforge.net から Aquaterm を入手し、

```
putenv("GNUTERM=aqua");
```

の行をシステムスタートアップファイル `CHHOME/config/chrc` または各ユーザーのホームディレクトリにあるスタートアップファイル `.chrc` に追加します。

23.1　プロットのクラス

プロット作成クラス **CPlot** を使用すると、高度なプロットの作成と操作ができます。**CPlot** クラスのメンバ関数を表 23.1に示します。各関数の詳細については、『*Ch* **言語環境リファレンスガイド**』のプロット作成に関する章を参照してください。以降のセクションでは、2 次元プロットと 3 次元プロットのどちらにも該当する機能について説明します。

23.1.1　プロットのためのデータ

プロット作成には、データセットが必要です。プログラムのメモリまたはファイルに、プロットのためのデータを格納できます。2 次元プロットで使用されるデータの最も単純な形式は、プログラム 23.1に示すように 2 つの配列から成ります。一方の配列が x 軸を示し、もう一方が y 軸を示します。図 23.1は、プログラム 23.1によって生成されたプロットです。

関数 **CPlot**::**data2D**() は、プロットのためのデータを追加します。関数 **CPlot**::**plotting**() が呼び出されると、プロットが生成されます。関数 $x\sin(x)$ がプロットされる場合、プログラム 23.1で、ステートメント `y = x*sin(x)` を要素単位の配列乗算演算子を持つ `y = x.*sin(x)` に変更する必要があります。

メンバ関数

```
int CPlot::data2D(array double  x[&], array double &y);
```

表 23.1: CPlot クラスのメンバ関数

関数	説明
CPlot()	クラスコンストラクタ。クラスの新しいインスタンスを作成し、初期化します。
˜CPlot()	クラスデストラクタ。クラスのインスタンスに関連付けられているメモリを解放します。
arrow()	矢印をプロットに追加します。
autoScale()	プロット軸の自動スケール調整を有効または無効にします。
axis()	2次元プロットで、x-y 軸の描画を有効または無効にします。
axisRange()	プロット軸の範囲を設定します。
axes()	データセットの軸を指定します。
barSize()	エラーバーのサイズを設定します。
border()	プロット周囲の境界の描画を有効または無効にします。
borderOffsets()	プロット境界のプロットオフセットを設定します。
boxBorder()	ボックス型のプロットで、境界の描画の有効と無効を切り替えます。
boxWidth()	ボックスの幅を設定します。
changeViewAngle()	3次元プロットの視野角を変更します。
circle()	2次元プロットに円を追加します。
colorBox()	3次元サーフェスプロットのカラーボックスの描画を有効または無効にします。
contourLabel()	3次元サーフェスプロットの輪郭ラベルを有効または無効にします。
contourLevels()	特定の位置に表示される3次元プロットに輪郭レベルを設定します。
contourMode()	3次元サーフェスプロットに輪郭表示モードを設定します。
coordSystem()	3次元プロットに座標系を設定します。
data()	2次元、3次元、または多次元のデータを **CPlot** クラスのインスタンスに追加します。
data2D()	1つ以上の2次元のデータセットを **CPlot** クラスのインスタンスに追加します。
data2DCurve()	2次元曲線のデータセットを **CPlot** クラスのインスタンスに追加します。
data3D()	1つ以上の3次元のデータセットを **CPlot** クラスのインスタンスに追加します。
data3DCurve()	3次元曲線のデータセットを **CPlot** クラスのインスタンスに追加します。
data3DSurface()	3次元サーフェスのデータセットを **CPlot** クラスのインスタンスに追加します。
dataFile()	**CPlot** クラスのインスタンスにデータファイルを追加します。
dataSetNum()	**CPlot** クラスのインスタンス内の現在のデータセット番号を取得します。
deleteData()	以前に使用した **CPlot** クラスのインスタンスからデータを削除します。
deletePlots()	以前に使用した **CPlot** クラスのインスタンスからすべてのデータを削除し、オプションを既定値に再初期化します。
dimension()	2次元または3次元にプロット次元を設定します。
displayTime()	プロットの現在の日時を表示します。
enhanceText()	特殊シンボルに対して拡張テキストを使用します。
fillStyle()	ボックスまたは曲線を単色またはパターンで塗りつぶします。
func2D()	関数を使用して **CPlot** クラスのインスタンスに2次元のデータセットを追加します。
func3D()	関数を使用して **CPlot** クラスのインスタンスに3次元のデータセットを追加します。

表 23.1: **CPlot** クラスのメンバ関数 (続き)

関数	説明
funcp2D()	パラメータを指定した関数を使用して **CPlot** クラスのインスタンスに 2 次元のデータセットを追加します。
funcp3D()	パラメータを指定した関数を使用して **CPlot** クラスのインスタンスに 3 次元のデータセットを追加します。
getLabel()	軸のラベルを取得します。
getOutputType()	プロット出力の種類を取得します。
getSubplot()	サブプロットの要素へのポインタを取得します。
getTitle()	プロットのタイトルを取得します。
grid()	グリッドの表示を有効または無効にします。
isUsed()	**CPlot** クラスのインスタンスが使用されたかどうかをテストします。
label()	軸ラベルを設定します。
legend()	データセットの凡例を追加します。
legendLocation()	プロットの凡例の位置を指定します。
legendOption()	プロットの凡例のオプションを設定します。
line()	プロットに線を追加します。
lineType()	線の種類、幅、色、縦線、ステップなどを設定します。
margins()	プロットにマージンを設定します。
origin()	プロットの境界ボックスの原点の位置を設定します。
outputType()	プロット出力の種類を設定します。
plotType()	プロットの種類を設定します。
plotting()	**CPlot** クラスのインスタンスからプロットを生成します。
point()	プロットに線を追加します。
pointType()	点の種類、サイズおよび色を設定します。
polarPlot()	2 次元プロットに極座標系を使用するように設定します。
polygon()	プロットに多角形を追加します。
rectangle()	2 次元プロットに長方形を追加します。
removeHiddenLine()	3 次元プロットの陰線消去を有効または無効にします。
scaleType()	プロットの軸のスケールの種類を設定します。
showMesh()	3 次元プロットのメッシュの表示を有効または無効にします。
size()	プロットのサイズを変更します。
size3D()	3 次元プロットのサイズを変更します。
sizeRatio()	プロットのアスペクト比を変更します。
smooth()	データの補間と近似でプロット曲線を滑らかにします。
subplot()	サブプロットのグループを作成します。
text()	プロットにテキスト文字列を追加します。
tics()	軸のチックマークの表示を有効または無効にします。
ticsDay()	軸のチックマークのラベルを曜日単位に設定します。
ticsDirection()	軸のどの方向にチックマークが描かれるかを設定します。

表 23.1: **CPlot** クラスのメンバ関数 (続き)

関数	説明
ticsFormat()	チックラベルの数の書式を設定します。
ticsLabel()	任意の軸ラベルについて、表示位置とテキストラベルを設定します。
ticsLevel()	3 次元プロットで、チックマーク描画時の z-軸オフセットを設定します。
ticsLocation()	軸のチックマークの位置が、境界または軸の上になるように指定します。
ticsMirror()	反対側の軸における軸のチックマークの表示を有効または無効にします。
ticsMonth()	軸のチックマークのラベルを月単位に設定します。
ticsPosition()	軸の指定された位置にチックマークを追加します。
ticsRange()	軸に対するチックの範囲を指定します。
title()	プロットタイトルを設定します。

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

int main() {
    int numpoints = 36;
    array double x[numpoints], y[numpoints];
    class CPlot plot;

    lindata(-M_PI, M_PI, x);
    y = sin(x);
    plot.data2D(x, y);
    plot.plotting();
}
```

プログラム 23.1: **CPlot** クラスを使用する単純なプログラム

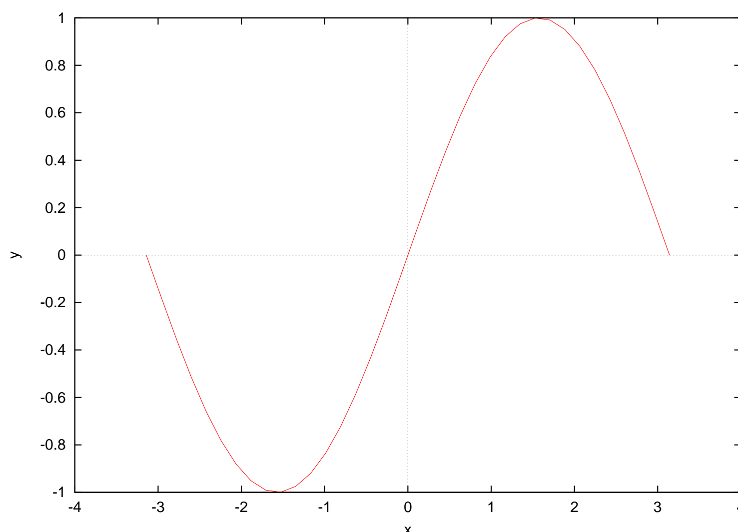


図 23.1: 非常に単純なプロット

の先頭のパラメータ x は、参照型の 1 次元配列です。関数 **CPlot**::**data2D**(x, y) の 2 番目のパラメータ y は、参照型の配列です。配列 x のサイズが n の場合、サイズ n の 1 次元配列またはサイズ m × n の 2 次元配列とすることができます。

y が m × n の行列である場合は、各 m 行に、x に対する y の列がプロットされます。y の列はそれぞれが個別のデータセットです。配列 x と y は実数型です。関数 **CPlot**::**data2D**(x, y) が多態性である場合、データは、内部処理で double 型へ変換されます。たとえば、計算配列の宣言であるプログラム 23.1の x と y のデータ型が、次のように double 型から float 型に変更されるとします。

```
array float x[numpoints], y[numpoints];
```

この場合、表示されるプロットに変更はありません。メンバ関数 **CPlot**::**data2D**() の引数 x と y のデータは、以下のように、計算配列の式とすることもできます。

```
array double complex z[numpoints];
/* ... */
plot.data2D(real(z), imag(z));
```

プロットが生成される前に、値 NaN の配列 y のデータ点を内部処理で削除します。この値に y の要素を手動で設定することによって、データセットに "すき間"を構築することができます。

メンバ関数は、次のように、2 次元曲線のプロットのデータを **CPlot** クラスのインスタンスに追加することができます。

```
int CPlot::data2DCurve(double  x[], double y[], int n);
```

1 次元配列 x および y には、サイズ n と同じ数の要素があります。

同様に、次のメンバ関数は、3 次元のプロット用のデータを **CPlot** クラスのインスタンスに追加することができます。

```
int CPlot::data3D(array double x[&], array double y[&],
                  array double &z);
```

直交データについては、x はサイズ n_x の 1 次元配列であり、y はサイズ n_y の 1 次元配列です。プロット対象のデータの種類に応じて、配列 z は 2 つの異なるサイズになります。3 次元曲線用のデータの場合、z はサイズ n_z の 1 次元配列またはサイズ m × n_z の 2 次元配列で、$n_x = n_y = n_z$ です。プログラム 23.2に対応するプロットは図 23.2です。この図は、空間曲線がどのように生成されるかを示します。

3 次元サーフェスまたはグリッドのデータの場合、z は m × n_z で、$n_z = n_x \cdot n_y$ です。円筒または球面のデータでは、データ x はサイズ n_x (θ を表す) の 1 次元配列であり、y はサイズ n_y(z または ϕ を表す) の 1 次元配列であり、z は $n_x = n_y = n_z$ と等しい m × n_z(r を表す) です。z の各 m 列は、x と y に対してプロットされ、個別のデータセットに対応します。どのような場合でも、データ配列は、サポートされているデータ型であれば、実数を指定できる任意のデータ型とすることができます。関数 **CPlot**::**data2D**() と同様に、double 型のデータの変換は内部処理されます。

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

int main() {
    array double x[360], y[360], z[360];
    class CPlot plot;

    lindata(0, 360, x);
    y = sin(x*M_PI/180);
    z = cos(x*M_PI/180);
    plot.data3D(x, y, z);
    plot.plotting();
}
```

プログラム 23.2: 3 次元曲線のプロットプログラム

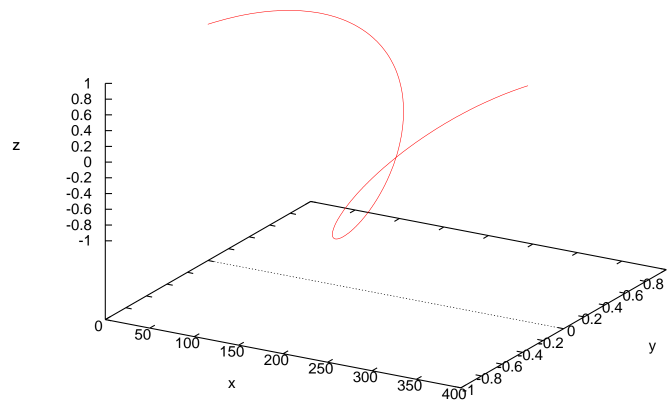


図 23.2: 3 次元曲線のプロット

3次元グリッドでは、zデータの順序が重要です。z値は、x値を一定に保ったまま全てのy値について計算されます。その後x値が1つインクリメントされ、再び全てのy値についてのz値の計算が行なわれます。すべてのデータの計算を終えるまで、これが繰り返されます。このため、10x20グリッドのデータは、次のような順序で並べられることになります。

x1 y1 z1
x1 y2 z2
.
.
.
x1 y19 z19
x1 y20 z20
x2 y1 z21
x2 y2 z22
.
.
.
x2 y19 z29
x2 y20 z30
x3 y1 z31
x3 y2 z32
.
.
.
x10 y18 z198
x10 y19 z199
x10 y20 z200

図 23.3の3次元プロットは、プログラム 23.3によって生成されます。プログラム 23.2とは異なり、プログラム 23.3の配列 z の要素数 (600) は、配列 x の要素数 (20) と配列 y の要素数 (30) の積です。メンバ関数 **CPlot**::**colorBox**() を呼び出して、3次元プロットのカラーパレットの最大値と最小値の間を変化する滑らかな色のグラデーションのカラーボックスを非表示にすることもできます。

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

#define NUMX 20
#define NUMY 30
int main() {
    double x[NUMX], y[NUMY], z[NUMX*NUMY];
    double r;
    int i, j;
    class CPlot plot;

    lindata(-10, 10, x);
    lindata(-10, 10, y);
    for(i=0; i<NUMX; i++) {
        for(j=0; j<NUMY; j++) {
            r = sqrt(x[i]*x[i]+y[j]*y[j]);
            z[30*i+j] = sin(r)/r;
        }
    }
    plot.data3D(x, y, z);
    plot.plotting();
}
```

プログラム 23.3: 3 次元グリッドのプロットプログラム

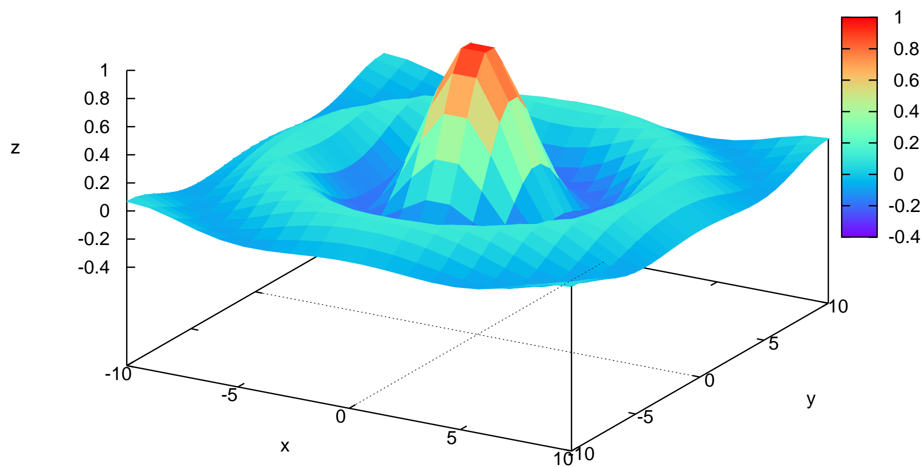


図 23.3: 3 次元グリッドのプロット

また、次のメンバ関数を使うと、**CPlot** クラスのインスタンスに3次元曲線プロットのデータを追加することができます。

```
int CPlot::data3DCurve(double  x[], double y[], double z[], int n);
```

1次元配列 x、y、および z には、サイズ n と同じ数の要素があります。次のメンバ関数を使うと、**CPlot** クラスのインスタンスに 3D サーフェスプロットのデータセットを追加することができます。

```
int CPlot::data3DSurface(double  x[], double y[], double z[],
                                int n, int m);
```

1次元配列 x の要素数がサイズ n であり、y がサイズ m である場合、z は n サイズ $n_z = n \cdot m$ の1次元配列になります。直交座標系で、配列 x、y、z は、それぞれ X-Y-Z 座標の値を表します。円柱座標系では、配列 x、y、z は、それぞれ θ 座標、z 座標、r 座標を表します。球座標系では、配列 x、y、z は、それぞれ θ 座標、ϕ 座標、r 座標を表します。また、プロットのデータを最初にファイルに格納した後、次の関数を使用してそのデータを取得することもできます。

```
int CPlot::dataFile(string_t filename, ... /* string_t option */);
```

各データファイルは、1つのデータセットに対応しています。データファイルでは、各データ点が別々の線上に配置されるように、データ形式を設定する必要があります。2次元データは、データ点1個につき、2つの値によって指定されます。2次元データファイルに空白の行があると、プロット内の曲線が中断します。この方法では、複数の曲線をプロットできますが、すべての曲線が同じプロットスタイルになります。データファイル内で、 # 記号が行の先頭にある場合、その行がコメントアウトされていることを示します。たとえば、プログラム 23.4は、図 23.1に示すプロットを生成します。

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

int main() {
    string_t filename;
    int i;
    class CPlot plot;
    FILE *out;

    filename = tmpnam(NULL);       //Create a temporary file.
    out=fopen (filename,"w");      //Write data to the file.
    for (i=-180;i<=180;i++)
       fprintf(out,"%i %f \n",i,sin(i*M_PI/180));
    fclose(out);
    plot.dataFile(filename);
    plot.plotting();
    remove(filename);
}
```

プログラム 23.4: ファイルのデータを使用するプロットプログラム

3 次元データは、データ点 1 個につき、3 つの値によって指定されます。3 次元のグリッドデータまたは 3 次元のサーフェスデータでは、データファイル内の各列は、空白行によって区切られます。たとえば、3 x 3 のグリッドは、次のように表されます。

```
# This is a comment line
x1  y1  z1
x1  y2  z2
x1  y3  z3

x2  y1  z4
x2  y2  z5
x2  y3  z6

x3  y1  z7
x3  y2  z8
x3  y3  z9
```

データファイル内に 2 行連続して空白行があると、プロットに中断が生じます。この方法では、複数の曲線またはサーフェスをプロットできますが、すべての曲線とサーフェスは同じプロットスタイルになります。3 次元データファイルを追加するには、その前に、引数値 3 を指定したメンバ関数 **CPlot**::**dimension**() を呼び出す必要があります。

23.1.2 注釈

軸のタイトルとラベルに注釈を追加できます。それには、次のメンバ関数

```
void CPlot::title(string_t title);
```

および

```
void CPlot::label(int axis, string_t label);
```

をそれぞれ使用します。メンバ関数 **CPlot**::**label**() の引数 *axis* は、設定する軸を示します。軸に対して使用可能なマクロを、表 23.2に示します。

表 23.2: 軸のためのマクロ

PLOT_AXIS_X	Select the x axis only.
PLOT_AXIS_X2	Select the x2 axis only.
PLOT_AXIS_Y	Select the y axis only.
PLOT_AXIS_Y2	Select the y2 axis only.
PLOT_AXIS_Z	Select the z axis only.
PLOT_AXIS_XY	Select the x and y axes.
PLOT_AXIS_XYZ	Select the x, y, and z axes.

図 23.4は、メンバ関数 **CPlot**::**title()** および **CPlot**::**label**() を使用するプログラム 23.5によって生成されたプロットを示します。既定では、タイトルは表示されませんが、座標軸にはそれぞれ x、y、z という記号によるラベルが設定されます。

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

int main() {
    int numpoints = 36;
    array double x[numpoints], y[numpoints];
    class CPlot plot;
    string_t title="Sine Wave",
             xlabel="degree",
             ylabel="amplitude";

    lindata(0, 360, x);
    y = sin(x*M_PI/180);
    plot.data2D(x, y);
    plot.title("Sine Wave");
    plot.label(PLOT_AXIS_X, xlabel);
    plot.label(PLOT_AXIS_Y, ylabel);
    plot.plotting();
}
```

プログラム 23.5: 注釈付きのプロットプログラム

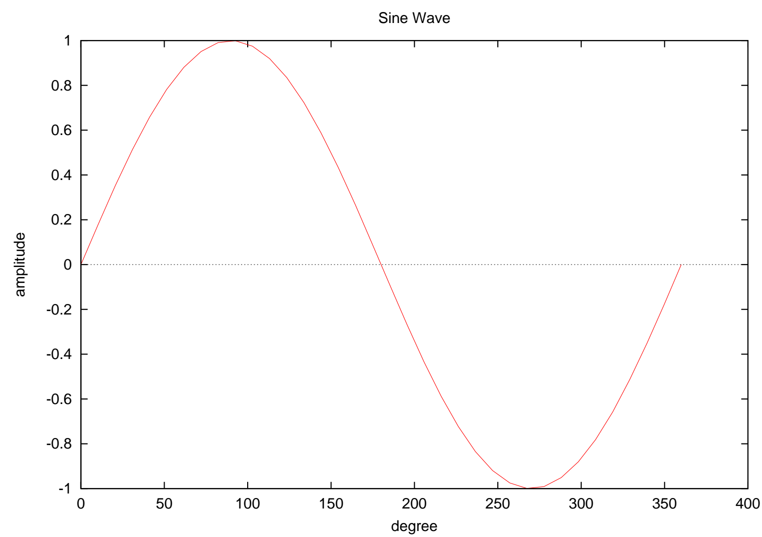


図 23.4: 注釈付きのプロット

プログラム 23.6は、矢、テキスト、軸の限界、グリッド、境界、および軸が、CPlot クラスのメンバ関数によって、どのように処理されるかを示します。

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

int main() {
    int numpoints = 36;
    array double x[numpoints], y[numpoints];
    class CPlot plot;
    string_t title="Sine Wave",
             xlabel="degree",
             ylabel="amplitude";
    double x1=180, y1=0.0,  z1=0;
    double x2=225, y2=0.1,  z2=0;

    lindata(0, 360, x);
    y = sin(x*M_PI/180);
    plot.data2D(x, y);
    plot.title("Sine Wave");
    plot.label(PLOT_AXIS_X, xlabel);
    plot.label(PLOT_AXIS_Y, ylabel);
    plot.axisRange(PLOT_AXIS_X, 0, 360);
    plot.ticsRange(PLOT_AXIS_X, 30, 0, 360);
    plot.axisRange(PLOT_AXIS_Y, -1, 1);
    plot.ticsRange(PLOT_AXIS_Y, .25, -1, 1);
    plot.axis(PLOT_AXIS_XY, PLOT_OFF);
    plot.border(PLOT_BORDER_ALL, PLOT_ON);
    plot.grid(PLOT_ON);
    plot.arrow(x1, y1, z1, x2, y2, z2);
    plot.text("inflection point", PLOT_TEXT_LEFT, x2, y2, z2);
    plot.plotting();
}
```

プログラム 23.6: 各種の機能を使用するプロットプログラム

図 23.5は、プログラム 23.6によって生成されたプロットを示します。

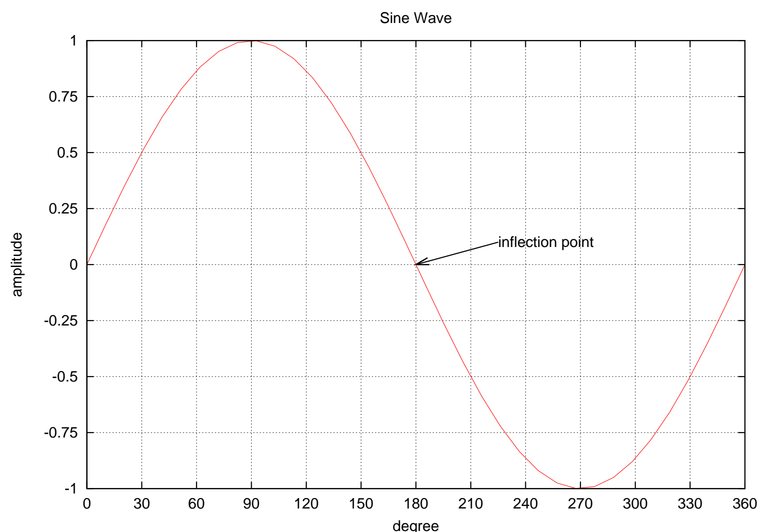


図 23.5: 各種の機能を使用するプロット

プログラム 23.6では、軸の限界は、メンバ関数 **CPlot::axisRange**(), によって以下のように設定されます。

```
void CPlot::axisRange(int axis, double minimum, double maximum);
```

軸に対して使用可能なマクロを、表 23.2に示します。4 つの境界 x (下部)、x2 (上部)、y (左)、および y2 (右) を、それぞれ独立した軸として使用できます。2 番目と 3 番目の引数は、それぞれ軸の最小値および最大値を指定します。次の関数は、軸のチックマークを設定します。

```
void CPlot::ticsRange(int axis, double incr, ...
                      /* [double start], [double end] */);
```

incr は、目盛りの間の増分を指定します。既定では、この値は内部処理されます。目盛りの開始位置と終了位置は、オプションの引数で指定します。たとえば、次の関数呼び出し

```
plot.axisRange(PLOT_AXIS_X, 0, 360);
plot.ticsRange(PLOT_AXIS_X, 30, 0, 360);
```

は、0〜360 度までの x 軸の範囲に、30 度刻みで目盛りを設定します。メンバ関数

```
void CPlot::axes(int num, char *axes);
```

により、次のように `num` で指定されたデータセットを、どの軸のペアに対してプロットするかを選択できます。利用可能な軸は 4 セットあります。引数 `axes` は、特定の線をスケーリングする軸を選択するために使用されます。文字列`"x1y1"`は、下部の軸と左側の軸を示します。`"x2y2"`は、上部の軸と右側の軸を示します。`"x1y2"`は、下部の軸と右側の軸を示します。`"x2y1"`は、上部の軸と左側の軸を示します。2 次元プロットにおける `x` 軸と `y` 軸の描画は、次のメンバ関数を使用して有効または無効にすることができます。

```
void CPlot::axis(int axis, int flag);
```

axis に指定可能なマクロは、その他のメンバ関数に指定可能なマクロと同じです。*flag* に **PLOT_ON** を設定すると、指定した軸の描画を有効にできます。**PLOT_OFF** を設定すると、指定した軸の描画を無効にできます。プログラム 23.5では、x 軸と y 軸の描画は、同時に関数呼び出し `plot.axis(PLOT_AXIS_XY, PLOT_OFF)` を使用することによって無効にされます。次のメンバ関数

```
void CPlot::border(int location, int flag);
```

は、プロット周囲の境界の表示のオンとオフを切り替えます。既定では、2 次元プロットの左側と下部に境界が描かれます。3 次元プロットでは、x-y 平面の四辺に境界が描かれます。関数 **CPlot**::**border**() の *location* に設定可能な値を表 23.3に示します。

表 23.3: 境界位置のためのマクロ

PLOT_BORDER_BOTTOM	プロットの下辺
PLOT_BORDER_LEFT	プロットの左側
PLOT_BORDER_TOP	プロットの上辺
PLOT_BORDER_RIGHT	プロットの右側
PLOT_BORDER_ALL	プロットのすべての境界線

図 23.5は、関数呼び出し `CPlot::border(PLOT_BORDER_ALL, PLOT_ON).` によって生成された四辺の境界を示します。次のメンバ関数は、x-y 平面におけるグリッドの表示を有効または無効にします。

```
void CPlot::grid(int flag, ... /* char *option */);
```

グリッドの表示を有効にするには *flag* に **PLOT_ON** を設定し、無効にするには **PLOT_OFF** を設定します。極プロットの場合、極座標グリッドが表示されます。それ以外の場合は、グリッドは長方形です。既定では、グリッドは表示されません。次の関数により、プロットに矢の注釈を付けることができます。

```
void CPlot::arrow(double x_head, y_head, z_head, x_tail, y_tail,
                  z_tail, ... /* char *option */);
```

(*x_head, y_head, z_head*) と (*x_tail, y_tail, z_tail*) は、それぞれ矢のヘッドとテールの座標を示します。矢は (*x_tail*, *y_tail*, *z_tail*) から (*x_head*, *y_head*, *z_head*) に向かって描かれます。これらの座標は、プロットの曲線と同じ座標系を使って指定されます。矢に関する他の属性を指定するには、オプションの引数を使用できます。次のメンバ関数は、プロットにテキストの注釈を追加します。

```
void CPlot::text(string_t string, int just,
                 double x, double y, double z);
```

テキスト *string* は、2次元プロットの場合は (*x*,*y*) の位置に、3次元プロットの場合は (*x*,*y*,*z*) の位置に配置されます。テキストの位置は、プロット座標系で測定されます。テキストの位置は、引数 *just* によって調整されます。表 23.4は、引数 *just* に指定可能なマクロを示します。

表 23.4: テキスト位置のためのマクロ

PLOT_TEXT_LEFT	テキスト文字列の左側
PLOT_TEXT_RIGHT	テキスト文字列の右側
PLOT_TEXT_CENTER	テキスト文字列の中心

図 23.5では、矢のテールは、関数 **CPlot::arrow()** および **CPlot::text()** を使って左揃えにされたテキスト `testing text` の位置にあります。

各種のチックマークやデータのスケールなどの追加機能については、**CPlot** クラスのリファレンスを参照してください。

23.1.3 複数のデータセットと凡例

図 23.6に示すように、複数のデータセットがあるプロットを生成できます。

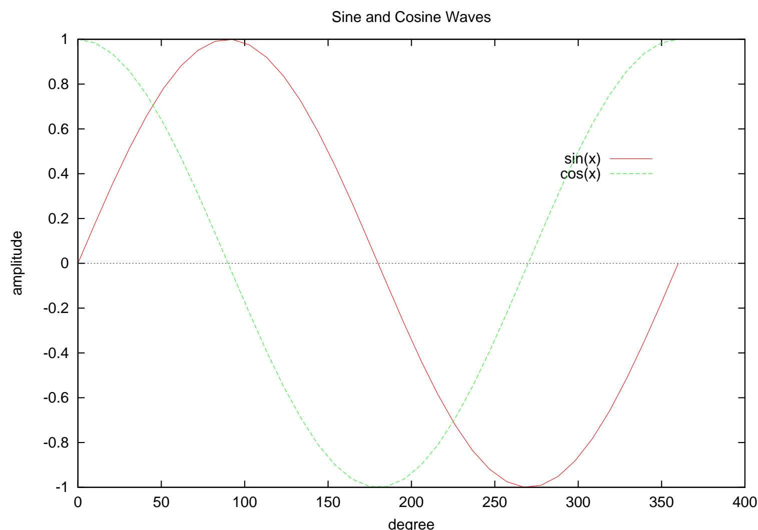


図 23.6: 2つのデータセット、タイトル、ラベル、凡例から成るプロット

プログラム 23.7または 23.8により、凡例付きの図 23.6を生成することができます。

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

int main() {
  int numpoints = 36;
  array double x1[numpoints], y1[numpoints];
  array double x2[numpoints], y2[numpoints];
  string_t title="Sine and Cosine Waves",
         xlabel="degree",
         ylabel="amplitude";
  class CPlot plot;

  lindata(0, 360, x1);
  lindata(0, 360, x2);
  y1 = sin(x1*M_PI/180);
  y2 = cos(x2*M_PI/180);
  plot.data2D(x1, y1);
  plot.data2D(x2, y2);
  plot.legend("sin(x)", 0);
  plot.legend("cos(x)", 1);
  plot.legendLocation(350, 0.5);
  plot.title(title);
  plot.label(PLOT_AXIS_X, xlabel);
  plot.label(PLOT_AXIS_Y, ylabel);
  plot.plotting();
}
```

プログラム 23.7: タイトル、ラベル、凡例を付けて、同じプロットに 2 つの関数をプロットするプログラム

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

int main() {
  int i, numdataset = 2, numpoints = 36;
  array double x[numpoints], y[numdataset][numpoints];
  string_t title="Sine and Cosine Waves",
         xlabel="degree",
         ylabel="amplitude";
  class CPlot plot;

  lindata(0, 360, x);
  for(i = 0; i < numpoints; i++) {
    y[0][i] = sin(x[i]*M_PI/180);
    y[1][i] = cos(x[i]*M_PI/180);
  }
  plot.data2D(x, y);
  plot.legend("sin(x)", 0);
  plot.legend("cos(x)", 1);
  plot.legendLocation(350, 0.5);
  plot.title(title);
  plot.label(PLOT_AXIS_X, xlabel);
  plot.label(PLOT_AXIS_Y, ylabel);
  plot.plotting();
}
```

プログラム 23.8: タイトル、ラベル、凡例を付けて、同じプロットに 2 つの関数をプロットする別のプログラム

プログラム 23.7は、プログラム 23.8と意味的に同じです。プログラム 23.8では、配列 y は 2 次元の 2 個のデータセットを示し、メンバ関数 **CPlot::data2D()** は、プロットにデータを追加するために 1 回だけ呼び出されます。次のメンバ関数は、プロットに凡例の文字列を追加します。

```
void CPlot::legend(string_t legend, int num);
```

2 番目の引数 num は、凡例が追加されるデータセットの数を示します。データセットの番号付けはゼロから始まります。新しい凡例は、以前に指定された凡例を置き換えます。このメンバ関数は、メンバ関数 **CPlot::data2D()**、**CPlot::data2DCurve()**、**CPlot::data3D()**、**CPlot::data3DCurve()**、**CPlot::data3DSurface()**、または
CPlot::dataFile() によって追加されたデータのプロット後に、呼び出す必要があります。次のメンバ関数

```
void CPlot::legendLocation(double x, double y, ... /* [double z] */);
```

は、プロット座標 *(x, y, z)* を使用してプロット凡例の位置を指定します。指定された位置は、図 23.7に示すように、凡例のマーカーとラベル用の矩形の右上隅です。

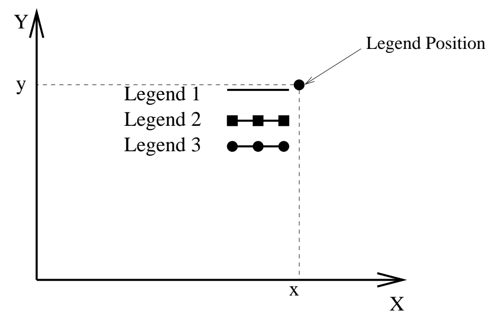


図 23.7: 凡例の位置

既定では、凡例の位置は、プロットの右上隅付近になります。同様に、複数のデータセットがある 3 次元プロットを生成できます。図 23.8の 3 次元プロットは、プログラム 23.9または23.10によって生成されます。

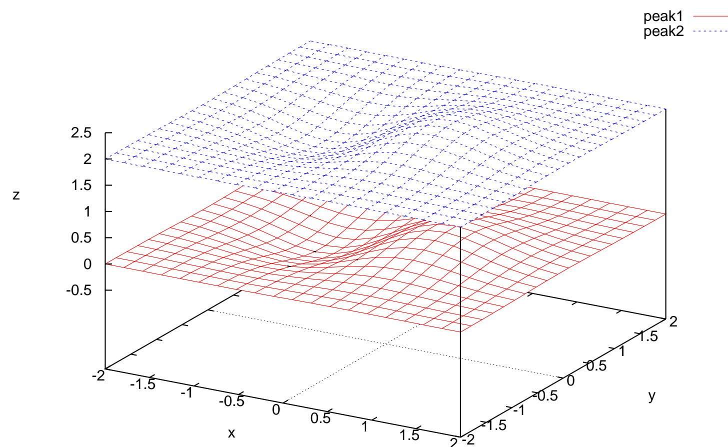


図 23.8: 2 つのデータセットを持つ 3 次元プロット

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

#define NUMX 20
#define NUMY 20
#define NUMCURVE 2
int main() {
    array double x[NUMX], y[NUMY], z[NUMCURVE][NUMX*NUMY];
    int datasetnum =0, i, j;
    class CPlot plot;

    lindata(-2, 2, x);
    lindata(-2, 2, y);
    for (i=0; i<NUMX; i++) {
        for(j=0; j<NUMY; j++) {
            z[0][i*NUMX+j] = x[i]*exp(-x[i]*x[i]-y[j]*y[j]);
            z[1][i*NUMX+j] = z[0][i*NUMX+j] +2;
        }
    }
    plot.data3D(x, y, z);
    plot.plotType(PLOT_PLOTTYPE_LINES, datasetnum++);
    plot.plotType(PLOT_PLOTTYPE_LINES, datasetnum);
    plot.legend("peak1", 0);
    plot.legend("peak2", 1);
    plot.colorBox(PLOT_OFF);
    plot.plotting();
}
```

プログラム 23.9: 同じ 3 次元プロット上に 2 つの関数をプロットするプログラム

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

#define NUMX 20
#define NUMY 20
#define NUMCURVE 2
int main() {
    array double x[NUMX], y[NUMY], z1[NUMX*NUMY], z2[NUMX*NUMY];
    int datasetnum =0, i, j;
    class CPlot plot;

    lindata(-2, 2, x);
    lindata(-2, 2, y);
    for (i=0; i<NUMX; i++) {
        for(j=0; j<NUMY; j++) {
            z1[i*NUMX+j] = x[i]*exp(-x[i]*x[i]-y[j]*y[j]);
            z2[i*NUMX+j] = z1[i*NUMX+j] +2;
        }
    }
    plot.data3D(x, y, z1);
    plot.plotType(PLOT_PLOTTYPE_LINES, datasetnum++);
    plot.data3D(x, y, z2);
    plot.plotType(PLOT_PLOTTYPE_LINES, datasetnum);
    plot.legend("peak1", 0);
    plot.legend("peak2", 1);
    plot.colorBox(PLOT_OFF);
    plot.plotting();
}
```

プログラム 23.10: 同じ 3 次元プロット上に 2 つの関数をプロットする別のプログラム

プログラム 23.9は、プログラム 23.10と意味的に同じです。プログラム 23.9では、配列 z は 2 次元の 2 個のデータセットを示します。メンバ関数 **CPlot**::**data3D**() は、プロットにデータを追加するために、プログラム 23.10のように 2 回呼び出されるのではなく、1 回だけ呼び出されます。

プログラム 23.11は、図 23.9に示す出力のサーフェスにどのように曲線を重ねるかを示します。非グリッドデータからは隠線を除去できないため、隠線消去はメンバ関数 **CPlot**::**removeHiddenLine**() によって無効にされます。

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

#define NUMX 20
#define NUMY 20
#define NUMCURVE 2
#define NUM  20
int main() {
    array double x[NUMX], y[NUMY], z[NUMCURVE][NUMX*NUMY];
    array double x0[NUM], y0[NUM], z0[NUM];
    int datasetnum=0, i, j, linetype, linewidth;
    class CPlot plot;

    lindata(-2, 2, x);
    lindata(-2, 2, y);
    lindata(-2, 2, x0);
    y0 = (array double [NUM])-1;
    for (i=0; i<NUMX; i++) {
        for(j=0; j<NUMY; j++) {
            z[0][i*NUMY+j] = x[i]*exp(-x[i]*x[i]-y[j]*y[j]);
            z[1][i*NUMY+j] = z[0][i*NUMY+j] +2;
        }
    }
    for (i=0; i<NUM; i++)
        z0[i] = x0[i]*exp(-x0[i]*x0[i]-y0[i]*y0[i]);
    plot.data3D(x, y, z);
    plot.plotType(PLOT_PLOTTYPE_LINES, datasetnum++);
    plot.plotType(PLOT_PLOTTYPE_LINES, datasetnum++);
    plot.data3D(x0, y0, z0);
    plot.legend("peak1", 0);
    plot.legend("peak2", 1);
    linetype = 5;
    linewidth = 2;
    plot.plotType(PLOT_PLOTTYPE_LINES, datasetnum);
    plot.lineType(datasetnum, linetype, linewidth);
    plot.legend("curve", datasetnum);
    plot.removeHiddenLine(PLOT_OFF);
    plot.colorBox(PLOT_OFF);
    plot.plotting();
}
```

プログラム 23.11: サーフェスに曲線を重ねるプログラム

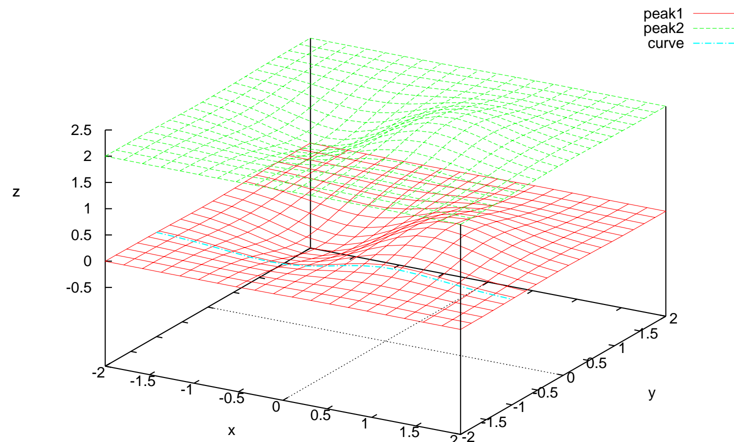


図 23.9: サーフェスの上に重ねられた曲線のある3次元プロット

23.1.4 事前定義済みの幾何プリミティブ

ユーザーの便宜を図るために、行、円、長方形、多角形などのいくつかの幾何プリミティブは、プロット対象クラスのメンバ関数として事前に定義されています。次の関数を使用して、線を描くことができます。

```
int CPlot::line(double x1, double y1, double z1, double x2,
                double y2, double z2);
```

2次元長方形プロットおよび3次元直交プロットの場合、(*x1*, *y1*, *z1*) と (*x2*, *y2*, *z2*) は、x軸、y軸、z軸の単位で指定された線の終点の座標です。ただし、2次元プロットでは、*z1* と *z2* は無視されます。2次元極プロットおよび3次元円筒プロットの場合、極座標で指定された終端は、引数 *x* に θ、引数 *y* に r を指定します。*z* は不変です。ただし、2次元プロットでは、*z1* と *z2* は無視されます。球座標を使用する3次元プロットの場合、引数 *x* に θ、引数 *y* に ϕ、引数 *z* に r を指定します。次の関数

```
int CPlot::circle(double x, double y, double r);
```

は、2次元プロットに円を追加します。長方形プロットの場合、*x* と *y* は円の中心の座標で、*r* は円の半径です。*x*、*y*、*r* はすべて *x* 軸と *y* 軸の単位で指定されます。極プロットの場合、円の中心の位置は極座標 (r, θ) で指定され、引数 *x* に θ、引数 *y* に r を指定します。次の関数

```
int CPlot::rectangle(double x, double y,
                     double width, double height);
```

は、2次元プロットに長方形を追加します。長方形プロットの場合、*x* と *y* は長方形の左下隅の座標を示します。極プロットの場合、左下隅の座標は、引数 *x* に θ、引数 *y* に r を指定した極座標 (r, θ) で指定されます。どちらの場合も、*width* と *height* には、長方形の座標で表した長方形の次元を指定します。次の関数

```
int CPlot::polygon(double x[:], double y[:], double z[:]);
```

は、プロットに多角形を追加します。2次元の長方形プロットと3次元の直交プロットの場合、*x*、*y*、*z*は、x軸、y軸、z軸の単位で指定された多角形の頂点を示します。ただし、2次元プロットでは*z*は無視されます。2次元の極プロットおよび3次元の円筒プロットの場合、極座標で指定された頂点の位置は、引数*x*にθ、引数*y*にrを指定した極座標(r, θ)で指定されます。*z*は不変です。ここでも、2次元プロットでは、*z*は無視されます。球面座標を使用する3次元プロットの場合、引数*x*にθ、引数*y*にϕ、引数*z*にrを指定します。それぞれのデータ点は、隣のデータ点に閉鎖的に接続されます。

これらのメンバ関数によって追加された幾何プリミティブは、**CPlot::legend**()および**CPlot::plotType**()に対する後からの呼び出しのためのデータセットとして見なされます。例として、セクション12.2.2に説明がある関数$f(x) = e^{\frac{1}{x}}$は、プログラム23.12で生成された271ページの図12.1に示すように、原点で連続していません。

```
/* expx.ch */
#include <chplot.h>

int main() {
    array double x1[100], f1[100];
    array double x2[100], f2[100];
    CPlot plot;

    lindata(0.5, 10, x1);
    f1=exp((array double [100])1.0./x1); // f1=exp(1.0./x1);
    lindata(x2, -10, -0.5);
    f2=exp((array double [100])1 ./x2); // f1=exp(1 ./x1);
    plot.label(PLOT_AXIS_X, "x");
    plot.label(PLOT_AXIS_Y, "exp(1/x)");
    plot.data2D(x1, f1);
    plot.data2D(x2, f2);
    plot.line(-10, 1, 0, 10, 1, 0);
    plot.plotType(PLOT_PLOTTYPE_LINES, 2, 3, 0);
    plot.line(0, 0, 0, 0, 8, 0);
    plot.plotType(PLOT_PLOTTYPE_LINES, 3, 3, 0);
    plot.plotting();
    return 0;
}
```

プログラム 23.12: プロット関数$e^{1/x}$.

プログラムは、関数$f(x) = e^{\frac{1}{x}}$のために2つの曲線を描きます。これらの2つの曲線のデータセットは、計算配列型の引数が代入された、型汎用数学関数**exp**()を使用して計算されます。点(0, 1)で交差する水平と垂直の2本の線は、幾何プリミティブメンバ関数**CPlot::line**()を使用して描かれます。線の種類は、メンバ関数**CPlot::plotType**()によって指定されます。

23.1.5　サブプロット

次の関数を使用することで、複数のプロットを同じ画面に表示して、同じ紙に出力できます。

```
int CPlot::subplot(int row, int col);
```

関数 **CPlot**::**subplot**() は、図を、`m` × `n` の行列で示される小さなサブプロットに分割します。これらのサブプロットは、引数で指定した行数と列数から成る 2 次元の行列に見立てられ、番号が振られます。各インデックスは 0 から始まります。次の関数は、位置 `(i、j)` のサブプロットのためのハンドルとして、**CPlot** クラスへのポインタを取得できます。

```
class CPlot* CPlot::getSubplot(int row, int col);
```

row および col はそれぞれ、目的のサブプロット要素の行数と列数を示します。番号はゼロから始まります。たとえば、プログラム 23.13は、2 × 2 の行列から成る 4 つのサブプロットにプロットを分割します。

```
#include <float.h>
#include <math.h>
#include <chplot.h>

#define NUM1 36
#define NUM2 101
#define NUMX 20
#define NUMY 30
int main() {
    array double x[NUM1], y[NUM1];
    double x3[NUMX], y3[NUMY], z3[NUMX*NUMY], r;
    array double x4[NUM2], y4[NUM2];
    int i, j;
    class CPlot subplot, *plot;

    lindata(-M_PI, M_PI, x);
    y = sin(x);
    subplot.subplot(2, 2);
    plot = subplot.getSubplot(0, 0);
    plot->data2D(x, y);

    plot = subplot.getSubplot(0, 1);
    plot->data2D(x, y);
    plot->axisRange(PLOT_AXIS_Y, -1, 1);
    plot->ticsRange(PLOT_AXIS_Y, 0.25, -1, 1);
    plot->grid(PLOT_ON);

    lindata(-20, 20, x4);
    x4 = x4+(x4==0)*DBL_EPSILON; /* if x4==0, x4 becomes epsilon */
    y4 = sin(x4)./(x4);
    plot = subplot.getSubplot(1, 0);
    plot->data2D(x4, y4);
    plot->label(PLOT_AXIS_Y, "sin(x)/x");

    lindata(-10, 10, x3);
    lindata(-10, 10, y3);
    for(i=0; i<NUMX; i++) {
        for(j=0; j<NUMY; j++) {
            r = sqrt(x3[i]*x3[i]+y3[j]*y3[j]);
            z3[NUMY*i+j] = sin(r)/r;
        }
    }
    plot = subplot.getSubplot(1, 1);
    plot->data3D(x3, y3, z3);
    plot->colorBox(PLOT_OFF);

    subplot.plotting();
}
```

プログラム 23.13: サブプロットを作成するプロットプログラム

各サブプロットには、それぞれが個別のプロットであるかのように、タイトル、ラベルなどの注釈を追加できます。図 23.10は、プログラム 23.13によって生成されたプロットを示します。

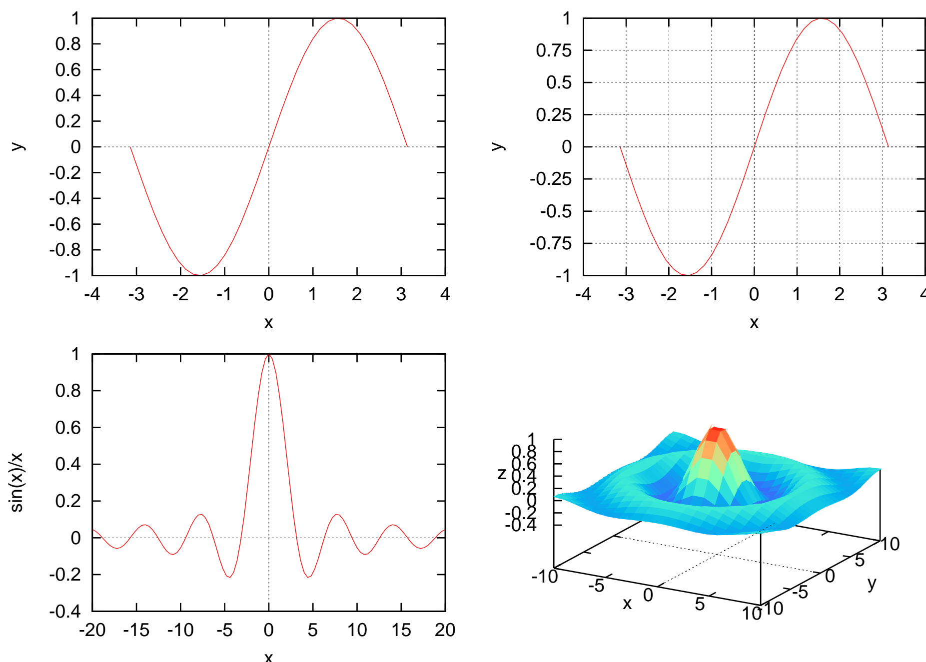


図 23.10: サブプロットのあるプロット

プログラム 23.13では、for ループや while ループなどの繰り返しのループを使用せずに、関数 $\sin(x)/(x)$ のデータを (1, 1) に配置されるサブプロットとしてプロットが生成されます。2 つの配列の要素単位の分割のための配列演算子`./`が、このプログラムに適用されます。ゼロ除算を避けるために、小さい浮動小数点値 `DBL_EPSILON` が使用されます。

23.1.6　プロットのエクスポートとズーム

プロットは、端末の画面に表示できるだけでなく、各種のアプリケーションに合わせてさまざまな形式でエクスポートすることができます。次のメンバ関数を使用して、各種の形式による出力を行うことができます。

```
void CPlot::outputType(int outputtype, ...,
                       /* [string_t terminal, string_t filename] */);
```

引数 *outputtype* には、マクロ **PLOT_OUTPUTTYPE_DISPLAY**、**PLOT_OUTPUTTYPE_STREAM**、**PLOT_OUTPUTTYPE_FILE**、のいずれかを指定できます。出力の種類 **PLOT_OUTPUTTYPE_DISPLAY** は、プロットを画面上に表示します。プロットは、そのプロット専用の個別のウィンドウに表示されます。UNIX で q キーを押すことによって、プロットウィンドウを閉じることができます。既定の出力の種類は、**PLOT_OUTPUTTYPE_DISPLAY** です。

出力の種類が **PLOT_OUTPUTTYPE_STREAM** である場合、プロットエンジンからの出力は、標準の出力ストリームです。**PLOT_OUTPUTTYPE_STREAM** は、Web ブラウザ表示で、Ch プログラムを Web サーバーで CGI スクリプトとして使用し、png または gif ファイル形式の標準の出力ストリームとしてプロットを動的に生成する場合に役立ちます。

PLOT_OUTPUTTYPE_FILE を指定した場合、2 つのオプション引数 *terminal* と *filename* を指定することで、プロットを各種のファイル形式で保存できます。サポートされる端末の種類を表 23.5に示します。一部の端末では、引数 *terminal* の文字列の一部として、プロットのサイズや色を決める追加のパラメータを指定できます。各端末の詳細については、『*Ch* **言語環境リファレンスガイド**』のプロットの章を参照してください。

表 23.5: プロット出力タイプ別の端末の種類

端末	説明
aifm	Adobe Illustrator 3.0.
corel	CorelDRAW 用の EPS ファイル
dxf	AutoCAD DXF.
dxy800a	Roland DXY800A プロッタ
eepic	Extended LaTeX 拡張
emtex	emTeX 特殊文字対応 LaTeX 拡張
epson-180dpi	Epson LQ 型 24-pin プリンタ 180dpi
epson-60dpi	Epson LQ 型 24-pin プリンタ 60dpi
epson-lx800	Epson LX-800、Star NL-10、および NX-100
excl	Talaris プリンタ
fig	Xfig 3.1
gif	GIF ファイル形式
gpic	gpic/groff パッケージ
hp2648	HP2647 および HP2648
hp500c	Hewlett Packard DeskJet 500c
hpdj	Hewlett Packard DeskJet 500
hpgl	HPGL 出力
hpljii	HP LaserJet II.
hppj	HP PaintJet および HP3630 プリンタ
latex	LaTeXpicture
mf	MetaFont
mif	Frame Maker MIF 3.00
nec-cp6	NEC CP6 および Epson LQ-800.
okidata	OKIDATA 320/321 9 ピンプリンタ
pcl5	Hewlett Packard LaserJet III.
pbm	Portable BitMap.
png	Portable Network Graphics.
postscript	Postscript.
pslatex	postscript 特殊文字対応 LaTeXpicture
pstricks	PSTricks マクロ対応 LaTeXpicture
starc	Star カラープリンタ
tandy-60dpi	Tandy DMP-130 シリーズプリンタ
texdraw	LaTeXtexdraw 形式
tgif	TGIF X-Window 描画形式
tpic	tpic 特殊文字対応 LaTeXpicture

最後のオプション引数 *filename* は、プロットが保存されるファイル名を含む文字列です。パイプをサポートするコンピュータでは、コマンド名の前に記号 “`postscript`”を追加し、コマンド名全体

をファイル名として使用することで、出力を別のプログラムにパイプで渡すことができます。たとえば、UNIX システムでは、*terminal* に“*postscript*”を指定し、*filename* に“`|lp`”を指定すると、プロットを直接ポストスクリプトプリンタに送信できます。

プログラム 23.14は、EPS、latex、pbm、gif、png の各形式でプロットをエクスポートする方法を示します。プログラム 23.14の 2 次元プロット関数 **plotxy()** に関する詳細な説明は、セクション 23.2.3を参照してください。

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

int main() {
  int numpoints = 36;
  array double x[numpoints], y[numpoints];
  string_t title="Sine Wave",                      // Define labels.
         xlabel="degree",
         ylabel="amplitude";
  class CPlot plot;

  lindata(0, 360, x);
  y = sin(x*M_PI/180);
  plotxy(x,y,title,xlabel,ylabel,&plot);

  /* create a postscript file */
  plot.outputType(PLOT_OUTPUTTYPE_FILE, "postscript eps color", "demo.eps");
  plot.plotting();

  /* create a latex file */
  plot.outputType(PLOT_OUTPUTTYPE_FILE, "latex roman 11", "demo.tex");
  plot.plotting();

  /* create a pbm file */
  plot.outputType(PLOT_OUTPUTTYPE_FILE, "pbm", "demo.pbm");
  plot.plotting();

  /* create a gif file */
  plot.outputType(PLOT_OUTPUTTYPE_FILE, "gif", "demo.gif");
  plot.plotting();

  /* create a png file */
  plot.outputType(PLOT_OUTPUTTYPE_FILE, "png", "demo.png");
  plot.plotting();
}
```

プログラム 23.14: プロットをエクスポートするプログラム

Windows では、プロットの左上隅にあるメニューを使用して、画面に表示されたプロットをクリップボードにコピーできます。その後、Word (Microsoft のワードプロセッサプログラム) などの他の Windows アプリケーションプログラムに、コピーしたプロットを貼り付けることができます。プロットの一部をズームするには、マウスを右クリックして左上隅を選択し、マウスを右下隅にドラッグしてから右クリックして、その領域を選択し、ズームインします。ズームイン後、‘p’ キーを押すとズームアウトし、‘u’ キーを押すと元のプロットに戻ります。

23.1.7　プロットの出力

Windows でのプロットの出力

Windows にプロットを出力するには、以下の 2 つの方法のいずれかを使用できます。
方法 1
手順 1. プロット機能を持つ Ch プログラムを実行し、プロットが表示されているウィンドウの左上隅をクリックします。
手順 2. [オプション] メニューから [印刷] を選択し、構成を行います。構成に従って出力します。

方法 2 手順 1. プロットで Ch プログラムを実行し、プロットが表示されているウィンドウの左上隅をクリックします。
手順 2. オプションメニューから [クリップボードへのコピー] を選択します。
手順 3. [スタート] ボタンをクリックし、[すべてのプログラム]、[アクセサリ]、[ペイント] の順にクリックします。
手順 4. [編集] メニューの [貼り付け] をクリックするか、または`<Ctrl><V>`キー押してプロットを貼り付けます。
手順 5. bmp ファイルとしてプロットを保存します。
手順 6. プロットを出力します。

UNIX でのプロットの出力

UNIX では、まず、前のセクションで説明した端末の種類に従ってプロットを保存します。次にそれを出力します。たとえば、ポストスクリプトプリンタの場合、プロットを、まず関数 `plot.outputType(PLOT_OUTPUTTYPE_FILE,"postscript eps color", "filename.eps")` を使用して filename.eps というファイル名でカラーの eps ファイルとして保存します。次に、コマンド **lp** を使用して、postscript ファイル **filename.eps** を出力します。または、関数呼び出し `plot.outputType(PLOT_OUTPUTTYPE_FILE, "postscript eps color","| lp").` を使用して、プロットの出力の種類を設定することによって、プロットを出力することもできます。

23.2　2 次元プロット

前のセクションで説明した機能は、2 次元および 3 次元プロットのどちらでも使用できます。このセクションでは 2 次元プロットに特有の機能について説明します。

23.2.1　プロットの種類、線のスタイル、マーカー

次の関数を使用して、さまざまなプロットの種類を選択できます。

```
void CPlot::plotType(int plot_type, int num, ...
     /* [string_t option]*/);
```

関数 **CPlot**::**plotType**() は、プロット対象のデータセットに、目的のプロットの種類を設定します。引数 *plot_type* の有効なマクロを表 23.6に、マクロに対応するプロットを図 23.11に示します。

表 23.6: 各種 2 次元プロットのマクロ

PLOT_PLOTTYPE_BOXERRORBARS	プロットの種類 **PLOT_PLOTTYPE_BOXES** と **PLOT_PLOTTYPE_YERRORBARS** の組み合わせです。
PLOT_PLOTTYPE_BOXES	指定された x 座標を中心にボックスを描画します。
PLOT_PLOTTYPE_BOXXYERRORBARS	プロットの種類 **PLOT_PLOTTYPE_BOXES** と **PLOT_PLOTTYPE_XYERRORBARS** の組み合わせです。
PLOT_PLOTTYPE_CANDLESTICKS	財務データまたは統計データの箱ひげ図を表示します。
PLOT_PLOTTYPE_DOTS	各データ点をマークするためにドットを使用します。
PLOT_PLOTTYPE_FILLEDCURVES	曲線で囲まれた領域を、単色またはパターンで塗りつぶします。
PLOT_PLOTTYPE_FINANCEBARS	財務データを表示します。
PLOT_PLOTTYPE_FSTEPS	隣接しているデータ点は、(x1,y1) から (x1,y2) へと、(x1,y2) から (x2,y2) への 2 つの線分で接続されます。
PLOT_PLOTTYPE_HISTEPS	データ点 x1 は、((x0+x1)/2,y1) から ((x1+x2)/2,y1) への水平線によって表されます。 ((x1+x2)/,y1) から ((x1+x2)/2,y2) までの隣接している行は、縦線で連結されます。
PLOT_PLOTTYPE_IMPULSES	x 軸 (2 次元プロット用) または x-y 平面 (3D プロット用) からデータ点までの縦線を表示します。
PLOT_PLOTTYPE_LINES	データ点は、線で接続されます。
PLOT_PLOTTYPE_LINESPOINTS	各データ点にマーカーを表示し、線で接続します。
PLOT_PLOTTYPE_POINTS	各データ点にマーカーを表示します。
PLOT_PLOTTYPE_STEPS	隣接しているデータ点は、(x1,y1) から (x2,y1) へと、(x2,y1) から (x2,y2) への 2 つの線分で接続されます。
PLOT_PLOTTYPE_VECTORS	ベクトルを表示します。
PLOT_PLOTTYPE_XERRORBARS	水平の誤差指示線がある点線を表示します。
PLOT_PLOTTYPE_XERRORLINES	水平の誤差線がある折れ線と点を表示します。
PLOT_PLOTTYPE_XYERRORBARS	水平および垂直の誤差指示線がある点線を表示します。
PLOT_PLOTTYPE_XYERRORLINES	水平および垂直の誤差線がある折れ線と点を表示します。
PLOT_PLOTTYPE_YERRORBARS	垂直の誤差指示線がある点線を表示します。
PLOT_PLOTTYPE_YERRORLINES	垂直の誤差線がある折れ線と点を表示します。

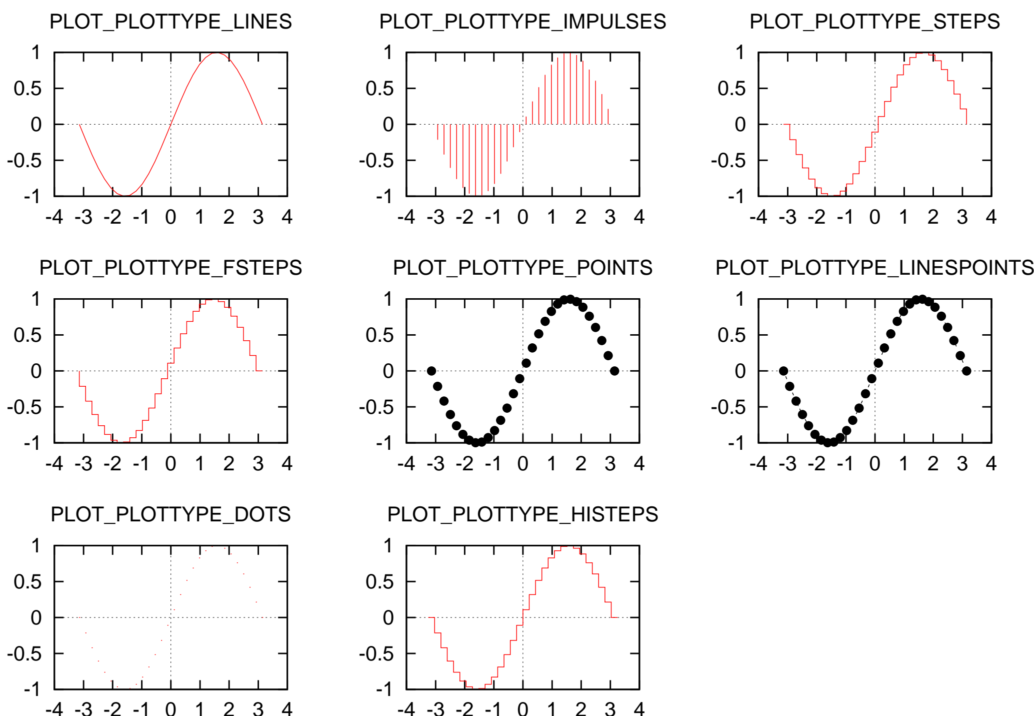


図 23.11: 各種の 2 次元プロット

既定では、2 次元プロットは **PLOT_PLOTTYPE_LINES** というプロットの種類を使用します。同じプロットのデータセットに、別々のプロットの種類を設定することができます。引数 *num* は、プロットの種類が適用されているデータセットを示します。データセットの番号付けはゼロから始まります。新しく設定されたプロットの種類は、以前に指定された種類を上書きします。

線の種類、色、縦線、ステップなどは関数

```
void lineType(int num, int line_type, int line_width, ...
              /* [char *line_color] */);
```

で設定することができます。関数 **CPlot::lineType()** はプロットされるデータセットに対する線のスタイルを設定します。プロットの線のスタイルおよびマーカータイプは、自動的に選択されます。*line_type* は、線の描画に使用される線の種類のインデックスを指定します。関数 CPlot::outputType() の説明にあるとおり、線の種類は、使用される端末の種類に依存します。通常は、プロットが表示される際に線の種類を変更すると、線の色が変わります。線の種類を変更すると、プロットがポストスクリプトファイルとして保存される際に、破線、点線または他の形状になります。すべての端末で、少なくとも 6 つの線種がサポートされます。既定の線種は 1 です。*line_width* は線幅を指定します。*line_width* に既定幅を掛けた値が線幅となります。既定幅は通常、1 ピクセルです。オプションの 4 番目の引数は、色の名前または RGB 値で線の色を、青色に対して `"blue"` または `"#0000ff"` のように指定可能です。

プログラム 23.15は、異なった線種がどのように使用されるかを示します。Windows に表示されるプロットは、図 23.12に示されます。

```
#include <chplot.h>

int main() {
    double x, y, xx[2], yy[2];
    string_t text;
    int line_type = -1, line_width = 2, datasetnum = 0;
    class CPlot plot;

    plot.axisRange(PLOT_AXIS_X, 0, 5);
    plot.axisRange(PLOT_AXIS_Y, 0, 4);
    plot.ticsRange(PLOT_AXIS_Y, 1, 0, 4);
    plot.title("Line Types in Ch Plot");
    for (y = 3; y >= 1; y--) {
        for (x = 1; x <= 4; x++) {
            sprintf(text, "%d", line_type);
            lindata(x, x+.5, xx);
            lindata(y, y, yy);
            plot.data2D(xx, yy);
            plot.plotType(PLOT_PLOTTYPE_LINES, datasetnum);
            plot.lineType(datasetnum, line_type, line_width);
            plot.text(text, PLOT_TEXT_RIGHT, x-.125, y, 0);
            datasetnum++;
            line_type++;
         }
    }
    plot.plotting();
}
```

プログラム 23.15: さまざまな線種のためのプロットプログラム

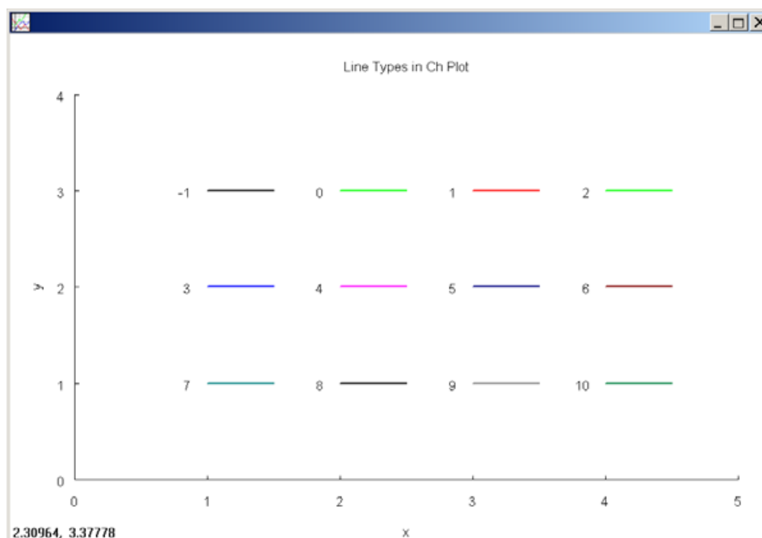


図 23.12: Windows に表示された、プログラム内で線の色を指定されたプロット

線種には通常、色が関連付けられています。プログラム 23.16は、プログラム内で線の色を指定する方法を示します。

図 23.13は、postscript ファイル形式で生成されたプロットを示します。

```
/* File: color1.ch */
#include <chplot.h>

/* colors of lines for displayed plot */
char *color[] = {
  "black",
  "white",
  "grey",
  "grey40",
  "grey60",
  "red",
  "yellow",
  "green",
  "blue",
  "navy",
  "cyan",
  "magenta",
  "orange",
  "gold",
  "brown",
  "purple",
};


int main() {
    double x[2], y[2];
    int i, line_type= 1, line_width = 1, datasetnum = 0, n;
    CPlot plot;

    plot.title("Line Colors in Ch Plot");
    n = sizeof(color)/sizeof(color[0]);
    y[0] = 0; y[1] = 1;
    for (i = 0; i < n; i++) {
       x[0] = i+1; x[1] = i+1;
       plot.data2D(x, y);
       plot.plotType(PLOT_PLOTTYPE_LINES, datasetnum);
       plot.lineType(datasetnum, line_type, line_width, color[i]);
       datasetnum++;
    }
    /* color of the horizontal line added below is green */
    x[0] = 1; x[1] = 15;
    y[0] = 0.5; y[1] = .5;
    plot.data2D(x, y);
    plot.plotType(PLOT_PLOTTYPE_LINES, datasetnum);
    plot.lineType(datasetnum, line_type, line_width, "green");
    plot.plotting();
}
```

プログラム 23.16: プログラム内で曲線の色を指定する例

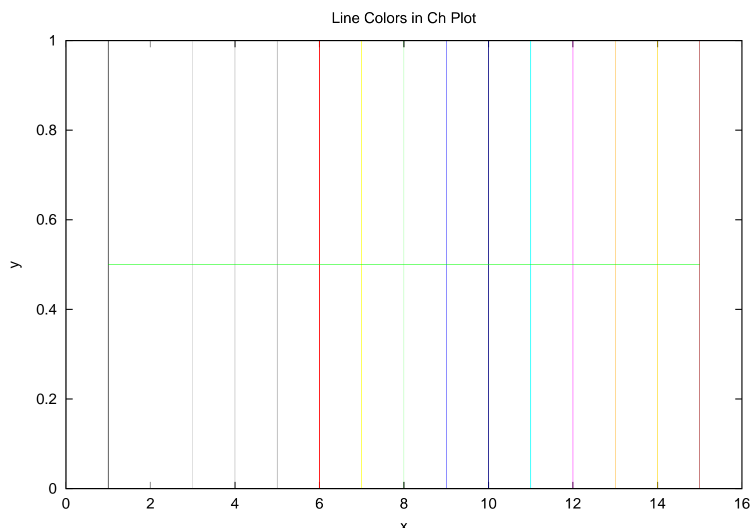


図 23.13: postscript ファイルに保存した、プログラム内で色を指定したプロット

プログラム 23.17は、クラスの同じインスタンスを使用して複数のプロットを生成する方法を示します。プログラム 23.17の実行時には、最初に、赤い曲線のあるプロットが表示されます。次に、青い曲線のあるプロットが表示されます。そして、別の赤い曲線を重ねることで、この曲線の色を変更します。最後に重ねた色が優先されます。最後の色は、実行時にプログラムで動的に決定できます。最終的に、マゼンタ色の新しい曲線がプロットに追加されます。図 23.14は、プログラム 23.17によって生成された 4 つのプロットをそれぞれ示します。

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

int main() {
    array double x[36], y[36];
    int line_type = 1, line_width = 1, datasetnum = 0;
    CPlot plot;

    lindata(-M_PI, M_PI, x);
    y = sin(x);

    plot.data2D(x, y);
    plot.plotType(PLOT_PLOTTYPE_LINES, datasetnum);
    plot.lineType(datasetnum, line_type, line_width, "red");
    plot.legend("red line", datasetnum);
    plot.plotting();

    /* change the color of the curve from the same data set to blue  */
    plot.lineType(datasetnum, line_type, line_width, "blue");
    plot.legend("blue line", datasetnum);
    plot.plotting();

    /* overlaying blue curve with red curve .*/
    plot.data2D(x, y);
    datasetnum++;
    plot.lineType(datasetnum, line_type, line_width, "red");
    plot.legend("red line", datasetnum);
    plot.plotting();

    /* add a new curve with color of magenta to the plot */
    y = sin(x)+0.5;
    plot.data2D(x, y);
    datasetnum++;
    plot.lineType(datasetnum, line_type, line_width, "magenta");
    plot.legend("magenta line", datasetnum);
    plot.plotting();
}
```

プログラム 23.17: さまざまな色の新しい曲線を重ねることで、曲線の色を変更する例

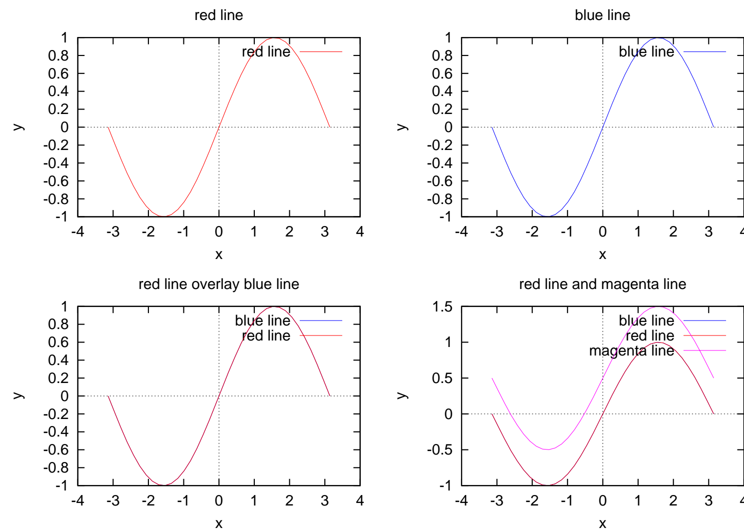


図 23.14: 新しい曲線を重ねることによって曲線の色を変更したプロット

点の種類、サイズ、および色付きの点は、関数

```
void pointType(int num, int point_type, int point_size, ...
               /* [char *point_color] */);
```

により設定することができます。

関数 **CPlot::pointType()** はプロットするデータセットに対する点の種類を設定します。*point_type* は、点の描画に対して、その点の種類のインデックスを指定します。点の種類は、使用される端末の種類に依存して変わります（**CPlot::outputType** を参照）。*point_type* の値は、データ点の外観 (色およびマーカーの種類) を変更することに使用されます。

この値には、目的のデータ点の種類のインデックスを示す整数を指定します。すべての端末で、少なくとも 6 つの線種がサポートされます。

point_size はオプションの引数で、データ点のサイズの変更に使用されます。*point_size* は、既定のサイズで乗算されたデータ点サイズです。*point_type* と *point_size* にゼロまたは負の数が設定されている場合、既定値が使用されます。

オプションの 4 番目の引数は、色の名前または RGB 値で線の色を、青色に対して `"blue"` または `"#0000ff"` のように指定可能です。

プログラム 23.18は、さまざまなデータ点の種類がどのように使用されるかを示します。図 23.15に、Windows に表示されたプロットを示します。

```
#include <chplot.h>

int main() {
    double x, y;
    string_t text;
    int datasetnum=0, point_type = 1, point_size = 5;
    class CPlot plot;

    plot.axisRange(PLOT_AXIS_X, 0, 7);
    plot.axisRange(PLOT_AXIS_Y, 0, 5);
    plot.title("Point Types in Ch Plot");
    for (y = 4; y >= 1; y--) {
        for (x = 1; x <= 6; x++) {
            sprintf(text, "%d", point_type);
            plot.point(x, y, 0);
            plot.plotType(PLOT_PLOTTYPE_POINTS, datasetnum);
            plot.pointType(datasetnum,  point_type, point_size);
            plot.text(text, PLOT_TEXT_RIGHT, x-.25, y, 0);
            datasetnum++; point_type++;
        }
    }
    plot.plotting();
}
```

プログラム 23.18: 各種のデータ点のためのプロットプログラム

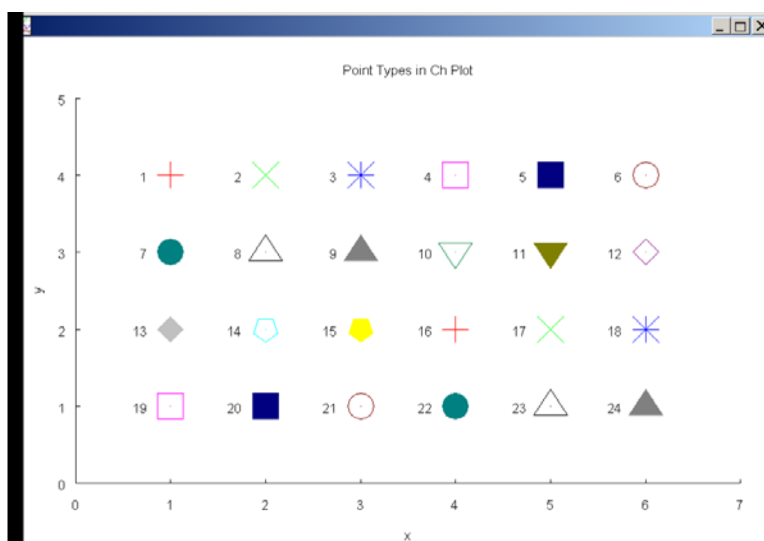


図 23.15: Windows に表示されたさまざまな線種によるプロット

図 23.16では、同じプロットに2つのデータセット (線とデータ点) を表示します。プログラム 23.19に、この図を生成するソースコードを示します。

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

int main() {
    array float x1[75], y1[75];
    array float x2[300], y2[300];
    class CPlot plot;
    int numdataset=0, pointtype =1, pointsize=1,
                      linetype =3,  linesize=1;

    lindata(-2*M_PI, 2*M_PI, x);
    lindata(-2*M_PI, 2*M_PI, x2);
    y1 = x1.*x1+5*sin(10*x1);
    y2 = x2.*x2+5*sin(10*x2);
    plot.data2D(x1, y1);
    plot.data2D(x2, y2);
    plot.plotType(PLOT_PLOTTYPE_POINTS, numdataset);
    plot.pointType(numdataset, pointtype, pointsize);
    numdataset++;
    plot.plotType(PLOT_PLOTTYPE_LINES, numdataset);
    plot.lineType(numdataset, linetype, linesize);
    plot.plotting();
}
```

プログラム 23.19: さまざまな線種のためのプロットプログラム

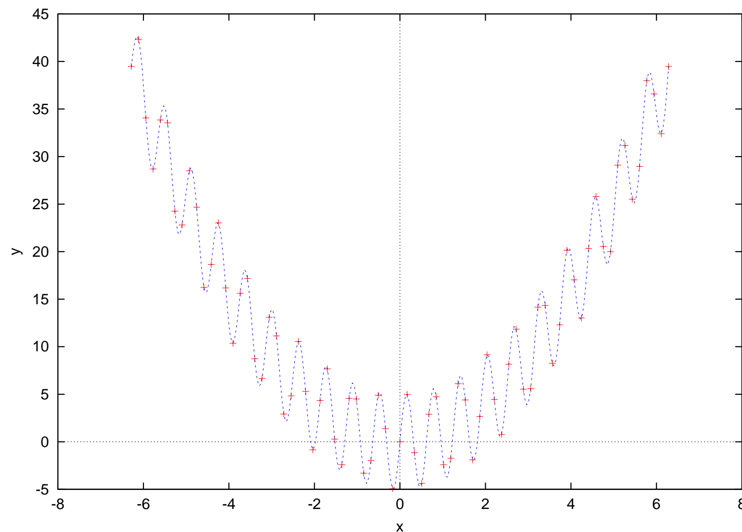


図 23.16: データ点と線によるプロット

23.2.2 極座標

極座標系の 2 次元プロットは、次のメンバ関数を使用して指定できます。

```
void CPlot::polarPlot(int angle_unit);
```

引数 *angle_unit* には、角度位置の測定単位を指定します。角度の単位を度で表す **PLOT_ANGLE_DEG** マクロまたはラジアンで表す **PLOT_ANGLE_RAD** マクロを指定できます。プログラム 23.20に示すように、極座標系 (r,θ) では、メンバ関数呼び出し `plot.data2D(theta, r)` の先頭および 2 番目の配列引数はそれぞれ、プロット対象の位相角とデータ点の大きさです。

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

int main() {
    int numpoints = 360;
    array double theta[numpoints], r[numpoints];
    class CPlot plot;

    lindata(0, M_PI, theta);
    r = sin(5*theta);
    plot.polarPlot(PLOT_ANGLE_RAD);
    plot.data2D(theta, r);
    plot.sizeRatio(1);
    plot.grid(PLOT_ON);
    plot.plotting();
}
```

プログラム 23.20: 極座標系を使用するプロットプログラム

図 23.17に示す極グリッドは、関数呼び出し `plot.grid(PLOT_ON)` によって作成されます。

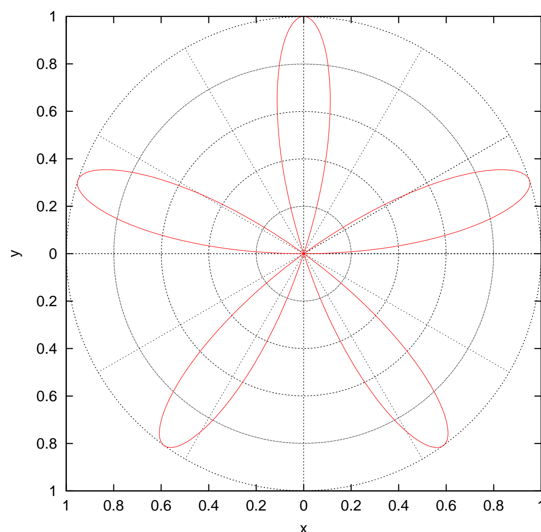


図 23.17: 極座標系のプロット

次のメンバ関数は、プロットのアスペクト比を設定できます。

```
void CPlot::sizeRatio(float ratio);
```

ratio に設定された値によって、比率の意味は変化します。*ratio* に正の値を設定した場合、x 軸の長さに対する y 軸の長さの比率を表します。*ratio* が 2 の場合、y 軸は x 軸の 2 倍の長さになります。*ratio* にゼロを設定した場合、端末の種類に設定された既定のアスペクト比が使用されます。*ratio* に負の値を設定した場合、x 軸単位に対する y 軸単位の比率を表します。比率が-2 の場合、y 軸の 1 単位は、x 軸上の 1 単位の 2 倍の長さになります。極プロットの場合、プログラム23.20に示すように、アスペクト比を 1 に設定する必要があります。

23.2.3 2 次元プロット関数

高度なプロット関数 **fplotxy**()、**plotxy**()、および **plotxyf**() は、使いやすく、2 次元プロットでの使用に便利です。**CPlot** メンバ関数と共にこれらの関数を使用して、より高度なプロットを作成できます。

プロット関数 **plotxy**() のプロトタイプは、次のとおりです。

```
int plotxy(double x[&], array double &y, ...
  /* [string_t title, xlabel, ylabel], [class CPlot *plot] */);
```

配列 x と y は実数型です。データは、内部処理で **double** 型へ変換されます。SIGL グラフィックスライブラリとの互換性を保つために、配列 x の要素数を表す 3 番目の引数は任意指定となっています。

関数 **plotxy**() の残りのオプション引数はそれぞれ、タイトル、x 軸のラベル、y 軸のラベルを指定します。また、プロット **CPlot** クラスへのポインタをこの関数に渡すことができます。

引数 *plot* が NULL 以外の場合、引数 *plot* がポイントするクラスのインスタンスは、関数 **plotxy**() に渡されたパラメータで初期化されます。プロットは、メンバ関数 **CPlot**::**plotting**() を使用して表示することもできます。以前に初期化済みの **CPlot** 変数が渡された場合も、関数パラメータによって再初期化されます。ポインタまたは NULL ポインタが内部処理で渡されない場合、**CPlot** クラスのインスタンスが使用され、**CPlot**::**plotting**() メンバ関数を呼び出さずにプロットが表示されます。次のコードセグメント

```
class CPlot plot;
plotxy(x, y, title, xlabel, ylabel, &plot);
```

は、次のコードセグメントと同等です。

```
class CPlot plot;
plot.data2D(x, y);
plot.title(title);
plot.label(PLOT_AXIS_X, xlabel);
plot.label(PLOT_AXIS_Y, ylabel);
```

また次のコードセグメント

```
class CPlot plot;
plotxy(x, y, n);
```

は、次のコードセグメントと同等です。

```
class CPlot plot;
plot.data2D(x, y, n);
```

最も単純なフォームでは、プログラム 23.21と図 23.1のプロットに示すように、関数 **plotxy()** はスカラ型の配列を 2 つとります。

```
/* plot a sine wave */
#include<math.h>
#include<chplot.h>
int main() {
  int numpoints = 36;
  array double x[numpoints], y[numpoints];

  lindata(0, 360, x);
  y = sin(x*M_PI/180);
  plotxy(x,y);
}
```

プログラム 23.21: プロット関数 **plotxy()** クラスを使用する単純なプログラム

プログラム 23.21のプロット関数呼び出し `plotxy(x, y)` が `plotxy(x, y, "Sine Wave", "degree", "amplitude")` に変更された場合、図 23.4に示すように、プロットのタイトルとラベルが作成されます。関数 **plotxy()** を、 **CPlot** クラスのメンバ関数と共に使用することで、プログラム 23.22に示すように、複数のデータセットをプロットできます。プログラム 23.22は、凡例付きの図 23.6を生成します。

```
/* two plots in a same figure */
#include<math.h>                                   // M_PI defined
#include<chplot.h>
int main() {
  int numpoints = 36;
  array double x1[numpoints], y1[numpoints];
  array double x2[numpoints], y2[numpoints];
  string_t title="Sine and Cosine Waves",
         xlabel="degree",
         ylabel="amplitude";
  class CPlot plot;

  lindata(0, 360, x1);
  lindata(0, 360, x2);
  y1 = sin(x1*M_PI/180);
  y2 = cos(x2*M_PI/180);
  plotxy(x1,y1,title,xlabel,ylabel,&plot);
  plot.legend("sin(x)", 0);                        // add legend for 1st plot
  plot.data2D(x2, y2);                             // Add data for 2nd plot
  plot.legend("cos(x)", 1);                        // add legend for 2nd plot
  plot.plotting();                                 // do plotting
}
```

プログラム 23.22: タイトル、ラベル、凡例を付けて、同じプロットに 2 つの関数をプロットするプログラム

プログラム 23.14では、同じプロットインスタンスを使用して、各種のファイル形式でプロットをエクスポートしています。

配列からのデータを使用する代わりに、プロット関数 **plotxyf()** はファイルからのデータを使用します。プロット関数 **plotxyf()** のプロトタイプは、次のとおりです。

```
int plotxyf(string_t filename, ...
 /* [string_t title, xlabel, ylabel], [class CPlot *plot] */);
```

`filename` のデータ形式は、メンバ関数 **CPlot::dataFile()** の引数で使用されるファイルのデータ形式と同じです。

関数 **fplotxy()** は、プログラム 23.23と図 23.18の出力に示すように、関数によって生成されたデータを使用します。

```
#include <math.h>
#include <chplot.h>
double func(double x) {
  double y;
  y =  sin(x)/(x);
  return y;
}
int main() {
  double x0 = -20, xf = 20;
  fplotxy(func, x0, xf);
}
```

プログラム 23.23: プロット関数 **fplotxy()** を使用するプログラム

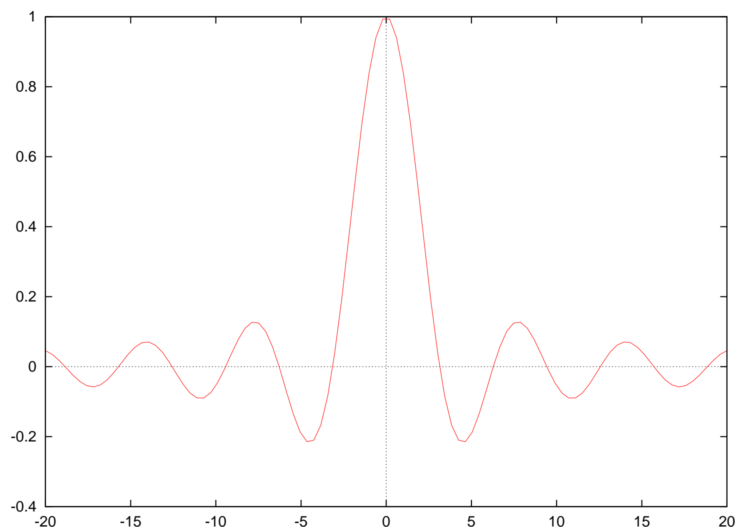


図 23.18: 関数 **fplotxy()**. によって生成されるプロット

プロット関数 **fplotxy()** のプロトタイプは、次のとおりです。

```
int fplotxy( double (*func)(double x), double x0, double xf, ...
/* [num, [string_t title, xlabel, ylabel], [class CPlot *plot]] */);
```

関数 **fplotxy**() は、範囲 $x0 \leq x \leq xf$ の *x* に対する関数の値をプロットします。プロット対象の関数 *func* は、関数へのポインタとして指定され、引数として *double* 型を受け取り、*double* 型の値を返します。引数 *x0* および *xf* は、プロット対象範囲のエンドポイントです。オプション引数 *num* は、プロット対象範囲のデータ点の数を指定します。プロットされるデータ点は、範囲内に均等に配置されます。既定では、100 個のデータ点がプロットされます。また、関数の **plotxy**() と **plotxyf**() のように、プロットにオプション引数の *title*、*xlabel*、*ylabel*、および *plot* を指定することもできます。

23.3　3次元プロット

このセクションでは、3 次元プロットだけに該当する機能について説明します。

23.3.1　プロットの種類

2 次元プロットと同様に、3 次元のプロットの種類は、メンバ関数 **CPlot**::**plotType**() によって指定できます。プロットの種類に有効なマクロを表 23.7に、マクロに対応するプロットを図 23.19に示します。既定では、3 次元プロットでは、**PLOT_PLOTTYPE_LINES** というプロットの種類が使用されます。

表 23.7: 各種 3 次元プロットのマクロ

PLOT_PLOTTYPE_LINES	データ点は、線で接続されます。
PLOT_PLOTTYPE_IMPULSES	xy 平面からデータ点まで、縦線を表示します。
PLOT_PLOTTYPE_POINTS	各データ点にマーカーを表示します。
PLOT_PLOTTYPE_LINESPOINTS	各データ点にマーカーを表示し、線で結びます。
PLOT_PLOTTYPE_SURFACES	データ点は、滑らかなサーフェスで接続され、網掛け表示されます。
PLOT_PLOTTYPE_VECTORS	ベクトルを表示します。

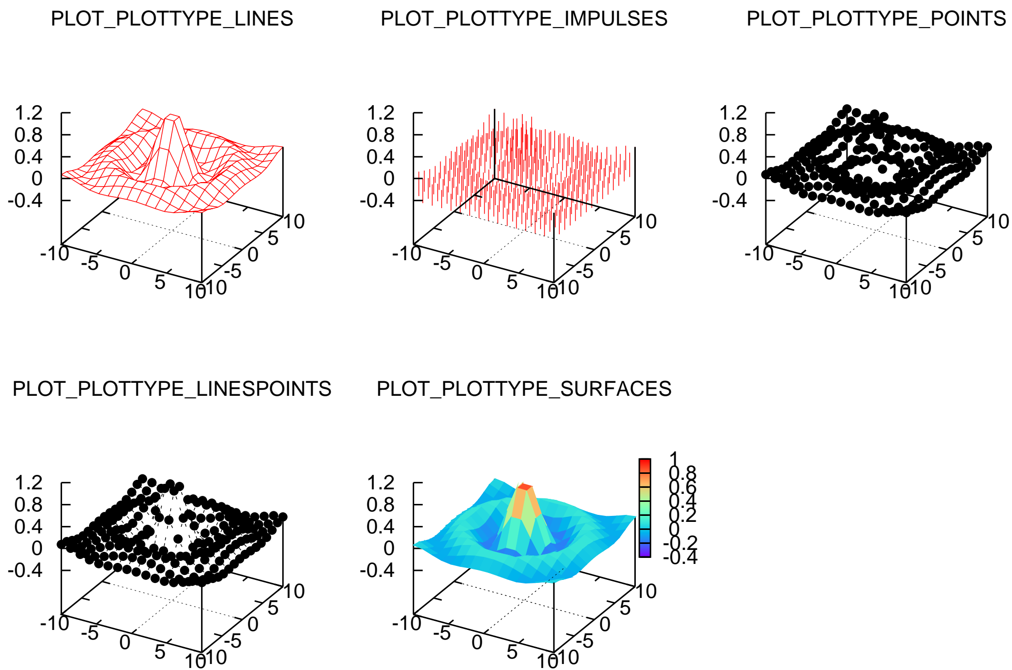


図 23.19: 各種の3次元プロット

23.3.2 各種の座標系を使用したプロット

2次元プロットでは、直交座標系または極座標系でデータセットをプロットできます。3次元プロットの場合、直交座標系、球座標系、円柱座標系のいずれかでデータセットをプロットできます。座標系は、次のメンバ関数を使用して指定できます。

```
void CPlot::coordSystem(int coord_system, .../* int angle_unit */);
```

座標系の引数 *coord_system* は、それぞれ、直交座標系、球座標系、円柱座標系を示す3つのマクロ **PLOT_COORD_CARTESIAN**、**PLOT_COORD_SPHERICAL**、**PLOT_COORD_CYLINDRICAL** のうち、いずれかのマクロを指定できます。既定では、3次元プロットには直交座標系を使用します。各座標系のデータ点は、3つの値から成ります。それぞれ、直交座標系は (x,y,z)、球座標系は (r,θ,ϕ)、円柱座標系は (r,θ,z) です。

次の公式を使用して、球座標系のデータ点を直交座標空間にマッピングします。

$$
\begin{aligned}
x &= r\cos(\theta)\cos(\phi) \\
y &= r\sin(\theta)\cos(\phi) \\
z &= r\sin(\phi)
\end{aligned}
$$

プログラム 23.24は、図 23.20に示すように、球座標系のプロットを生成します。

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

#define NUMT 37
#define NUMP 19
int main() {
    array double theta[NUMT], phi[NUMP], r[NUMT*NUMP];
    class CPlot plot;

    lindata(0, 2*M_PI, theta);
    lindata(-M_PI/2, M_PI/2, phi);
    r = (array double [NUMT*NUMP])1;
    plot.data3D(theta, phi, r);
    plot.coordSystem(PLOT_COORD_SPHERICAL);
    plot.axisRange(PLOT_AXIS_XY, -2.5, 2.5);
    plot.plotting();
}
```

プログラム 23.24: 球座標系を使用するプロットプログラム

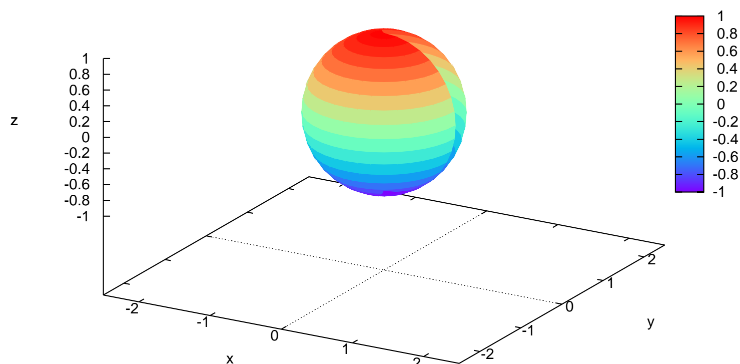


図 23.20: 球座標系によるプロット

円柱座標系の場合、データ点は、次の式で直交座標空間にマップされます。

$$\begin{aligned} x &= r\cos(\theta) \\ y &= r\sin(\theta) \\ z &= z \end{aligned}$$

プログラム 23.25は、図 23.21に示すように、球座標系でプロットを生成します。

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

#define NUMT 36
#define NUMZ 20
int main() {
    int numpoints = 36;
    array double theta[NUMT], z[NUMZ], r[NUMT*NUMZ];
    int i, j;
    class CPlot plot;

    lindata(0, 360, theta);
    lindata(0, 2*M_PI, z);
    for(i=0; i<NUMT; i++) {
        for(j=0; j<NUMZ; j++) {
            r[i*20+j] = 2+cos(z[j]);
        }
    }
    plot.data3D(theta, z, r);
    plot.coordSystem(PLOT_COORD_CYLINDRICAL, PLOT_ANGLE_DEG);
    plot.axisRange(PLOT_AXIS_XY, -4, 4);
    plot.plotting();
}
```

プログラム 23.25: 円柱座標系を使用するプロットプログラム

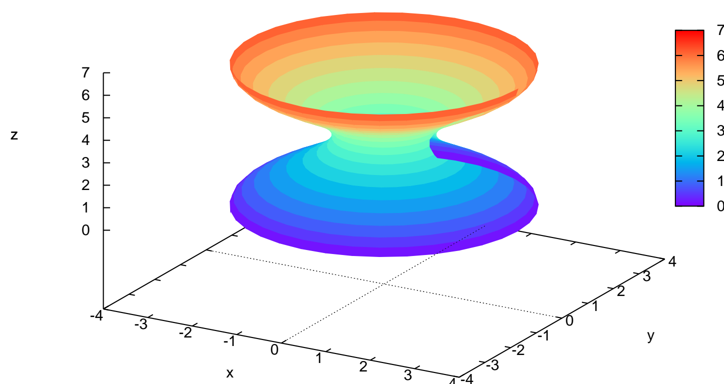


図 23.21: 円柱座標系によるプロット

メンバ関数 **CPlot::coordSystem()** のオプション引数 *angle_unit* は、球座標系および円柱座標系の角度位置の測定単位を指定します。オプション引数 *angle_unit* の有効なマクロは、角度の測定単位が度の場合は **PLOT_ANGLE_DEG** で、ラジアンの場合は **PLOT_ANGLE_RAD** です。球座標系または円柱座標系の場合、既定では、角度位置の単位はラジアンです。

23.3.3　3 次元プロット関数

2 次元プロットの関数の **fplotxy()**、**plotxy()**、および **plotxyf()** のように、高度な 3 次元プロット関数の **fplotxyz()**、**plotxyz()**、および **plotxyzf()** は、3 次元プロットを作成しやすいように設計されています。**CPlot** メンバ関数と共にこれらの関数を使用して、より高度なプロットを作成できます。

プロット関数 **plotxyz()** のプロトタイプは、次のとおりです。

```
int plotxyz(double x[&], array double y[&], array double &z, ...
/* [string_t title, xlabel, ylabel, zlabel], [class CPlot *plot] */);
```

配列 x、y および z は実数型です。**double** 型へのデータ変換は、内部処理されます。オプション引数で、タイトルとラベルを指定できます。また、関数 **plotxyz()** に渡された値を取得するために、プロット **CPlot** クラスへのポインタをこの関数に渡すことができます。次のコードセグメント

```
class CPlot plot;
plotxyz(x, y, z, title, xlabel, ylabel, zlabel, &plot);
```

は、次のコードセグメントと同等です。

```
class CPlot plot;
plot.data3D(x, y, z);
plot.title(title);
plot.label(PLOT_AXIS_X, xlabel);
plot.label(PLOT_AXIS_Y, ylabel);
plot.label(PLOT_AXIS_Z, zlabel);
```

最も単純なフォームでは、プログラム 23.26と図 23.3のプロットに示すように、関数 **plotxyz()** はスカラ型の配列を 3 つ使用します。

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

#define NUMX 20
#define NUMY 30
int main() {
    double x[NUMX], y[NUMY], z[NUMX*NUMY];
    double r;
    int i, j;

    lindata(-10, 10, x);
    lindata(-10, 10, y);
    for(i=0; i<NUMX; i++) {
        for(j=0; j<NUMY; j++) {
            r = sqrt(x[i]*x[i]+y[j]*y[j]);
            z[30*i+j] = sin(r)/r;
        }
    }
    plotxyz(x, y, z);
}
```

プログラム 23.26: プロット関数 **plotxyz()** を使用するプログラム

配列からのデータを使用する代わりに、プロット関数 **plotxyzf()** はファイルからのデータを使用します。プロット関数 **plotxyzf()** のプロトタイプは、次のとおりです。

```
int plotxyzf(string_t filename, ...
 /* [string_t title, xlabel, ylabel, zlabel], [class CPlot *plot] */);
```

`filename` のデータ形式は、メンバ関数 **CPlot::dataFile()** の引数で使用されるファイルのデータ形式と同じです。

関数 **fplotxyz()** は、プログラム 23.27と、図 23.22の出力に示すように、関数によって生成されたデータを使用します。

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

int main() {
    string_t title="fplotxyz()",
             xlabel="X-axis",
             ylabel="Y-axis",
             zlabel="Z-axis";
    double x0 = -3, xf = 3, y0 = -4, yf = 4;
    int x_num = 20, y_num = 50;

    double func(double x, double y) {   // function to be plotted

        return 3*(1-x)*(1-x)*exp(-(x*x) - (y+1)*(y+1) )
               - 10*(x/5 - x*x*x - pow(y,5))*exp(-x*x-y*y)
               - 1/3*exp(-(x+1)*(x+1) - y*y);
    }
    fplotxyz(func, x0, xf, y0, yf, x_num, y_num, title, xlabel, ylabel, zlabel);
}
```

プログラム 23.27: プロット関数 **fplotxyz()** を使用するプログラム

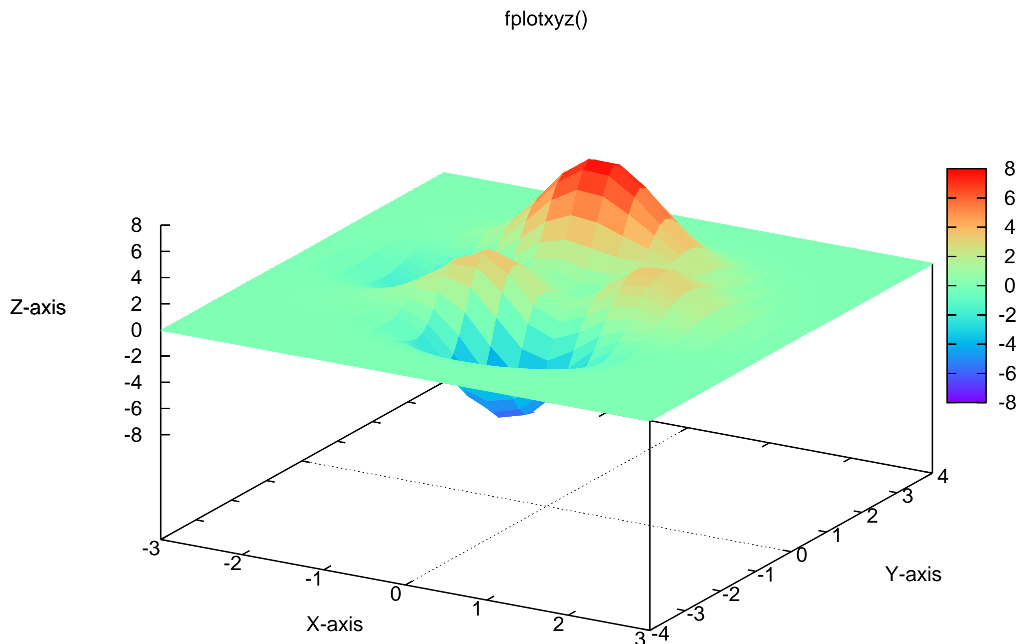


図 23.22: 関数 **fplotxyz()** によって生成されるプロット

プロット関数 **fplotxyz()** のプロトタイプは、次のとおりです。

```
int fplotxyz( double (*func)(double x, double y), double x0,
 double xf, double y0, double yf, ..., /* [x_num,  y_num,
 [string_t title, xlabel, ylabel, zlabel], [class CPlot *plot]] */);
```

関数 **fplotxy()** は、変数 *x* と *y* の関数を範囲 $x0 \le x \le xf$ と $y0 \le y \le yf$. にプロットします。プロット対象の関数 *func* は、関数へのポインタとして指定され、2 つの引数 x と y を受け取り、double 型の値を返します。引数 *x*0 および *xf* は *x* のエンドポイントであり、引数 *y*0 および *yf* は *y* のエンドポイントです。オプション引数 *x_num* と *y_num* は、*x* 座標と *y* 座標で表すデータ点をいくつプロットするかをそれぞれ指定します。プロットされるデータ点の数は、範囲内に均等に配置されます既定では、*x* 座標と *y* 座標それぞれについて、25 個のデータ点がプロットされます。また、関数 **plotxyz()** と **plotxyzf()** のように、プロットにオプションの引数 *title*、*xlabel*、*ylabel*、*zlabel*、および *plot* を指定することもできます。

23.4　動的な Web プロット

CGI プログラムを通じたプロットは、多くの Web ベースのアプリケーションで非常に有用です。Ch Professional Edition と CGI ツールキットを使用すると、動的なプロットをオンラインで簡単に生成することができます。動的なプロットを生成する方法については、このセクションで説明します。さらに、ブラウザ、Web サーバー、CGI プログラム間でデータ転送を行う際に、データを暗号化および復号化するしくみについても説明します。

図 23.23および対応する HTML をプログラム 23.28に示す Web ベースのプロットでは、プロットのパラメータは Web ブラウザから送信されますが、その際にブラウザで暗号化されています。

プログラム 23.29に示す先頭の CGI プログラム **webplot1.ch** のメンバ関数 **CRequest::getFormNameValue()** が、名前と値の組み合わせによるパラメータを復号化します。

プログラム 23.30に示すように、復号化されたパラメータは、クエリ文字列として 2 番目の CGI プログラム **webplot2.ch** に渡されます。メンバ関数 **CRequest::getFormNameValue()** を使用することで、これらのパラメータを再び取得できます。

図 23.24に、PNG ファイルとして生成され、Web ブラウザを介して表示されるプロットを示します。

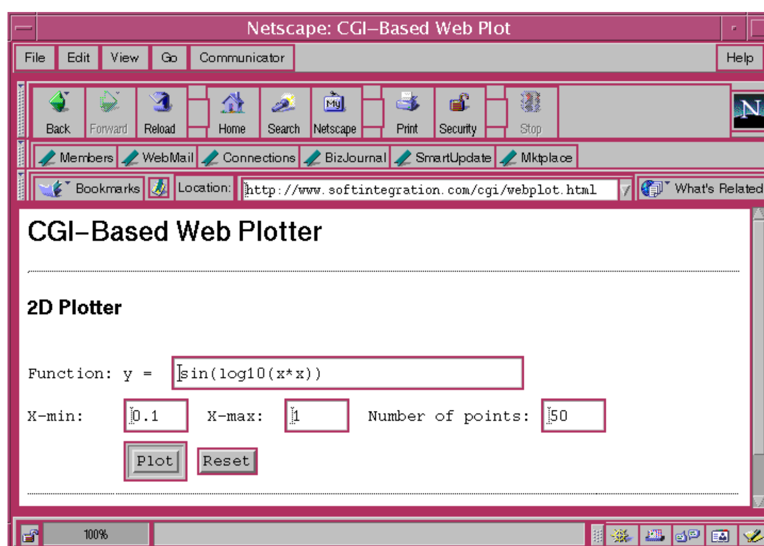


図 23.23: 入力フォームベースの Web プロッタ

```
<HTML>
<HEAD>
<TITLE>
CGI-Based Web Plot
</TITLE>
</HEAD>
<BODY bgcolor="#FFFFFF" text="#000000" vlink="#FF0000">
<H1>
CGI-Based Web Plotter
</H1>

<HR>
<H2>2D Plotter</H2>
<PRE>
<FORM method="post" action="/cgi-bin/chcgi/toolkit/demos/sample/webplot1.ch">
Function: y =  <INPUT name="expression" value="sin(log10(x*x))" size=35>
X-min:    <INPUT name="xMin" value="0.1" size=5>  X-max:  <INPUT name="xMax"
value="1" size=5>  Number of points: <INPUT name="numpoints" value="50" size=5>
          <INPUT type="submit" value="Plot"> <INPUT type="reset" value="Reset">
<HR>
</BODY>
</HTML>
```

プログラム 23.28: プロットパラメータを送信する HTML ファイル

```
#!/bin/ch
#include <cgi.h>

int main() {
   int i, num;
   chstrarray name, value;
   class CResponse Response;
   class CRequest Request;
   class CServer Server;

   num = Request.getFormNameValue(name, value);
   Response.setContentType("text/html");
   Response.begin();
   Response.title("Web Plot");
   printf("<center>\n");
   printf("<img src=\"/cgi-bin/chcgi/toolkit/demos/sample/webplot2.ch");
   for (i=0; i<num; i++){
      putc(i == 0 ? '?' : '&', stdout);
      fputs(Server.URLEncode(name[i]),stdout);
      putc('=', stdout);
      fputs(Server.URLEncode(value[i]),stdout);
   }
   printf("\">\n");
   printf("</center>\n");
   Response.end();
}
```

プログラム 23.29: CGI プログラム webplot1.ch

```
#!/bin/ch
#include <cgi.h>
#include <chplot.h>

int main() {
   double MinX, MaxX, Step, x, y;
   int pointsX, pointsY, i;
   chstrarray name, value;
   class CResponse Response;
   class CRequest Request;
   class CPlot plot;

   Request.getFormNameValue(name, value);
   MinX = atof(value[1]);
   MaxX = atof(value[2]);
   pointsX = atoi(value[3]);
   double x1[pointsX], y1[pointsX];

   Step = (MaxX - MinX)/(pointsX-1);
   for(i=0;i<pointsX;i++) {
     x = MinX + (i*Step);
     y = streval(value[0]);
     x1[i] = x;
     y1[i] = y;
   }

   Response.setContentType("image/png");
   Response.begin();
   plotxy(x1, y1, value[0], "X", "Y", &plot);
   /* output plot in color png file format */
   plot.outputType(PLOT_OUTPUTTYPE_STREAM, "png");
   plot.plotting();
   Response.end();
}
```

プログラム 23.30: CGI プログラム webplot2.ch

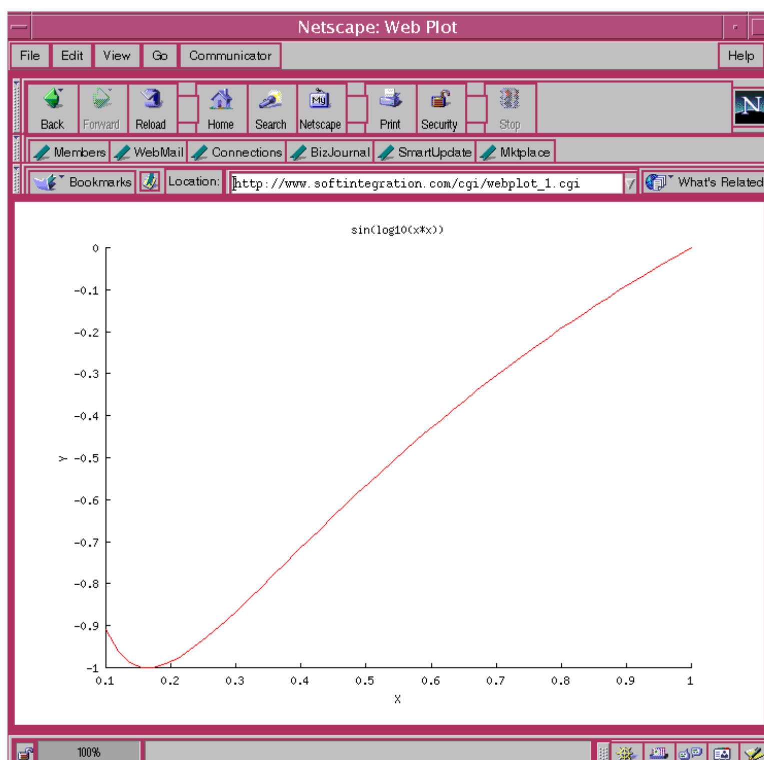


図 23.24: Web プロットを通じて生成されるプロット

第24章　数値解析

Chの数値解析は、Cの精神を損なうことなく、可能な限り簡潔に拡張されています。多くの場合、数値解析の複雑な問題がChではたった1回の関数呼び出しで解決できます。Chの高度な数値解析の特徴は、工学および科学分野での用途で非常に有用です。

Chの数値解析関数のカテゴリを表24.1に示します。ヘッダーファイル**numeric.h**でプロトタイプ化されている数値解析関数については、表24.2に示します。ヘッダーファイル**numeric.h**には、**math.h**、**stdarg.h**、**array.h**、および**dlfcn.h**のヘッダーファイルが含まれています。**maxloc()**、**mean()**、**median()**、**minloc()**、**product()**、**sort()**、**std()**、**sum()**　などの関数では参照関数の引数が使用されます。これらの関数は、次元やデータ型が異なる配列を扱うことができます。オプションの2番目の引数は、**mean()**、**median()**、**product()**、**sort()**、**std()**、　および**sum()**の関数ではdouble型の配列を指定する必要があります。この2番目の引数は、関数**cproduct()**と**csum()**ではdouble complex型の配列を指定する必要があります。これらの関数では、ベクトルのオプションの2番目の引数に、2次元配列の先頭の引数の各行を計算した結果が含まれます。同様に、関数minv()とmaxv()は、それぞれ、入力された2次元配列引数の各行の最大値と最小値を返します。

本章では、科学技術計算に使用できる数値解析関数について説明します。各関数の詳細については、『*Ch* **言語環境リファレンスガイド**』の数値解析に関する章を参照してください。

表 24.1: Ch の数値解析関数のカテゴリ

データ解析と統計
データ補間とカーブフィッティング
関数の最小化または最大化
多項式
非線形方程式
常微分方程式
微分
積分
行列解析関数
行列分解
特殊行列
線形方程式
固有値と固有ベクトル
高速 Fourier 変換
畳み込みとフィルタリング
相互相関
特殊数学関数

24.1 数学関数

初等数学関数の多くは、標準の C ヘッダーファイル **math.h** で定義されています。このセクションでは、標準の C ライブラリでは定義されていない使用頻度の高い数学関数を説明します。

24.1.1 外積

関数 **cross**() のプロトタイプは次のとおりです。

```
array double cross(array double a[&], array double b[&])[3];
```

3 つの要素を含む 2 つのベクトルの外積を計算します。実数型のベクトル a と b の外積が返されます。
たとえば、2 つのベクトルの外積を評価するには、次のコマンドを使用できます。

```
> array double a[3] = {1, 2, 3}
> array float b[3] = {2, 3, 4}
> cross(a, a)
0.0000 0.0000 0.0000
> cross(a, b)
-1.0000 2.0000 -1.0000
```

表 24.2: 数値解析関数

関数	説明
balance()	一般的な実数型の行列を調整し、固有値計算の精度を高めます。
ccompanionmatrix()	複素数型の随伴行列を計算します。
cdeterminant()	複素行列の行列式を求めます。
cdiagonal()	複素行列の対角ベクトルを求めます。
cdiagonalmatrix()	複素ベクトルから対角行列を作成します。
cfevalarray()	複素配列関数を評価します。
cfunm()	一般的な複素数型の行列関数を評価します。
charpolycoef()	行列の固有多項式の係数を計算します。
choldecomp()	対称正定値行列 A の Cholesky 分解を計算します。
cinverse()	複素数型の正方行列の逆行列を求めます。
clinsolve()	LU 分解で複素連立一次方程式の解を求めます。
cmean()	複素配列のすべての要素の平均値と各行の平均値を計算します。
combination()	n 個から一度に k 個を選ぶ組み合わせの数を計算します。
companionmatrix()	随伴行列を計算します。
complexsolve()	複素方程式の解を求めます。
condnum()	行列の条件数を計算します。
conv()	1 次元離散 Fourier 変換 (DFT) で畳み込みを計算します。
conv2()	2 次元離散 Fourier 変換 (DFT) で畳み込みを計算します。
corrcoef()	相関係数を計算します。
correlation2()	2 次元相関係数を求めます。
cpolyeval()	複素多項式の値を複素数のデータ点で計算します。
cproduct()	すべての要素の積または任意の次元を持つ複素配列の要素の積、および 2 次元複素配列の各行の要素の積を計算します。
covariance()	共分散行列を求めます。
cross()	ベクトルの外積を計算します。
csum()	すべての要素の和または任意の次元の複素配列の要素の和、および 2 次元複素配列の各行の要素の和を計算します。
ctrace()	複素行列の対角要素の合計を計算します。
ctriangularmatrix()	複素行列の上三角部または下三角部を求めます。
cumprod()	配列内の要素の累積積を計算します。
cumsum()	配列内の要素の累積和を計算します。
curvefit()	データ点 x と y のセットを、指定したベース関数の線形結合へ一致させます。
deconv()	1 次元離散型 Fourier 変換 (DFT) で逆重畳を計算します。
derivative()	所定のデータ点での関数の導関数を数値計算します。
derivatives()	複数のデータ点での関数の導関数を数値計算します。
determinant()	行列の行列式を求めます。
diagonal()	行列の対角ベクトルを求めます。
diagonalmatrix()	ベクトルの対角行列を作成します。
difference()	配列の隣接要素の差を計算します。

表 24.2: 数値解析関数 (続き)

関数	説明
dot()	ベクトルの内積を計算します。
eigen()	固有値と固有ベクトルを求めます。
expm()	行列の指数を計算します。
factorial()	階乗を計算します。
fevalarray()	配列関数を評価します。
fft()	N 次元の高速 Fourier 変換 (FFT) を計算します。
filter()	データをフィルタ処理します。
filter2()	2 次元離散型 Fourier 変換で FIR フィルタ処理を行います。
findvalue()	配列の要素のゼロ以外のインデックスを求めます。
fliplr()	行列を左右に反転します。
flipud()	行列を上下に反転します。
fminimum()	1 次元関数の最小値を求め、最小値に対応する位置を特定します。
fminimums()	n 次元関数の最小の位置と値を求めます。
fsolve()	非線形連立方程式のゼロ位置を求めます。
funm()	一般的な実数型の行列関数を評価します。
fzero()	1 つの変数を含む非線形関数のゼロ点を求めます。
gcd()	整数型の 2 つの配列の対応する値の最大公約数を得ます。
getnum()	コンソールから既定値の数を取得します。
hessdecomp()	直交行列またはユニタリ行列の相似変換で、一般的な実数型の行列を 上位の Hessenberg 形式に還元します。
histogram()	ヒストグラムを計算し、プロット作成します。
householdermatrix()	Householder 行列を求めます。
identitymatrix()	単位行列を作成します。
ifft()	N 次元の逆高速 Fourier 変換 (FFT) を計算します。
integral1()	1 次元関数の数値積分を行います。
integral2()	2 次元関数の数値積分を行います。積分範囲は 2 つの変数共に固定です。
integral3()	3 次元関数の数値積分を行います。積分範囲は 3 つの変数全てについて固定です。
integration2()	2 次元関数の数値積分を行います。一方の変数に対する積分範囲は 他方の変数に依存します。
integration3()	3 次元関数の数値積分を行います。2 つの変数に対する積分範囲は残り 1 つの 変数に依存します。
interp1()	1 次元補間を行います。
interp2()	2 次元補間を行います。
inverse()	正方行列の逆行列を求めます。
lcm()	整数型の 2 つの配列の対応する値の最小公倍数を求めます。
lindata()	等間隔の連続データを生成します。
linsolve()	LU 分解で連立一次方程式の解を求めます。
linspace()	等間隔の連続配列を生成します。
llsqcovsolve()	線形最小 2 乗法で、既知の共分散を含む連立一次方程式の解を求めます。
llsqnonnegsolve()	線形最小 2 乗法で、ゼロ以外の値を含む連立一次方程式の解を求めます。

表 24.2: 数値解析関数 (続き)

関数	説明
llsqsolve()	最小 2 乗法で一次方程式の解を求めます。
logm()	行列の自然対数を計算します。
logdata()	対数軸上で等間隔に区切られたデータを生成します。
logspace()	対数軸上で等間隔に区切られた配列を生成します。
ludecomp()	一般的な n x m 行列を LU 分解します。
maxloc()	配列の最大値の位置を求めます。
maxv()	行列の各行の要素の最大値を求めます。
mean()	配列のすべての要素の平均値と各行の平均値を計算します。
median()	配列のすべての要素の中央値と各要素の中央値を計算します。
minloc()	配列の最小値の位置を計算します。
minv()	2 次元配列の各行の最小値を計算します。
norm()	ベクトルと行列のノルムを計算します。
nullspace()	行列の null 空間を計算します。
oderk()	ルンゲクッタ法を使用して、常微分方程式の解を求めます。
orthonormalbase()	行列の直交ベースを計算します。
pinverse()	行列の Moore-Penrose 擬似逆行列を計算します。
polyder()	多項式の導関数を求めます。
polyder2()	2 つの多項式の積または商の導関数を計算します。
polyeval()	多項式の値とその導関数を計算します。
polyevalarray()	点列で多項式を評価します。
polyevalm()	行列多項式の値を計算します。
polyfit()	データ点のセットを多項式関数に一致させます。
product()	任意の次元を持つ配列のすべての要素の積、または 2 次元配列の各列の要素の積を計算
qrdecomp()	行列の直交三角形の QR 分解を計算します。
rank()	行列のランクを計算します。
residue()	部分端数の展開または剰余を計算します。
rcondnum()	行列の逆比例条件を見積ります。
roots()	多項式の根を計算します。
rot90()	行列を 90 度回転させます。
specialmatrix()	特別な行列を生成します。
schurdecomp()	Schur 分解を計算します。
sign()	引数の符号を求めます
sort()	要素を昇順に並べ換えて、ランク付けします。
sqrtm()	マトリクスの平方根を計算します。
std()	データセットの標準偏差を計算します。
sum()	配列のすべての要素の合計または各行の要素の合計を計算します。
svd()	特異値を分解します。
trace()	対角要素の合計を計算します。

表 24.2: 数値解析関数 (続き)

関数	説明
triangularmatrix()	行列の上三角または下三角を求めます。
unwrap()	π より大きい絶対ジャンプを $2*\pi$ 補数へ変えることにより、 入力される配列 x の各要素のラジアン位相をアンラップします。
urand()	一様乱数を生成します。
xcorr()	1 次元相互相関ベクトルを求めます。

24.1.2 内積

関数 **dot**() のプロトタイプは次のとおりです。

```
double dot(array double a[&], array double b[&]);
```

2 つのベクトルの内積を計算します。実数型のベクトル a と b の内積が返されます。この 2 つのベクトルの要素の数は同じでなければなりません。ベクトルの要素の数が異なると、関数 **dot**() は NaN を返します。

たとえば、2 つのベクトルの内積を評価するには、次のコマンドを使用できます。

```
> array double a[3] = {1, 2, 3}
> array double b[] = {1, 2, 3, 4, 5}
> dot(a, a)
14.0000
> dot(b, b)
55.0000
```

24.1.3 一様乱数

ヘッダーファイル **stdlib.h** で定義されている、C の標準の関数 **srand**() と **rand**() を使用して、乱数を求めることができます。関数 **urand**() のプロトタイプは次のとおりです。

```
double urand(array double &x);
```

周期 2^{32} の乱数ジェネレータを使用して、0〜1 の範囲の連続する擬似乱数が得られます。配列引数 x のパラメータが NULL でない場合、乱数は引数 x に格納され、配列の先頭の要素の値が返されます。関数 **urand**() の引数が NULL の場合は、一様乱数のみが返されます。次に例を示します。

```
> urand(NULL)
0.494766
> array double x[2], y[2][3]
> urand(x)
> x
0.513871 0.175726
```

```
> urand(y)
> y
0.3086 0.5345 0.9476
0.1717 0.7022 0.2264
```

24.1.4　符号関数

関数 **sign()** のプロトタイプは次のとおりです。

```
int sign(double x);
```

引数 x の符号を求めます。$x > 0$ の場合は 1、$x < 0$ の場合は-1、$x = 0$ の場合は 0 が返されます。次に例を示します。

```
> sign(-10)
-1
> sign(10)
1
```

24.1.5　最大公約数

関数 **gcd()** のプロトタイプは次のとおりです。

```
int gcd(array int &u,  array int &v, array double &g, ...
    /* [array int c[&], array int d[&]] */);
```

整数型の配列 u と v の対応する要素の最大公約数が得られます。配列 u と v は、非負の整数型データを含む同じサイズの配列でなければなりません。出力配列 g は、u と同じサイズの正の整数型配列になります。オプションの出力配列 c と d は、u と同じサイズで、方程式 $u.*c+v.*d=g.$ を満たします。関数 **gcd()** は、成功すると 0 を返し、失敗すると-1 を返します。次に例を示します。

```
> array int u[2][3] = {1,  2, 7,\
                       15, 3, 4}
> array int v[2][3] = {2,  4, 8,\
                       3,  8, 3}
> array int g[2][3],c[2][3],d[2][3]
> gcd(u,v,g,c,d)
> g
1 2 1
3 1 1
> u.*c+v.*d
1 2 1
3 1 1
```

24.1.6　最小公倍数

関数 **lcm()** のプロトタイプは次のとおりです。

```
int lcm(array int &g,  array double &u, array double &v);
```

整数型の 2 つの配列の対応する要素の最小公倍数が得られます。配列 *u* と *v* は、非負の整数型データを含む同じサイズの配列でなければなりません。出力配列 *g* は、*u* と同じサイズの正の整数型配列になります。関数 **lcm()** は、成功すると 0 を返し、失敗すると-1 を返します。次に例を示します。

```
> array int u[2][3] = {1,2,7,15,3,4}
> array int v[2][3] = {2,4,8,3,8,3}
> array int g[2][3]
> lcm(g,u,v)
> g
2 4 56
15 24 12
```

24.1.7　複素方程式

複素方程式は、次のような一般的な極形式で表すことができます。

$$R_1 e^{i\phi_1} + R_2 e^{i\phi_2} = z_3 \tag{24.1}$$

ここで、z_3 は、デカルト座標 $x_3 + iy_3$ では `complex(x3, y3)` と表すことができ、極座標 $R_3 e^{i\phi_3}$ では `polar(R3, phi3)` と表すことができます。この複素方程式は、実数部と虚数部に分けることができるため、R_1、ϕ_1、R_2、および ϕ_2 の 4 つのパラメータのうち 2 つの未知数はこの方程式で解くことができます。R_1、ϕ_1、R_2、および ϕ_2 のパラメータは、それぞれ 1、2、3、および 4 の位置に置かれます。関数 **complexsolve()** のプロトタイプは次のとおりです。

```
int complexsolve(int n1, int n2,
     double phi_or_r1, double phi_or_r2, double complex z3,
     double &x1, double &x2, double &x3, double &x4);
```

これで、2 つの未知数を求める方程式 (24.1) を解くことができます。引数 *n1* と *n2* は、それぞれ方程式 (24.1) の左側の 1 番目と 2 番目に位置する未知数です。位置を示す有効値は、1、2、3、または 4 です。引数 *phi_or_r1* と *phi_or_r2* は、方程式の左側にある残り 2 つの既知の変数の値です。引数 *z3* は式の右側の複素数です。引数 *x1* と *x2* には、それぞれ 1 番目と 2 番目の未知数の結果が含まれます。解法が複数ある場合は、引数 *x3* と *x4* には、2 つ目の解法による 1 番目と 2 番目の未知数の結果がそれぞれ含まれます。関数 **complexsolve()** は、0、1、または 2 のいずれかの値を含む解法の数を返します。

たとえば、方程式 $3.5e^{i4.5} + R_2 e^{i\phi_2} = 1 + i2$ の未知数は、次のコマンドで求めることができます。

```
> int n1 = 3, n2 = 4
> double phi_or_r1= 3.5, phi_or_r2 = 4.5
> double R2, phi2, x3, x4
```

```
> complex z3 = complex(1, 2)
> complexsolve(n1, n2, phi_or_r1, phi_or_r2, z3, R2, phi2, x3, x4)
1
> R2
5.6931
> phi2
1.2606
```

たとえば、方程式 $3.5e^{i\phi_1} + 4.5e^{i\phi_2} = e^{i} + 3e^{i4}$ の未知数 ϕ_1 と ϕ_2 は、次のコマンドで求めることができます。

```
> int n1 = 2, n2 = 4
> double phi_or_r1= 3.5, phi_or_r2 = 4.5
> double phi1, phi2, phi1_2, phi2_2
> complex z3 = polar(1, 1)-polar(3, 4)
> complexsolve(n1, n2, phi_or_r1, phi_or_r2, z3, \
               phi1, phi2, phi1_2, phi2_2)
2
> phi1
-0.3890
> phi2
1.7354
> phi1_2
2.1765
> phi2_2
0.0521
```

24.2 データ解析と統計

24.2.1 コンソールからの数値の取得

関数 **getnum()** のプロトタイプは次のとおりです。

```
double getnum(string_t msg, double d);
```

標準の入力ストリームでコンソールから実数を取得するか、2 番目の引数の既定値を取得します。この関数は、復帰文字が入力されるか、または新しい数値として *stdin* から倍精度の浮動小数点数が入力されると、既定の数値を返します。ただし、入力した数値が無効な場合は、別の数値の入力が求められます。文字列 *msg* に記述されたメッセージが出力されます。次に例を示します。

```
> double d
> d = getnum("Please enter a number[10.0]: ", 10);
Please enter a number[10.0]:
90
```

```
> d
90.0000
```

24.2.2 配列へのデータの代入

配列の初期化と代入に加え、関数 **lindata()** と **logdata()** を使用して値を配列に代入することができます。関数 **lindata()** のプロトタイプは次のとおりです。

```
int lindata(double first, double last, ... /*[array] type a[:]...[:]*/);
```

3 番目の引数に渡される配列 a の各要素に、*first* から *last* までの等間隔の連続する値を代入します。点の数は内部的に配列 a から取得されます。関数 **lindata()** は配列 a の要素数を返し、失敗すると 0 を返します。関数 **logdata()** は、3 番目の引数の配列の要素に代入される値が対数軸上で等間隔に区切られることを除き、**lindata()** と同じです。次に例を示します。

```
> array double a[6];
> lindata(0, 2, a);
> a
0.0000 0.4000 0.8000 1.2000 1.6000 2.0000
```

関数 **linspace()** のプロトタイプは次のとおりです。

```
int linspace(array double &a, double first, double last);
```

配列 a の各要素に、*first* から *last* までの等間隔の連続する値を代入します。点の数は内部的に配列 a から取得されます。関数 **linspace()** は配列 a 要素の数を返し、失敗すると 0 を返します。関数 logspace() は、先頭の引数の配列の要素に代入される値が 10^{first} から 10^{last} までを対数軸上等間隔に区切った値であることを除き、**linspace()** と同じです。次に例を示します。

```
> array double a[6], b[6];
> linspace(a, 0, 2);
> a
0.0000 0.4000 0.8000 1.2000 1.6000 2.0000
> logspace(b, 0, 2);
> b
1.0000 2.5119 6.3096 15.8489 39.8107 100.0000
```

log2.5119=0.4000、log6.3096=0.8000、log15.8489=1.2000、log39.8107=1.6000 および log100.0000=2.0000 ですから、これらの値は対数軸上では等間隔に並ぶことになります。関数 **linspace()** と **logspace()** の使用は推奨されなくなっており、段階的に廃止される予定です。関数 **lindata()** と **logdata()** を使用することを推奨します。次の関数呼び出し

```
linspace(a, 0, 2);
logpace(a, 0, 2);
```

は、以下のように置き換えることができます。

```
lindata(0, 2, a);
logdata(0, 2, a);
```

24.2.3 最小値と最大値

汎用関数 **min()** を使用して、スカラ型および配列型の式リストに含まれる最小の数値を求めることができます。引数がすべて整数型の場合、**min()** は int 型の値を返します。それ以外の場合は、double 型の値を返します。関数 **minloc()** のプロトタイプは

```
int minloc(array double &a);
```

であり、配列 *a* で最小の値を含む要素のインデックスを返します。関数 **minv()** のプロトタイプは

```
array double minv(array double a[&][&])[:];
```

であり、2 次元配列の各行ごとの最小の値を含む配列を返します。2 次元配列の各列ごとの最小の値を含む配列を計算するには、関数 **transpose()** を使用して、上記の配列の転置行列を求めます。

汎用関数 **max()** は **min()** と同様です。最小値の代わりに、関数 **max()** は引数の式リストに含まれる最大値を返します。関数 **maxloc()** のプロトタイプ。

```
int maxloc(array double &a);
```

であり、配列 *x* で最大の値を含む要素のインデックスを返します。関数 **maxv()** のプロトタイプは

```
array double maxv(array double a[&][&])[:];
```

であり、2 次元配列の各行ごとの最大の値を含む配列を返します。次に例を示します。

```
> array double a[3] = {10, 2, 3}
> float f= 2.5;
> min(a, f, 2.4)
2.0000
> minloc(a)
1
> array int i[3][2] = {1, 2, 3, 4, 5, 6}
> minv(i)
1.0000 3.0000 5.0000
> minv(transpose(i))
1.0000 2.0000
> max(4, 5, i, 10)
10
```

24.2.4 合計

関数 **sum()** のプロトタイプは

```
double sum(array double &a, ... /* [array double v[:]] */);
```

であり、配列内のすべての要素を合計します。配列が 2 次元行列の場合、関数は各行の合計を計算し、結果を 1 次元配列の 2 番目のオプション引数に格納します。次に例を示します。

```
> array double a[3] = {10, 2, 3}
> sum(a)
15.0000
> array double b[3][2] = {1, 2, 3, 4, 5, 6}
> array double v[3]
> sum(b, v)
> v
3.0000 7.0000 11.0000
```

csum() のプロトタイプは

```
double complex csum(array double complex &a, ...
                    /* [array double complex v[:]] */);
```

であり、複素配列の要素の合計に使用します。ベクトル $x = [x_1, x_2, \cdots, x_n]$ に対する累積和 $y = [y_1, y_2, \cdots, y_n]$ は、次の式で定義されます。

$$y_i = x_1 + x_2 + \cdots + x_i \qquad i = 1, 2, \cdots, n$$

関数 **cumsum**() のプロトタイプは

```
int cumsum(array double complex &y, array double complex &x);
```

であり、入力された配列 x の累積和を計算します。入力 x がベクトルの場合、x の要素の累積和が計算されます。入力 x が 2 次元行列の場合は、各行の累積和が計算されます。入力 x が 3 次元配列の場合、1 次元の累積和が計算されます。3 次元より高次の配列の累積和の計算には使用できません。次に例を示します。

```
> array double x[6]={1,2,3,4,5,6}, y[6]
> cumsum(y, x)
> y
1.0000 3.0000 6.0000 10.0000 15.0000 21.0000
> array double complex zx[3][2]={complex(1,1),2,3,complex(2,2),5,6}
> array double complex zy[3][2]
> cumsum(zy, zx)
complex(1.0000,1.0000) complex(3.0000,1.0000)
complex(3.0000,0.0000) complex(5.0000,2.0000)
complex(5.0000,0.0000) complex(11.0000,0.0000)
```

24.2.5 積

関数 **product()** のプロトタイプは

```
double product(array double &a, ... /* [array double v[:]] */);
```

であり、配列内のすべての要素の積を計算します。配列が 2 次元行列の場合、関数は各行の積を計算し、結果を 1 次元配列の 2 番目のオプション引数に格納します。次に例を示します。

```
> array double a[3] = {10, 2, 3}
> product(a)
60.0000
> array double b[3][2] = {1, 2, 3, 4, 5, 6}
> array double v[3]
> product(b, v)
> v
2.0000 12.0000 30.0000
```

関数 **cproduct()** のプロトタイプは

```
double complex cproduct(array double complex &a, ...
                        /* [array double complex v[:]] */);
```

であり、複素配列の要素の積に使用します。ベクトル $x = [x_1, x_2, \cdots, x_n]$ に対する累積積 $y = [y_1, y_2, \cdots, y_n]$ は、次の式で定義されます。

$$y_i = x_1 * x_2 * \cdots * x_i \qquad i = 1, 2, \cdots, n$$

関数 cumprod() のプロトタイプは

```
int cumprod(array double complex &y, array double complex &x);
```

であり、入力された配列 x の累積積を計算します。入力 x がベクトルの場合、x の要素の累積積が計算されます。入力 x が 2 次元行列の場合は、各行の累積積が計算されます。入力 x が 3 次元配列の場合、1 次元の累積積が計算されます。3 次元より高次の配列の累積積の計算には使用できません。次に例を示します。

```
> array double x[6]={1,2,3,4,5,6}, y[6]
> cumprod(y, x)
> y
1.0000 2.0000 6.0000 24.0000 120.0000 720.0000
> array double complex zx[3][2]={complex(1,1),2,3,complex(2,2),5,6}
> array double complex zy[3][2]
> cumprod(zy, zx)
complex(1.0000,1.0000) complex(2.0000,2.0000)
complex(3.0000,0.0000) complex(6.0000,6.0000)
complex(5.0000,0.0000) complex(30.0000,0.0000)
```

24.2.6 平均値

関数 **mean()** のプロトタイプは

```
double mean(array double &a, ... /* [array double v[:]] */);
```

であり、次元にかかわらず、実数型の配列のすべての要素の平均値を計算できます。2 次元行列の配列では、各行の平均値を計算できます。各行の平均値は、1 次元配列の 2 番目のオプション引数 v で渡されます。関数 **cmean()** のプロトタイプは

```
double complex cmean(array double complex &a, ...
                     /* [array double complex v[:]] */);
```

であり、複素配列の平均値の計算に使用します。たとえば、次のコマンドを使用して平均値を評価できます。

```
> double a[2][3] = {1, 2, 3, 6, 5, 4}
> double m
> m = mean(a)
3.5000
> array double v[2]
> mean(a, v)
> v
2.0000 5.0000
```

24.2.7 中央値

関数 **median()** のプロトタイプは次のとおりです。

```
double median(array double &a, ... /* [array double v[:]] */);
```

次元にかかわらず、実数型の配列のすべての要素の中央値を計算できます。2 次元行列の配列では、各行の中央値を計算できます。各行の中央値は、1 次元配列の 2 番目のオプション引数 v で渡されます。たとえば、次のコマンドを使用して中央値を評価できます。

```
> double a[2][3] = {1, 2, 3, 6, 5, 4}
> double m
> m = median(a)
3.5000
```

24.2.8　標準偏差

データセット σ の標準偏差は次のように定義されます。

$$\sigma = \sqrt{\frac{1}{N-1}\sum_{i=1}^{N}(x_i - \overline{x})^2}$$

ここで、N はデータセット x の観測数で、$\overline{x}$ はデータセット x の平均値です。関数 **std**() のプロトタイプは

```
double std(array double &a, ... /* [array double v[:]] */);
```

であり、次元にかかわらず、実数型の配列のすべての要素の標準偏差を計算できます。2 次元行列の配列では、各行の標準偏差を計算できます。各行の標準偏差は、1 次元配列の 2 番目のオプション引数 v で渡されます。

たとえば、次のコマンドを使用して、標準偏差を評価できます。

```
> double a[2][3] = {1, 2, 3, 6, 5, 4}
> double m
> m = std(a)
1.8708
```

24.2.9　共分散と相関係数

n 個のデータを m 回観測して得られたデータから作られた n × m 個の要素から成る配列 **X** の場合、配列 **X** の共分散行列 **C** の要素 (i,j) は、次の式で定義されます。

$$\mathbf{C}[i][j] = E[(x_i - \mu_i)(x_j - \mu_j)] \qquad i, j = 1, 2, \cdots, N;$$

ここで、E は数学的期待値を表し、$\mu_i = Ex_i$ です。行列 **u** の (i, j) 要素を

$$u_{ij} = x_{ij} - \mu_i; \qquad i = 0, 1, \cdots, N; j = 0, 1, \cdots, M$$

ただし、

$$\mu_i = \frac{1}{M}\sum_{j=1}^{M} x_{ij}; \qquad i = 0, 1, \cdots, N$$

で定義すると、共分散行列は次の式で計算できます。

$$\mathbf{C} = \frac{1}{N-1}\mathbf{u} * \mathbf{u^T}$$

関数 **covariance**() のプロトタイプは

```
int covariance(array double &c, array double &x, ...
               /* [array double &y] */);
```

であり、配列 x の共分散を計算します。配列 x には、サポートされる算術データ型であれば、ベクトルまたは任意のサイズの $n \times m$ の行列 (x がベクトルの場合は、$1 \times m$ のサイズと見なされる) を入力できます。各列の x は観測点を表し、各行は変数です。オプション引数 y の入力値には、**double** データ型以外は指定できません。また、列数は x と同じでなければなりません。したがって、y のサイズは $n_1 \times m$(y がベクトルの場合は $1 \times m$) となります。内部処理で、新しく拡張される行列 [**X**] として y が x の行に追加されます。関数呼び出し **covariance**(c, x, y) は、**covariance**(c,[**X**]) と同じです [**X**] のサイズは $(n + n_1) \times (m)$ です。内部処理で、データは double 型へ変換されます。結果 c は、x と y の値によって、分散行列または共分散行列のいずれかになります。x がベクトルで、オプション引数 y の入力がない場合、c は分散行列となります。それ以外の条件では、共分散行列となります。c の対角は、各行に対して分散のベクトルとなります。配列 c の対角の平方根は、標準偏差のベクトルです。関数 **covariance**() では、結果を計算する前に各列の平均値が削除されます。関数 **covariance**() は、成功すると 0 を返し、失敗すると-1 を返します。

相関係数行列の要素 (i, j) は次のように定義されます。

$$\texttt{corrcoef}(i,j) = \frac{C_{ij}}{\sqrt{C_{ii} * C_{jj}}}$$

ここで、C_{ij} は共分散行列の要素です。関数 **corrcoef**() のプロトタイプは

```
int corrcoef(array double &c, array double &x, ...
             /* [array double &y] */);
```

であり、相関係数行列を計算します。配列 x には、サポートされる算術データ型であれば、ベクトルまたは任意のサイズの $n \times m$ の 2 次元の配列 (x がベクトルの場合は、$1 \times m$ のサイズと見なされる) を入力できます。各列の x は観測点を表し、各行は変数です。オプション引数 y の入力値には、double データ型以外は指定できません。また、列数は x と同じでなければなりません。したがって、y のサイズは $n_1 \times m$(y がベクトルの場合は $1 \times m$) となります。内部処理で、新しく拡張される行列 [**X**] として y が x の行に追加されます。関数呼び出し **corrcoef**(c, x, y) は **corrcoef**(c,[**X**]) と同じです ($\mathbf{X} = \begin{pmatrix} x \\ y \end{pmatrix}$ として定義される)。[**X**] のサイズは $(n + n_1) \times m$ です。内部処理で、データは double 型へ変換されます。結果 c は、相関係数の行列です。

たとえば、以下のコマンドで行列の共分散と相関係数を計算できます。

```
> #define N 3
> #define M 4
> array double  x[N][M]={1,2,3,4, \
                         0,0,0,1, \
                         2,3,4,5}
> array double c[N][N]
> covariance(c, x)
> c
```

```
1.6667 0.5000 1.6667
0.5000 0.2500 0.500000
1.6667 0.5000 1.6667
> corrcoef(c, x)
1.0000 0.7746 1.0000
0.7746 1.0000 0.7746
1.0000 0.7746 1.0000
```

関数 correlation2() のプロトタイプは

```
int correlation2(array double x[&][&], array double y[&][&])
```

であり、2 つの正方行列 *x* と *y* の相関係数を計算します。相関係数は次のように定義されます。

$$c = \frac{\sum_{i=1}^{n}\sum_{j=1}^{n}(xx_{ij} * yy_{ij})}{\sqrt{(\sum_{i=1}^{n}\sum_{j=1}^{n} xx_{ij}^2) * (\sum_{i=1}^{n}\sum_{j=1}^{n} yy_{ij}^2)}}$$

ただし、

$$xx_{ij} = x_{ij} - \mu_x$$
$$yy_{ij} = y_{ij} - \mu_y$$

であり、μ_x と μ_y は、行列 *x* と *y* の平均値です。
次に例を示します。

```
> array double x[3][3]={1,2,3, 3,4,5, 6,7,8}
> array double y[3][3]={3,2,2, 3,8,5, 6,2,5}
> correlation2(x, y)
0.3266
```

24.2.10 ノルム

ベクトルまたは行列のノルムは、ベクトルまたは行列の要素の大きさを測るスカラ値です。関数 **norm()** のプロトタイプは次のとおりです。

```
double norm(array double complex &a, char *char);
```

計算されるベクトル *a* または行列 *a* のノルムの型は、引数 *mode* の値によって異なります。表 24.3に、ベクトルと行列の両方について、さまざまなノルムのモードとアルゴリズムの定義を示します。

表 24.3: ノルムの定義

モード	ノルムの型	アルゴリズム
	ベクトルのノルム	
"1"	1 ノルム	$\|\|a\|\|_1 = \|a_1\| + \|a_1\| + ... + \|a_n\|$
"2"	2 ノルム	$\|\|a\|\|_2 = (\|a_1\|^2 + \|a_2\|^2 + ... + \|a_n\|^2)^{1/2}$
"p"	p ノルム	$\|\|a\|\|_p = (\|a_1\|^p + \|a_2\|^p + ... + \|a_n\|^p)^{1/p}$
		"p"は浮動小数点数
"i"	無限ノルム	$\|\|a\|\|_\infty = \max_i \|a_i\|$
"-i"	負の無限ノルム	$\|\|a\|\|_{-\infty} = \min_i \|a_i\|$
	m x n 行列のノルム	
"1"	1 ノルム	$\|\|a\|\|_1 = \max_j \sum_{i=1}^{m} \|a_{ij}\|$
"2"	2 ノルム	$\|\|a\|\|_2 = a$ の最大特異値
"i"	無限ノルム	$\|\|a\|\|_\infty = \max_i \sum_{j=1}^{m} \|a_{ij}\|$
"f"	フロベニウスのノルム	$\|\|a\|\|_F^2 = \sum_{i=1}^{m} \sum_{j=1}^{n} \|a_{ij}\|^2$
"m"	ノルム	$\|\|a\|\| = \max(\mathbf{abs}(A[i][j])))$

次に例を示します。

```
> array double a[6] = {1, 2, 3, 6, 5, 4}
> array double b[2][3] = {1, 2, 3, 6, 5, 4}
> norm(a, "1")
21.0000
> norm(b, "1")
7.0000
```

24.2.11 階乗

階乗は次の式で定義されます。

$$f = n!$$

関数 **factorial()** のプロトタイプは

```
unsigned long long factorial(unsigned int n);
```

であり、階乗の計算に使用します。たとえば、階乗 3! は、次のコマンドで評価できます。

```
> factorial(3)
6
```

24.2.12　組み合わせ

関数 **combination()** のプロトタイプは次のとおりです。

```
unsigned long long combination(unsigned int n, unsigned int k);
```

n 個から重複なしに一度に k 個を選択する組み合わせの数を計算できます。組み合わせの数は、各セットに種類の異なる k 個を含む、n 個からなるセットの数です。まったく同じ種類の k 個を含む 2 つのセットはありません。次の式で定義されます。

$$C_k^n = \frac{n!}{(n-k)!k!}$$

たとえば、組み合わせ C_2^5 は、次のコマンドで評価できます。

```
> combination(5, 2)
10
```

24.2.13　データの並べ替え

配列または変数リストに含まれる最小値または最大値およびその位置を特定する方法は、セクション 24.2.3で説明されています。関数 **minloc()** と関数 **maxloc()** は、 最小値と最大値のインデックス (最初の要素がインデックス 0) を特定します。データの並べ替えには、ヘッダーファイル **stdlib.h** で定義されている C の標準関数 **qsort()** を使用できます。このセクションでは、配列に格納されているデータを並べ替える方法を説明します。関数 **findvalue()** のプロトタイプは

```
int findvalue(array int y[&], array double complex x[&]);
```

であり、配列内のゼロ以外の要素のインデックスを特定します。ベクトル y は、x の要素数の整数型データ配列であり、配列 x 内のゼロ以外の要素のインデックスを含んでいます。y の残りの要素には-1 の値が含まれています。x が complex 型データの配列の場合は、ゼロのインデックスが付いた要素は、実数部、虚数部ともにゼロを含む値として定義されます。関数 **findvalue()** は、配列 x のゼロ以外の要素の数を返します。次に例を示します。

```
> array double x[5]={0,43.4,-7,-2.478,7}
> array int y[5]
> findvalue(y,x)
4
> y
1 2 3 4 -1
> findvalue(y,x<0)
2
> y
2 3 -1 -1 -1
```

関数 **sort**() のプロトタイプは

```
int sort(array double complex &y, array double complex &x, ...
              /* [string_t method], [array int &ind] */);
```

であり、要素を昇順に並べ換えて、ランク付けします。配列 *x* の元データでは、サポートされている任意の算術データ型および次元を使用できます。配列 *y* は、*x* と同じデータ型とサイズで、並べ替え後のデータが格納されます。*x* が complex 型データの場合、データは各要素の大きさを基準に並べ換えられます。*x* が NaN または ComplexNaN 要素を含むデータの場合、**sort**() はこれらのデータを末尾に置きます。配列 *ind* のインデックスには、並べ替えられた要素の配列 *x* でのインデックス（0 から始まる）インデックスが含まれています。オプション引数 *method* で並べ換えの方法を指定します。*x* が 2 次元配列の場合は、このパラメータを次のように指定して並べ換え方法を指定します。“array” (すべての要素を基準に並べ換える)、“row” (2 次元配列で行を基準に並べ換える)、“column”(2 次元配列で列を基準に並べ換える)。既定では、2 次元配列はすべての要素を基準に並べ換えられます。*x* が 2 次元配列以外の場合、配列はすべての要素を基準に並べ換えられます。関数 **sort**() は、成功すると 0 を返し、失敗すると-1 を返します。次に例を示します。

```
> array double x[4] = {0.1, NaN, -0.1, 3}, y[4]
> sort(y, x)
> y
-0.1000 0.1000 3.0000 NaN
> array double x2[2][3] = {5.0, NaN, -3.0, -6.0, 4.0, 3.0}, y2[2][3]
> array int ind[2][3];
> sort(y2, x2, "array", ind)
> y2
-6.0000 -3.0000 3.0000
4.0000 5.0000 NaN
> ind
3 2 5
4 0 1
> sort(y2, x2, "row", ind)
> y2
-3.0000 5.0000 NaN
-6.0000 3.0000 4.0000
> ind
2 0 1
0 2 1
```

24.2.14　アンラップ

関数 **unwrap**() のプロトタイプは

```
int unwrap(array double &y, array double &x, ...
         /* [double cutoff] */);
```

であり、π より大きい絶対ジャンプを 2π 補数へ変更することにより、入力される配列 x の各要素のラジアン位相をアンラップします。配列 x は、ベクトルまたは 2 次元配列のいずれかにできます。2 次元配列の場合、関数は配列のすべての行のラジアン位相をアンラップします。
配列引数 y は、x と同じ次元とサイズになります。アンラップ後のデータが含まれます。オプション引数 *cutoff* で、ジャンプ値を指定します。

cutoff の値を指定しない場合は、既定で π の値が指定されます。関数 **unwrap** は、成功すると 0 を返し、失敗すると-1 を返します。たとえば、プログラム 24.1を使用したクランクロッカーのメカニズムの動作解析では、ロッカーの解析結果は $0 \sim 2\pi$ の範囲になります。

```
#include <math.h>
#include <complex.h>
#include <chplot.h>

int main(){
    double r[1:4],theta1,theta31;
    int n1=2,n2=4, i;
    double complex z,p,rb;
    double  x1,x2,x3,x4;
    array double theta2[36],theta4[36],theta41[36];
    class CPlot subplot, *plot;

    /* four-bar linkage*/
    r[1]=5; r[2]=1.5; r[3]=3.5; r[4]=4;
    theta1=30*M_PI/180;
    linspace(theta2,0,2*M_PI);
    for (i=0;i<36;i++) {
       z=polar(r[1],theta1)-polar(r[2],theta2[i]);
       complexsolve(n1,n2,r[3],-r[4],z,x1,x2,x3,x4);
       theta4[i] = x2;
    }
    unwrap(theta41, theta4);
    subplot.subplot(2,1);
    plot = subplot.getSubplot(0,0);
    plot->data2D(theta2, theta4);
    plot->title("Wraped");
    plot->label(PLOT_AXIS_X,"Crank input: radians");
    plot->label(PLOT_AXIS_Y,"Rocker output: radians");

    plot = subplot.getSubplot(1,0);
    plot->data2D(theta2, theta41);
    plot->title("Unwraped");
    plot->label(PLOT_AXIS_X,"Crank input: radians");
    plot->label(PLOT_AXIS_Y,"Rocker output: radians");
    subplot.plotting();
}
```

プログラム 24.1: **unwrap()** を使用したプログラム例

このメカニズムでは、解析結果は図 24.1の上の図に示すような結果になります。θ_4 が π のときにジャンプが発生します。これは、クランクロッカーのメカニズムでは、$\theta_4 = \pi$ と $\theta_4 = -\pi$ が同じ点になるからです。関数 **unwrap()** を使用すると、図 24.1の下の図に示すように、出力角度 θ_4 は滑らかなカーブを描きます。

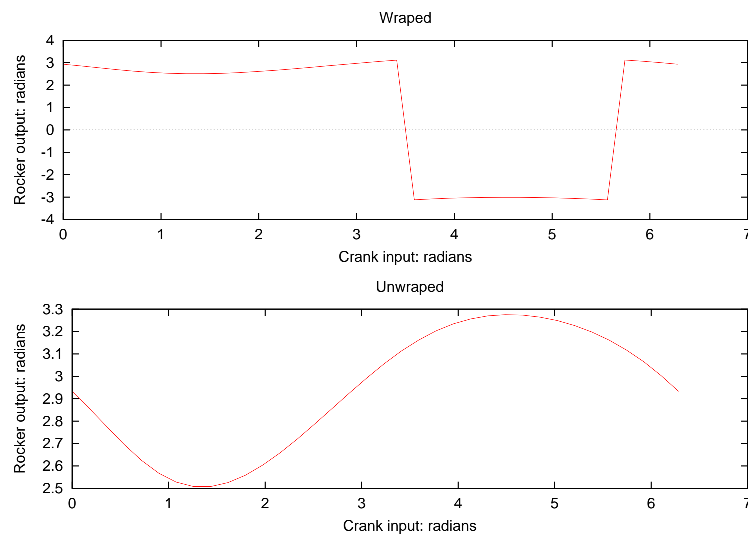


図 24.1: 関数 **unwrap()** を使用した場合と使用しない場合の比較

24.2.15　配列の要素に適用する関数

関数 **fevalarray()** のプロトタイプは

```
int fevalarray(array double &y, double (*func)(double),
    array double &x, ... /* [array int &mask, double value] */);
```

であり、配列 *x* の各要素に適用されたユーザー定義関数を評価します。配列には、任意の算術データ型および次元を入力できます。

配列の引数 *y* には、関数の評価結果が格納されます。引数 *func* は、ユーザーが定義した関数へのポインタです。配列 *mask* の要素の値が 1 の場合、関数の評価結果はその要素に適用されます。配列 *mask* の要素の値が 0 の場合、5 番目のオプション引数 *value* の値がその要素に使用されます。関数 **fevalarray()** は、成功すると 0 を返し、失敗すると-1 を返します。次に例を示します。

```
> double func(double x) {return x*x;}
> array double x[4] = {1, 2, 3, 4}, y[4]
> array int mask[4] = {1, 0, 1, 0}
> fevalarray(y, func, x);
> y
1.0000 4.0000 9.0000 16.0000
> fevalarray(y, func, x, mask, 5);
> y
1.0000 5.0000 9.0000 5.0000
```

複素数の配列の場合は、次のプロトタイプを持つ関数 **cfevalarray()** のプロトタイプを使用する必要があります。

```
int cfevalarray(array double complex &y,
    double complex (*func)(double complex), array double complex &x,
    ... /* [array int &mask, double complex value] */);
```

24.2.16 ヒストグラム

関数 **histogram()** のプロトタイプは

```
int histogram(array double &y, array double x[&], ...
                /* [array double hist[:]] */);
```

であり、配列 *x* で定義したビンを持つデータセット *y* のヒストグラムを計算できます。データセット *y* の配列は、どのような次元と実数型の配列にもできます。関数呼び出しでオプション引数 *hist* を指定しない場合、ヒストグラムはプロットされます。オプション引数 *hist* を指定した場合、プロットは行われず、ヒストグラムデータが配列 *hist* に格納されます。たとえば、次のコマンドを指定できます。

```
> array double yy[300], xx[300], x[21]
> lindata(-1, 1, x)
> lindata(-3.14, 3.14, xx)
> yy = sin(xx)
> histogram(yy, x)
```

上記のコマンドの出力結果は、図 24.2に示すとおりです。

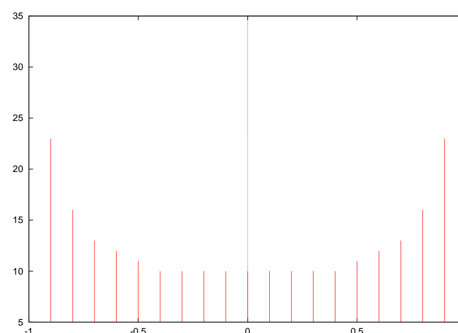


図 24.2: $-\pi <= x <= \pi$ の範囲の sin(x) のヒストグラム

24.3 データ補間とカーブフィッティング

24.3.1 1 次元補間

関数 **interp1()** のプロトタイプは

```
int interp1(double y[&], double x[&], double xa[&], double ya[&],
            char *method);
```

であり、直線または立体スプラインの補間によって配列 *x* に表現されたプロット点で、配列 *xa* と *ya* の条件で表現された関数の値を特定します。入力引数 *method* に`"linear"`または`"spline"`の文字列を指定して、補間方法を選択できます。関数 **interp1**() は、成功すると 0 を返し、失敗すると-1 を返します。たとえば、次のコマンドは有効です。

```
> array double x[1]=0.5, y[1], xa[180],ya[180]
> lindata(-3.14, 3.14, xa)
> ya = sin(xa)
> interp1(y, x, xa, ya, "spline")
> y
0.4794
```

プログラム 24.2で生成した図 24.3では、元データと、関数 **interp1**() を使用して補間されたプロット点を示します。

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

int main() {
    array double xa[36], ya[36];
    array double x[12], y[12];
    class CPlot plot;
    int numdataset=0, pointtype=7, pointsize=1;

    linspace(xa, -M_PI, M_PI);
    linspace(x,  -M_PI, M_PI);
    ya = sin(xa);
    interp1(y, x, xa, ya, "spline");

    plot.data2D(xa, ya);
    plot.plotType(PLOT_PLOTTYPE_POINTS, numdataset, pointtype, pointsize);
    plot.legend("original points", 0);
    numdataset=1, pointtype=1, pointsize=2;
    plot.data2D(x, y);
    plot.plotType(PLOT_PLOTTYPE_POINTS, numdataset, pointtype, pointsize);
    plot.legend("interpolated points", 1);
    plot.legendLocation(-1, 0.8);
    plot.plotting();
}
```

プログラム 24.2: 関数 **interp1**() を使用したプログラム

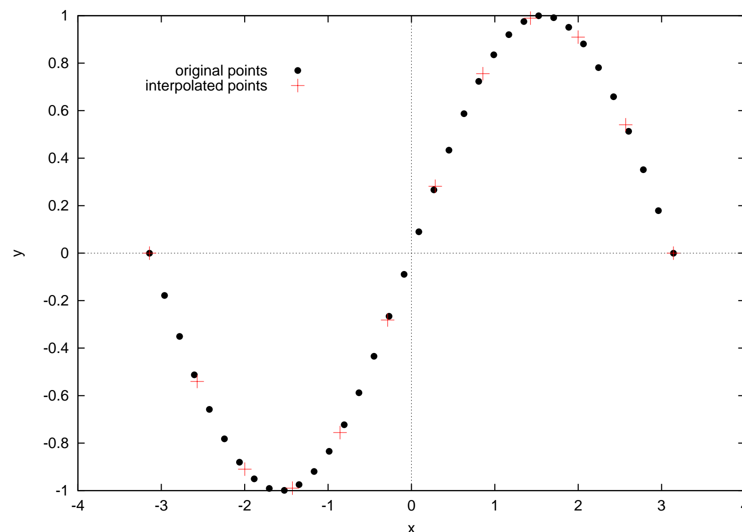


図 24.3: 元データと関数 **interp1()** を使って補間したプロット点

24.3.2 2次元補間

関数 **interp2()** のプロトタイプは

```
int interp2(double z[&][&], double x[&], double y[&],
    double xa[&], double ya[&], double za[&][&], char *method);
```

であり、直線または立体スプラインの補間によって配列 *x* と配列 *y* の 2 つの 1 次元配列で表現されたプロット点で 2 次元関数の値を特定します。関数は、次元 *m* の配列 *xa* と次元 n の配列 *ya* で、および *xa* と *ya* で定義される格子ポイントで表形式にされた次元 $m \times n$ の関数値 *za* の行列で、表形式の値で表現されます。配列 *xa*、*ya*、*x*、および *y* の次元は異なる場合もあります。入力引数 *method* に "linear"または"spline"の文字列を指定して、補間方法を選択します。関数 **interp2()** は、成功すると 0 を返し、失敗すると-1 を返します。

プログラム 24.3は、立体スプライン
$z(x,y) = 3(1-x)^2 e^{-x^2-y^2+1} - 10(\frac{x}{5} - x^3 - y^5)e^{-x^2-y^2} - \frac{1}{3}e^{-(x+1)^2-y^2}$
を使用して、2 次元関数を補間します。

```
#include <chplot.h>
#include <math.h>
#include <numeric.h>
#define M 20
#define N 30
#define NUM_X 40
#define NUM_Y 50

int main() {
    int datasetnum = 0, i,j;
    array double za1[M*N],za[M][N],xa[M],ya[N];
    array double z1[NUM_X*NUM_Y],z[NUM_X][NUM_Y],x[NUM_X],y[NUM_Y];
    class CPlot plot;

    /* Construct data set of the peaks function */
    linspace(xa, -3, 3);
    linspace(ya, -4, 4);
    for(i=0; i<M; i++) {
        for(j=0; j<N; j++) {
            za[i][j] = 3*(1-xa[i])*(1-xa[i])*
                       exp(-(xa[i]*xa[i])-(ya[j]+1)*(ya[j]+1))
                       - 10*(xa[i]/5 - xa[i]*xa[i]*xa[i]-
                       pow(ya[j],5))*exp(-xa[i]*xa[i]-ya[j]*ya[j])
                       - 1/3*exp(-(xa[i]+1)*(xa[i]+1)-ya[j]*ya[j]);
        }
    }
    linspace(x, -3, 3);
    linspace(y, -4, 4);
    interp2(z,x,y,xa,ya,za,"spline");

    za1 = (array double[M*N])za;
    /* add offset for display */
    z1 = (array double[NUM_X*NUM_Y])z + (array double[NUM_X*NUM_Y])50;
    plot.data3D(xa, ya, za1);
    plot.plotType(PLOT_PLOTTYPE_LINES, datasetnum++);
    plot.data3D(x, y, z1);
    plot.plotType(PLOT_PLOTTYPE_LINES, datasetnum++);
    plot.ticsLevel(0);
    plot.text("spline", PLOT_TEXT_RIGHT,4,4.5,55);
    plot.text("original", PLOT_TEXT_RIGHT,4,4.5,5);
    plot.colorBox(PLOT_OFF);
    plot.plotting();
}
```

プログラム 24.3: 関数 **interp2**() を使用したプログラム

2 次元配列の `z` と `za` は 1 次元配列の `z1` と `za1` にキャストされ、メンバ関数 **CPlot**::**data3D**() をプロットすることで使用できるようになります。プログラム 24.3の出力結果は、図 24.4に示すとおりです。

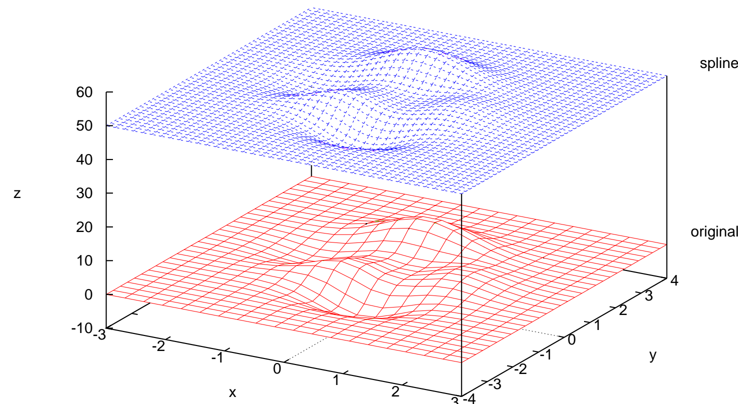


図 24.4: 2 次元補間関数 **interp2()** の出力結果

24.3.3 一般的なカーブフィッティング

フィッテイング式の一般的な形式は次のとおりです。

$$y(x) = \sum_{k=1}^{M} a_k f_k(x)$$

ここで、$f_1(x), ..., f_M(x)$ は x の任意の固定関数です。「ベース関数」とも呼ばれます。$f_k(x)$ は、x の不連続関数です。係数 a_k は、カイ 2 乗値を最小化することで求められます。次の式で定義されます。

$$\chi^2 = \sum_{i=1}^{N} \left(\frac{y_i - \sum_{k=1}^{M} a_k f_k(x_i)}{\sigma_i} \right)^2$$

ここで、σ_i は、$\sigma = 1$ 番目のデータ点の標準偏差です。標準偏差が未知の場合は、定数値 $\sigma = 1$ に設定できます。要素 (k, j) を持つ $n \times m$ 行列 α は、次の式で定義されます。

$$\alpha_{kj} = \sum_{i=1}^{N} \frac{f_j(x_i) f_k(x_i)}{\sigma_i^2}$$

共分散行列の要素 (k, j) は、次のように表すことができます。

$$cover_{kj} = [\alpha]_{kj}^{-1}$$

関数 **curvefit()** のプロトタイプは

```
int curvefit(double a[&], double x[&], double y[&],
    void (*funcs)(double, double []), ...
  /* double sig[], int ia[], double covar[:][:], double *chisq] */);
```

であり、同じ次元を持つ配列 *x* と *y* のデータ点のセットと配列 *sig* 内の各標準偏差を与えると、χ^2 の最小化が行われ、$y = a_i * funcs_i(x)$ に直線的に依存する関数の配列 a に含まれる係数の一部またはすべてが補正されます。プログラムは、χ^2 と共分散行列 *covar* を渡すことができます。配列引数 *ia* の値が定数の場合、共分散行列の対応する成分はゼロになります。

ユーザーは、配列 func によって *x* で評価された関数の値を渡すルーチン *funcs*(*x*,*func*) を指定します。それらが未知の場合、配列 sig の標準偏差を 1 に設定できます。関数 **curvefit**() は、成功すると 0 を返し、失敗すると-1 を返します。

```
#include <stdio.h>
#include <math.h>
#include <numeric.h>
#define NPT 100
#define SPREAD 0.1
#define NTERM 4

void funcs(double x, double func[]) {
    func[0]=cos(x);
    func[1]=sin(x);
    func[2]=exp(x);
    func[3]=1.0;
}
int main(void) {
    int i,j;
    array double a[NTERM],x[NPT],y[NPT],sig[NPT],u[NPT];

    linspace(x, 0, 10);
    urand(u);
    y = 4*cos(x) + 3*sin(x) +2*exp(x) +(array double [NPT])1.0 + SPREAD*u;
    curvefit(a,x,y,funcs);
    printf("    %s\n","parameters");
    for (i=0;i<NTERM;i++)
       printf("    a[%d] = %f\n", i,a[i]);
}
```

プログラム 24.4: **curvefit**() を使用したプログラム例

プログラム 24.4は、一様乱数偏差を適用して $f(x) = 4\cos(x) + 3\sin(x) + 2e^x + 1$ で生成されたデータ点をベース関数 $\cos(x)$、$\sin(x)$、e^x、および 1 に補正します。この例では、カーブフィッティング関数 (*a*, *x*, *y*, *funcs*) が呼び出されます。このプログラムからの出力結果は次のとおりです。

```
parameters
a[0] = 3.990255
a[1] = 2.994660
a[2] = 2.000000
a[3] = 1.051525
```

24.3.4 多項式関数を使用するカーブフィッティング

関数 **polyfit**() のアルゴリズムは関数 **curvefit**() のアルゴリズムに基づいています。多項式のカーブフィッティングでは、内部処理で、**curvefit**() のベース関数が多項式の条件に設定されます。フィッティング式の一般的な形式は次のとおりです。

$$y(x) \quad = \quad \sum_{k=1}^{M} a_k x^k$$

関数 **polyfit**() のプロトタイプは

```
int polyfit(double a[&], double x[&], double y[&],
  /* double sig[], int ia[], double covar[:][:], double *chisq] */);
```

であり、同じ次元を持つ配列 *x* と *y* のデータ点のセットと配列 *sig* の各標準偏差を与えると、χ^2 の最小化が行われ、配列 *a* の多項式の係数が補正されます。プログラムは、χ^2 と共分散行列 *covar* を渡すことができます。配列引数 *ia* の値が定数の場合、共分散行列の対応する成分はゼロになります。それらが未知の場合、配列 *sig* の標準偏差を 1 に設定できます。関数 **polyfit**() は、成功すると 0 を返し、失敗すると-1 を返します。

```
#include <stdio.h>
#include <numeric.h>
#define NPT 100                  /* Number of data points */
#define NTERM 5                  /* Number of terms */

int main() {
    int i,j,status;
    array double  u[NPT],x[NPT],y[NPT],a[NTERM];

    /* Create a data set of NTERM order polynomial with uniform random deviation*/
    linspace(x,0.1,0.1*NPT);
    y = 8*x.*x.*x.*x + 5*x.*x.*x + 3*x.*x + 6*x  + (array double[NPT])(7);
    urand(u);
    y += 0.1*u;

    status=polyfit(a,x,y);
    if(status) printf("Abnormal fit");
    printf("    %s\n","Coefficients");
    for (i=0;i<NTERM;i++)
        printf("    a[%1d] =  %8.6f \n",i,a[i]);
    printf("    y = %f at x = %f\n", polyeval(a, 2.0), 2.0);
}
```

プログラム 24.5: **polyfit**() を使用したプログラム例

プログラム 24.5は、次の多項式で生成されるデータ点を一定乱数偏差で補正します。

$$8x^4 + 5x^3 + 3x^2 + 6x + 7$$

多項式のカーブフィッティング関数 **polyfit**(*a*, *x*, *y*) を使用して、四次多項式の配列 *a* の係数を求めることができます。このプログラムからの出力結果は次のとおりです。

```
Coefficients
a[0] =  8.000048
a[1] =  4.999537
a[2] =  2.998743
a[3] =  6.015915
a[4] =  7.033636
y = 199.057500 at x = 2.000000
```

24.4 関数の最小化または最大化

このセクションでは、関数の最小化について説明します。関数 $f(x)$ の最大値は、関数 $-f(x)$ の最小値を特定することで求めることができます。

24.4.1 1つの変数を含む関数の最小化

関数 **fminimum()** のプロトタイプは

```
int fminimum(double *fminval, double *xmin, double (*func)(double),
double x0, double xf, ...  /* [double rel_tol], [double abs_tol]*/);
```

であり、変数を1つ含む関数の最小値を求め、その最小値に対応する位置を特定します。この関数によって、実数関数 func が最小値と見なす、*x0* と *xf* の間のデータ点が決定されます。ユーザーが定義する関数 *func* には、*x* 値を入力する引数があります。引数 *fminval* は、関数が計算した最小値を渡します。引数 *xmin* には、関数の最小値が特定される位置が含まれます。許容値は、x: $|x|rel_tol + abs_tol$ の関数で定義されます。ここで、*rel_tol* は相対精度で、*abs_tol* は絶対精度 (ゼロを指定することはできません) です。*rel_tol* と *abs_tol* の既定値は 10^{-6} です。関数 **fminimum()** は、成功すると 0 を返し、失敗すると-1 を返します。

関数が最小値を取る区間では、あるデータ点 *u* に対して、(a) *func* が [a,u) で厳密に単調減少し、[u,b] で厳密に単調増加するか、または (b) これら 2 つの区間がそれぞれ [a,u] と (u,b] で置き換えられるかのいずれかになると考えられます。

たとえば、以下のコマンドで、図 24.5に示されている次の関数の最小値を求めることができます。

$$f(x) = -\frac{1}{(x-0.3)^2+0.01} - \frac{1}{(x-0.9)^2+0.04} + 6$$

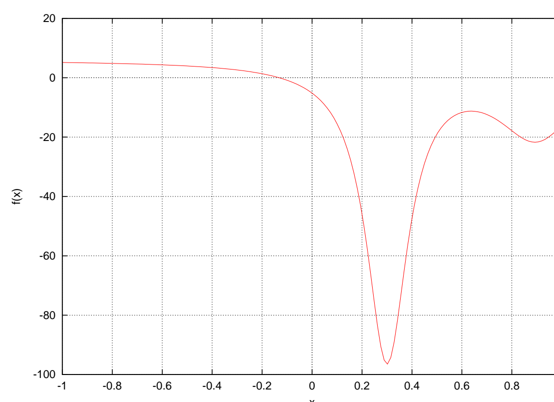


図 24.5: 関数 $f(x) = -\frac{1}{(x-0.3)^2+0.01} - \frac{1}{(x-0.9)^2+0.04} + 6$

ここで、区間は [-1,0]、[0,0.8]、[0.4,1]、および [0,1] で、許容値として既定値が使われます。区間が [0,1] の場合、この関数は、$x = 0.892716.$ の極小値のみを算出します。

```
> double xmin, fminval
> double func(double x) { return -1/((x-0.3)*(x-0.3)+0.01) -\
  1/((x-0.9)*(x-0.9)+0.04) + 6;}
> fminimum(&fminval, &xmin, func, -1, 0)
> xmin
0.0000
> fminval
-5.1765
> fminimum(&fminval, &xmin, func, 0, 0.8)
> xmin
0.3003
> fminval
-96.5014
> fminimum(&fminval, &xmin, func, 0.4, 1)
> xmin
0.4000
> fminval
-47.4481
> fminimum(&fminval, &xmin, func, 0, 1)
> xmin
0.8927
> fminval
-21.7346
```

24.4.2 複数の変数を含む関数の最小化

関数 **fminimums()** のプロトタイプは

```
int fminimums(double *fminval, double xmin[&],
      double (*func)(double[]), double x0[&], ...
      /* [double rel_tol], [double abs_tol], [int numfuneval] */);
```

であり、変数を複数含む関数の最小値を求め、その最小値に対応する位置を特定します。n 次元の関数と、ユーザーが入力した初期推定値を含む配列を与えると、この関数は関数が最小値を持つ位置の配列を計算します。次元の数は、内部処理で、入力された配列 *x* から取られます。関数 *func* は、*x* によって与えられるデータ点で最小化される関数の値を求めます。許容値は、$|x|rel_tol + abs_tol$ の関数で定義されます。ここで、*rel_tol* は相対精度で、*abs_tol* は絶対精度 (ゼロを指定することはできません) です。

引数 *numfunceval* は、許容される関数評価の最大数です。*rel_tol*、*abs_tol*、および *numfuneval* の既定値は、それぞれ、10^{-6}、10^{-6} および 250 です。関数 **fminimums()** は、成功すると 0 を返し、失敗すると負の値を返します。

たとえば、図 24.6の関数 $f(x_0, x_1) = 100(x_1 - x_0^2)^2 + (1.0 - x_0)^2$ は、$\mathbf{x} = (1, 1)$ で最小値 0 を取ります。

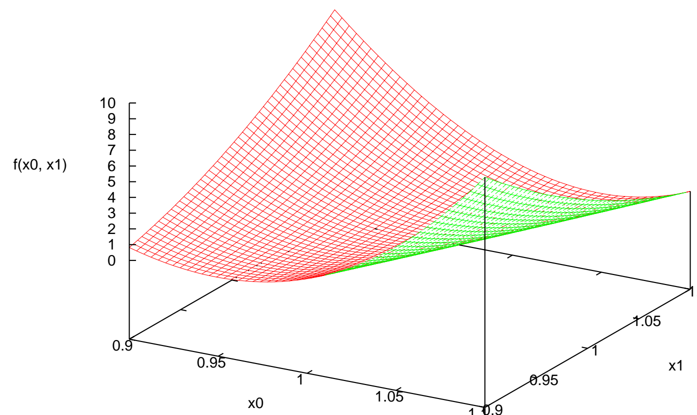


図 24.6: 関数 $f(x_0, x_1) = 100(x_1 - x_0^2)^2 + (1.0 - x_0)^2$.

この最小値とその位置は、$\mathbf{x_0} = (-1.2, 1.0)$ の初期推定値を含む関数 **fminimums()** を使用して、次のコマンドで求めることができます。

```
> double fminval;
> array double xmin[2], x0[2]= {-1.2, 1.0};
> double func(double x[]) { return 100*(x[1]-x[0]*x[0])* \
  (x[1]-x[0]*x[0]) + (1.0-x[0])*(1.0-x[0]);}
> fminimums(&fminval, xmin, func, x0);
> xmin
1.0000 1.0000
> fminval
0.0000
```

24.5　多項式

このセクションでは、多項式に関する関数を説明します。Ch では、多項式は、係数を降べきの順番に並べた配列で表されます。たとえば、次の多項式

$$p_0 x^n + p_1 x^{n-1} + ... + p_{n-1} x + p_n$$

は、$[p_0, p_1, ..., p_{n-1}, p_n]$ の $(n+1)$ 要素を含む配列で表されます。関数 **polyfit()** で多項式を使用するカーブフィッティングについては、セクション 24.3.4で説明しています。**polyevalm()** を使用する行列の行列多項式の評価については、セクション 24.10.4で説明します。関数 **conv()** と関数 **deconv()** をそれぞれ 2 つの多項式の乗算および除算に使用する方法については、セクション 24.15で説明します。

24.5.1 多項式の評価

関数 **polyeval()** のプロトタイプは

```
double polyeval(array double p[&], double x, ...
                /* array double dp[&] */);
```

であり、多項式とその導関数を x で評価します。多項式は、係数を使用して、配列 p で表されます。関数 **polyeval()** は x での多項式の値を返します。配列の double 型と次元 m のオプション引数 *dp* を与えると、関数 **polyeval()** には、多項式の導関数の値が 1 番目から m 番目の順番で含まれます。

たとえば、データ点 $x = 0.1$ の多項式 $P(x) = x^5 - 5x^4 + 10x^3 - 10x^2 + 5x - 1$ その 1 番目、2 番目、および 3 番目の導関数は、次のコマンドで求めることができます。

```
> array double p[6] = {1.0, -5.0, 10.0, -10.0, 5.0, -1.0}
> array double dp[3];
> polyeval(p, 0.1, dp);
-0.5905
> dp
3.2805 -14.5800 48.6000
```

多項式の係数または変数が複素数の場合、次のプロトタイプを持つ関数 **cpolyeval()**

```
double complex cpolyeval(array double complex p[&], double complex x);
```

が、多項式の評価に使用されます。関数 **polyevalarray()** のプロトタイプは

```
int polyevalarray(array double complex &y, array double complex p[&],
                  array double complex &x);
```

であり、点列で多項式を評価します。多項式の係数を含むベクトル p では、サポートされている任意の算術データ型およびサイズを使用できます。x の値はどのような算術型も取ることができます。データは、内部処理で double complex 型へ変換されます。x と同じ次元とサイズを持つ配列 *val* には、配列 x によって表されるデータ点の多項式の値が含まれます。c と x の両方が実数型であるなら、*val* は実数型です。それ以外の条件では、複素数型になります。関数 **polyevalarray()** は、成功すると 0 を返し、失敗すると-1 を返します。

たとえば、データ点 $\mathbf{x} = \{0, 0.5, 1.0, 1.5, 2.0\}$ に対する多項式 $P(x) = x^5 - 5x^4 + 10x^3 - 10x^2 + 5x - 1$ は、次のコマンドで求めることができます。

```
> array double p[6] = {1.0, -5.0, 10.0, -10.0, 5.0, -1.0}
> array double val[5], x[5]= {0, 0.5, 1, 1.5, 2}
> polyevalarray(val, p, x)
> val
-1.0000 -0.0313 0.0000 0.0313 1.0000
```

多項式の変数 x が正方行列の場合、その種の行列多項式の評価は、関数 **polyevalm()** を使用して実行できます。これについては、セクション 24.10.4で説明します。

24.5.2　多項式の導関数

次のように、x_i の係数 ($i = 0, 1, \cdots, N$) を持つ多項式 $P(t)$ の場合、

$$P(t) = x_0 t^n + x_1 t^{n-1} + x_2 t^{n-2} + \cdots + x_{n-1} t + x_n,$$

この多項式の導関数は次のようになります。

$$\begin{aligned} P'(t) &= n * x_0 t^{n-1} + (n-1) * x_1 t^{n-2} + (n-2) * x_2 t^{n-3} + \cdots + x_{n-1} \\ &= y_0 t^{n-1} + y_1 t^{n-1} + y_2 t^{n-3} + \cdots + y_{n-1} \end{aligned}$$

関数 **polyder()** のプロトタイプは

```
int polyder(array double complex y[&], array double complex x[&]);
```

であり、多項式 $P'(x)$ の導関数 $P(x)$ の係数を求めます。多項式 x の係数のベクトルは、サポートされていれば、任意の算術データ型とサイズのベクトルを取ることができます。データは、内部処理で double complex 型へ変換されます。ベクトル x がサイズ n を持つ実数型の場合、導関数のベクトル y はサイズ $(n-1)$ を取る実数型です。ベクトル x が複素数型の場合は、導関数のベクトル y は複素数型になります。

たとえば、多項式 $P(x) = x^5 - 5x^4 + 10x^3 - 10x^2 + 5x - 1$ の導関数は、$P'(x) = 5x^4 - 20x^3 + 30x^2 - 20x + 5$ です。P0(x) の係数は、次のコマンドで求めることができます。

```
> array double x[6] = {1.0, -5.0, 10.0, -10.0, 5.0, -1.0}
> array double y[5]
> polyder(y, x)
> y
5.0000 -20.0000 30.0000 -20.0000 5.0000
```

次の 2 つの多項式 $U(x)$ と $V(x)$

$$\begin{aligned} U(x) &= u_0 x^n + u_1 x^{n-1} + u_2 x^{n-2} + \cdots + u_{n-1} x + u_n \\ V(x) &= v_0 x^m + v_1 x^{m-1} + v_2 x^{m-2} + \cdots + v_{m-1} x + v_m \end{aligned}$$

の場合、2 つの多項式を乗算する $U(x) * V(x)$ の導関数は、次の式で表されます。

$$Q(x) = (U(x)V(x))' = U'(x)V(x) + V'(x)U(x)$$

また、2 つの多項式を除算する $U(x)/V(x)$ の導関数は、次の式で表されます。

$$\left(\frac{U(x)}{V(x)}\right)' = \frac{Q(x)}{R(x)} = \frac{U'(x)V(x) - V'(x)U(x)}{V^2(x)}$$

ここで、$Q(x)$ と $R(x)$ は、導関数のそれぞれ分子と分母です。2 つの多項式の積または商のいずれかの導関数に適用できる Q(x) の多項式と、$R(x)$ の多項式は次の式で表すことができます。

$$
\begin{aligned}
Q(x) &= q_0x^{n+m-1} + q_1x^{n+m-2} + \cdots + q_{n+m-2}x + q_{n+m-1} \\
R(x) &= r_0x^{2m-1} + r_1x^{2m-2} + \cdots + r_{2m-2}x + r_{2m-1}
\end{aligned}
$$

関数 **polyder**2() のプロトタイプは

```
int polyder2(array double complex q[&], array double complex r[&],
             array double complex u[&], array double complex v[&]);
```

であり、2 つの導関数 $U(x)$ と $V(x)$ の積または商の導関数から、$Q(x)$ と $R(x)$ の係数を求めることができます。2 つの多項式 *u* と *v* の積または商を計算する関数 **polyder**2() 内のアルゴリズムは、引数 *r* によって異なります NULL が引数 *r* に渡される場合、関数 **polyder**2() は 2 つの多項式 *u* と *v* の積 $(u*v)'$ の導関数を計算します。それ以外の条件では、商 $(u/v)'$ の導関数を計算します。多項式 *u* と *v* の係数ベクトルは、サポートされていれば、それぞれ n と m のサイズを持つ任意の算術データ型のベクトルにできます。データは、内部処理で double complex 型へ変換されます。ベクトル *u* と *v* の両方が実数型であれば、サイズ $n+m-2$ のベクトル q とサイズ $2*m-1$ のベクトル r は実数型になります。ベクトル *u* と *v* のどちらか一方が複素数型であるなら、ベクトル q と r は複素数型になります。たとえば、次のような 2 つの多項式があります。

$$
\begin{aligned}
U(x) &= x + 2 \\
V(x) &= x^2 + 2x
\end{aligned}
$$

これらの多項式の積と商の導関数の係数は、次の式で表されます。

$$
\begin{aligned}
(U(x)V(x))' &= 3x^2 + 8x + 4 \\
\left(\frac{U(x)}{V(x)}\right)' &= \frac{-x^2 - 4x - 4}{x^4 + 4x^3 + 4x^2}
\end{aligned}
$$

次のコマンドで、多項式のこれらの係数を求めることができます。

```
> int n=2, m = 3
> array double u[2] = {1.0, 2}, v[3] = {1, 2, 0}
> array double q[2+3-2], r[2*3-1]
> polyder2(q, NULL, u, v)
> q
3.0000 8.0000 4.0000
> polyder2(q, r, u, v)
> q
-1.0000 -4.0000 -4.0000
> r
1.0000 4.0000 4.0000 0.0000 -0.0000
```

24.5.3 代数方程式の根

関数 **roots**() のプロトタイプは

```
int roots(array double complex x[&], array double complex p[&]);
```

であり、代数方程式 $p_0x^n + p_1x^{n-1} + ... + p_{n-1}x + p_n = 0$ の根を計算します。この関数は実数型または複素数型の係数を含む多項式を扱うことができます。引数 p と x には、それぞれ多項式の係数と変数が含まれます。関数 roots() は、成功すると 0 を返し、失敗すると-1 を返します。

たとえば、次のコマンドで、代数方程式 $p = x^2 - 2x + 1 = 0$ の根を求めることができます。

```
> array double x[2]
> array double p[3] = {1, -2, 1}        /* p = x^2-2x+1 */
> roots(x, p)
> x
1.0000 1.0000
>
```

また、**roots**() 関数で、代数方程式の複素数の根の計算もできます。たとえば、代数方程式 $x^4 - 12x^3 + 25x + 116 = 0$ には、複素数の 2 つの根と実数の 2 つの根があります。代数方程式 $(3+i4)x^4 + (4+i2)x^3 + (5+i3)x^2 + (2+i4)x + (1+i5) = 0$ の係数は複素数です。これらの代数方程式の根は、次のコマンドで求めることができます。

```
> array double x1[4], p1[5]  = {1, -12, 0, 25, 116}
> array double complex z1[4]
> roots(x1, p1)
> x1
11.7473 2.7028 NaN NaN
> roots(z1, p1)
> z1
complex(11.7473,0.0000) complex(2.7028,0.0000) \
complex(-1.2251,1.4672) complex(-1.2251,-1.4672)
> array double complex z2[4], p2[5] = {complex(3,4), complex(4,2),\
  complex(5,3), complex(2,4), complex(1,5)}
> roots(z2, p2)
> z2
complex(0.2263,1.2815) complex(0.4311,-0.7280) \
complex(-0.7541,-0.7078) complex(-0.7034,0.5543)
```

24.5.4 代数方程式の係数

代数方程式 $P(x) = 0$ の根が判明している場合、次の式のように、多項式 $P(x)$ を式 $(x - x_i)$ の積として表すことができます。

$$\begin{aligned} P(x) &= (x-x_0)*(x-x_1)*...*(x-x_{n-1}) \\ &= p_0x^n+...+p_{n-1}x+p_n. \end{aligned}$$

代数方程式 $P(x)=0$ の根 x_i を与えると、関数 **polycoef()** で多項式 $P(x)$ の係数 p_i を求めることができます。関数 **polycoef()** のプロトタイプは

```
int polycoef(array double complex p[&], array double complex x[&]);
```

であり、配列引数 x には、多項式の根を与えます。配列引数 p には、計算された多項式の係数が含まれます。代数方程式の根を求める関数 **roots()** は、関数 **polycoef()** と相補的な関係にあるものです。たとえば、次の多項式の係数

$$\begin{aligned} P(x) &= (x-1)(x-2)(x-3)(x-4) \\ &= x^4-10x^3+35x^2-50x+24 \end{aligned}$$

は、次のコマンドで求めることができます。

```
> array double x[4]= {1, 2, 3, 4}
> array double p[5]
> polycoef(p, x)
> p
1.0000 -10.0000 35.0000 -50.0000 24.0000
> roots(x, p)
> x
3.0000 4.0000 2.0000 1.0000
```

24.5.5 多項式の因数分解の剰余

2 つの多項式の比が次のように表される場合を考えます。

$$\frac{U(s)}{V(s)}=\frac{u_0s^m+u_1s^{m-1}+\cdots+u_{m-1}s+u_m}{s^n+v_1s^{n-1}+\cdots+v_{n-1}s+v_n}$$

$V(s)=0$ が重根を持たない場合は、次のように展開されます。

$$\frac{U(s)}{V(s)}=\frac{r_0}{s-p_0}+\frac{r_1}{s-p_1}+\cdots+\frac{r_{n-1}}{s-p_{n-1}}+K(s)$$

p_i が重複度 1 の極の場合は、次のように展開されます。

$$\frac{U(s)}{V(s)}=\frac{r_0}{s-p_0}+\frac{r_1}{s-p_1}+\cdots+\frac{r_i}{(s-p_i)^l}+\cdots+\frac{r_{i+1}}{(s-p_i)^2}+\frac{r_{i+l}}{s-p_i}+\cdots+\frac{r_{n-1}}{s-p_{n-1}}+K(s)$$

ここで、$K(x)$ は直接項です。$m>n$ では、$K(s)$ は次のようになります。

$$K(s)=k_0s^{m-n}+k_1s^{m-n-1}+\cdots+k_{m-n}s+k_{m-n+1}$$

それ以外の条件では、$K(s)$ は空になります。関数 **residue()** によって、2 つの多項式 $V(s)$ と $U(s)$ の比の部分分数を展開するための剰余、極、および直接項が求められます。関数 **residue()** のプロトタイプは

```
int residue(array double u[&], array double v[&],
            array double complex r[&], array double complex p[&],
            array double k[&]);
```

であり、サイズ m のベクトル u とサイズ n のベクトル v で、s の降べきの多項式の係数を指定します。これらのベクトルは実数型にできます。データは、内部処理で、double 型へ変換されます。ベクトルサイズ $(n-1)$ の剰余 r とベクトルサイズ $(n-1)$ の極 p は、剰余計算に従って、互換性のあるいずれかのデータ型になります。実数型の引数を渡して結果が複素数型になった場合は、NaN の値が返されます。$(m-n+1)$ のサイズを持つ直接項 k は、常に実数型になります。$m \leq n$ の場合、k には NULL が含まれます。

たとえば、次のような、分母=0 が実数の単根を持ち、直接項のない部分分数の展開があります。

$$\frac{10s+6}{2s^3+12s^2+22s+12} = \frac{-1}{s+1} + \frac{7}{s+2} + \frac{-6}{s+3}$$

これは、次のコマンドで求めることができます。

```
> int M =2, N =4
> array double u[2] = {10, 6}
> array double v[4] = {2, 12, 22, 12}
> array double r[N-1], p[N-1]
> residue(u, v, r, p, NULL)
> r
-1.0000 7.0000 -6.0000
> p
-1.0000 -2.0000 -3.0000
```

次のような、分母=0 が複素数の単根を持ち、直接項のない部分分数の展開があります。

$$\frac{x+3}{x^2+2x+5} = \frac{0.5+i0.5}{s+(1+i2)} + \frac{0.5-i0.5}{s+(1-i2)}$$

ここでは、次のコマンドを使用できます。

```
> int M =2, N =3
> array double u[2] = {1, 3}
> array double v[3] = {1, 2, 5}
> array double r[N-1], p[N-1]
> array double complex zr[N-1], zp[N-1]
> residue(u, v, r, p, NULL)
> r
NaN NaN
```

```
> p
NaN NaN NaN
> residue(u, v, zr, zp, NULL)
> zr
complex(0.5000,0.5000) complex(0.5000,-0.5000)
> zp
complex(-1.0000,-2.0000) complex(-1.0000,2.0000)
```

分母=0 が単根と直接項を持つ部分分数は次のように展開されます。

$$\frac{2s^3 + 12s^2 + 22s + 12}{10s + 6} = \frac{0.2688}{s + 0.6} + 0.2s^2 + 1.08s + 1.552$$

```
> int M =4, N =2
> array double u[4] = {2, 12, 22, 12}
> array double v[2] = {10, 6}
> array double r[N-1], p[N-1], k[M-N+1]
> residue(u, v, r, p, k)
> r
0.2688
> p
-0.6000
> k
0.2000 1.0800 1.5520
```

分母=0 が重根を持ち、直接項のない部分分数は次のように展開されます。

$$\frac{s^2 + 2s + 3}{s^5 + 5s^4 + 9s^3 + 7s^2 + 2s} = \frac{3}{2(s+2)} - \frac{3}{(s+1)^3} - \frac{2}{s+1} + \frac{3}{2s}$$

項 $(s+1)^2$ の分子がゼロであることに注意してください。

```
> int M =3, N =6
> array double u[3] = {1, 2, 3}
> array double v[6] = {1, 5, 9, 7, 2}
> array double r[N-1], p[N-1]
> residue(u, v, r, p, NULL)
> r
1.5000 -3.0000 0.0000 -2.0000 1.5000
> p
-2.0000 -1.0000 -1.0000 -1.0000 0.0000
```

24.5.6　行列の特性多項式

正方行列 **A** の特性多項式があります。

$$p_0 x^n + ... + p_{n-1} x + p_n$$

は、行列 $(x\mathbf{I} - \mathbf{A})$ の行列式として定義されます。行列 **A** の特性方程式の根は、行列の固有値です。
関数 **charpolycoef**() のプロトタイプは

```
int charpolycoef(array double complex p[&],
                 array double complex a[&][&]);
```

であり、配列 a として渡される行列の特性方程式の係数を計算します。配列の引数 p には、計算された行列の特性多項式の係数が含まれます。関数 **charpolycoef**() は、成功すると 0 を返し、失敗すると-1 を返します。
たとえば、次の行列

$$\mathbf{A} = \begin{bmatrix} 0.8 & 0.2 & 0.1 \\ 0.1 & 0.7 & 0.3 \\ 0.1 & 0.1 & 0.6 \end{bmatrix},$$

の固有値 (1, 0.5, 0.6) および特性多項式

$$x^3 - 2.1x^2 + 1.4x - 0.3$$

の係数は、次のコマンドで求めることができます。

```
> array double a[3][3] = {0.8,0.2,0.1, 0.1,0.7,0.3, 0.1,0.1,0.6}
> array double p[4], x[3]
> charpolycoef(p, a)
> p
1.0000 -2.1000 1.4000 -0.3000
> roots(x, p)
> x
1.0000 0.5000 0.6000
> polycoef(p, x)
> p
1.0000 -2.1000 1.4000 -0.3000
```

24.6　非線形方程式

24.6.1　非線形方程式の解法

関数 **fzero**() のプロトタイプは

```
int fzero(double *x, double (*func)(double), ...
          /* [double x0] | [double x02[2]*/);
```

であり、1 つの変数を持つ非線形関数のゼロ点を求めます。引数 *func* は、ユーザーが定義する関数へのポインタです。関数がゼロになる点は、引数 *x* で渡されます。引数 *x0* には、ゼロ点の初期推定値が含まれます。引数 *x02* は、長さ 2 の double 型のベクトルです。関数は [*x02*[0], *x02*[1]] の区間で有界でなければならず、*func*(*x02*[0]) の符号は *func*(*x02*[1]) の符号と異なる必要があります。そうしないと、エラーが発生します。関数 **fzero**() は、成功すると 0 を返し、失敗すると-1 を返します。たとえば、次の関数

$$f(x) = x^2 - 2$$

のゼロ点 1.414213 は、初期推定値 $x_0 = 2.0$ として、次のコマンドで求めることができます。

```
> double x, func(double x) { return x*x-2.0;}
> fzero(&x, func, 2.0);
> x
1.4142
> double x02[2]={-2, 0}
> fzero(&x, func, x02);
> x
-1.4142
```

24.6.2 非線形連立方程式の解法

関数 **fsolve**() で非線形連立方程式の解を求めることができます。関数 **fsolve**() のプロトタイプは

```
int fsolve(double x[:], void (*func)(double[], double []),
           double x0[:]);
```

であり、配列の引数 *x* と *x0* には、計算されたゼロ位置とその初期推定値がそれぞれ含まれます。ユーザー定義の関数には 2 つの引数があります。最初の引数は入力用で、2 番目の引数は関数の値のためです。入力の引数は n 次元の配列です。計算された関数の値は、同じ次元を持つ 2 番目の配列引数に含まれます。次元の数値は、内部処理で、ユーザーが定義した関数から取られます。関数 **fsolve**() は、成功すると 0 を返し、失敗すると-1 を返します。たとえば、次の 2 つの非線形連立方程式

$$\begin{aligned} f_0 &= -(x_0^2 + x_1^2 - 2.0) = 0 \\ f_1 &= e^{x_0 - 1.0} + x_1^3 - 2.0 = 0 \end{aligned}$$

は、ゼロ点が (1, 1) です。関数 **fsolve**() のゼロ点の初期推定値を $x_0 = 2.0$, and $x_1 = 0.5$ とすると、次のコマンドで解を求めることができます。

```
> void func(double x[], double f[]){f[0]=-(x[0]*x[0]+x[1]*x[1]-2.0);\
  f[1]=exp(x[0]-1.0)+x[1]*x[1]*x[1]-2.0;}
> array double x[2], x0[2] = {2.0, 0.5};
> fsolve(x, func, x0);
> x
1.0000 1.0000
```

24.7 導関数と常微分方程式

24.7.1 差分

関数 **difference()** のプロトタイプは

```
array double difference(array double a[&])[:];
```

であり、配列の隣接する要素間の差分を計算します。入力される 1 次元配列は、実数型です。次に例を示します。

```
> array double a[6] = {1, 2, 10, 4, 5, 6}
> difference(a)
1.0000 8.0000 -6.0000 1.0000 1.0000
```

24.7.2 導関数

関数 derivative() のプロトタイプは

```
double derivative(double (*func)(double), double x, ...
        /* [double &err], [double h]*/);
```

であり、所定のデータ点 *x* で *func* が参照する関数の導関数を数値計算します。戻り値は、データ点 *x* の関数の導関数です。オプション引数 *err* には、導関数の計算に含まれる誤差の予測値が含まれます。これにより、ユーザーは結果を予測できます。オプション引数 *h* を与えると、導関数の計算には初期ステップのサイズ h が使用されます。*h* の値を小さくとる必要はありませんが、*x* の増分よりかなり大きく *func* が変化する必要があります。*h* の引数を代入しない場合、初期ステップのサイズとして、$0.02 * x$ または 0.0001 ($x < 0.0001$ の場合) の値が既定で使用されます。

たとえば、データ点 $x = 2.5$ とすると、関数 $x\sin(x)$ の導関数は次のように計算できます。

```
> double func(double x) {return x*sin(x);}
> derivative(func, 2.5);
-1.404387
```

関数 **derivatives()** のプロトタイプは

```
array double derivatives(double (*func)(double), double x[&], ...
        /* [double &err], [double h]*/)[:];
```

であり、複数のデータ点で関数の導関数を数値計算します。この関数は、配列 *x* に指定されたデータ点数で、関数 *func* の導関数値の配列 **array** を返します。*x* の値にはどのような実数型も指定できます。他の引数は、関数 **derivative()** と同じです。たとえば、$-\pi \le x \le \pi$ の範囲で均等間隔に配置された 36 個のデータ点で、関数 $\sin(x)$ の導関数は、次のコマンドで計算できます。

```
> array double x[36], y[36];
> double func(double x) {return sin(x);}
> lindata(-3.14, 3.14, x);
> y = derivatives(func, x);
> plotxy(x, y);
```

C とは異なり、Ch では、`sin()` などの汎用関数を関数へのポインタに渡すことができないので注意してください。したがって、上記のコードの汎用関数 `sin()` は関数 `func()` でラップされます。上記のコマンドの出力結果は、図 24.7に示すとおりです。

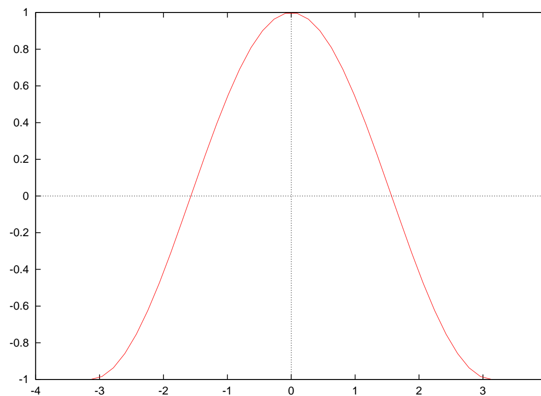


図 24.7: $-\pi <= x <= \pi$ の範囲の sin(x) のヒストグラム

24.8 常微分方程式の解法

関数 **oderk()** のプロトタイプは

```
int oderk( void (*func)(double x, double y[], double dydx[], void *param),
        double t0, double tf, double y0[:], void *param,
        double t[:], double *y, ... /* double tol */);
```

であり、ルンゲクッタ法を使って、次の常微分方程式 (ODE)

$$\frac{dy}{dt} = \mathrm{func}(t, y, p)$$

または、次の常微分連立方程式の解を数値計算します。

$$\frac{d\mathbf{y}}{dt} = \mathbf{func}(t, \mathbf{y}, p)$$

次のいずれかの形式を使用して、呼び出すことができます。引数 `param` を使用して、呼び出し元関数から ODE 関数へ情報を渡します。引数 `func` を 1 階微分方程式の関数へのポインタとして指定します。`t` の初期値および終了値を、引数 *t0* および引数 *tf* としてそれぞれ指定します。配列型の引数 `y0` には、微分方程式の初期値が含まれます。次元の数値は、従属変数の数と等しくなります。成功すれば、*t0* と *tf* の区間で計算されたデータ点の数が返されます。

失敗した場合は、-1 を返します。ベクトル *t* には *t0* と *tf* の間の値が含まれ、その区間内で ODE 関数の解が求められ、結果が変数 y が参照するメモリに格納されます。ユーザーは、常微分方程式のための 1 次元配列 *y* のアドレスか、または常微分連立方程式のための 2 次元配列のアドレスを、関数 **oderk**() の 2 番目の引数に代入する必要があります。オプション引数 *tol* を指定すると、アルゴリズムは、ユーザーが指定した許容値を使用して繰り返し停止を決定します。オプション引数 *tol* の指定がない場合は、既定で、10^{-8} の値が使用されます。通常、ユーザーは関数によって生成されるデータ点の個数を予測し、それに応じて配列 *t* と *y* のサイズを定義しなければなりません。関数 **oderk**() は自動的に、*tf* での結果を *t* および *y* の残余領域に埋め込みます。たとえば、次のような常微分方程式があります。

$$\frac{dy}{dt} = \sin(t)$$

初期条件を $t_0 = -\pi$ および $y_0 = 0$ とすると、次のコマンドを使用して、$t_0 = -\pi \sim t_f = \pi$ の区間の 50 個以上のデータ点で解を求めることができます。

```
> double t[50], y[50], y00[1]={0}
> void func(double t, double y[], double dydt[], void *param) {dydt[0]=sin(t);}
> oderk(func, -3.14, 3.14, y00, NULL, t, y)
34
> plotxy(t, y)
```

上記のコードに示すように、配列 `t` と `y` に 50 個の要素を割り当てましたが、ODE の解法による値は最初の 34 の要素にしか含まれません。残りの要素には、端点 `tf` での値が埋め込まれます。上記のコードの出力結果は、図 24.7に示す結果と同じです。関数 **oderk**() を使用して、次の Van der Pol 方程式を解くことができます。

$$\frac{d^2u}{dt^2} - \mu(1-u^2)\frac{du}{dt} + u = 0$$

ここで、t は $1 \le t \le 30$ の範囲、$\mu = 2$ で、初期条件は $t_0 = 1, u(t_0) = 1$、および $u'(t_0) = 0$ です。Van der Pol 方程式を、まず、2 つの従属変数を持つ 1 セットの 1 階微分方程式として再定式化することができます。y_0 および y_1 を

$$\begin{aligned} y_0 &= u \\ y_1 &= \frac{du}{dt} \end{aligned}$$

と定義すると、Van der Pol 方程式は、次のように再定式化できます。

$$\begin{aligned} \frac{dy_0}{dt} &= \frac{du}{dt} = y_1 \\ \frac{dy_1}{dt} &= \frac{d^2u}{dt^2} = \mu(1-u^2)\frac{du}{dt} - u = \mu(1-y_0^2)y_1 - y_0 \end{aligned}$$

ここで、初期条件は、$t_0 = 1, y_0(t_0) = 1$ および $y_1(t_0) = 0$ です。プログラム 24.6で、$1 \le t \le 30$ の範囲で上記の初期条件を持つ Van der Pol 方程式の解を求めることができます。パラメタ μ は、引数

param を介して関数 main() から ODE 関数へ渡されます。プログラム 24.6の出力結果は、図 24.8に示すとおりです。

```
#include <chplot.h>
#include <numeric.h>

#define NVAR 2
#define POINTS 256
void func(double t, double y[], double dydt[], void *param) {
    double mu;
    mu = *(double*)param;
    dydt[0] = y[1];
    dydt[1]=mu*(1-y[0]*y[0])*y[1] - y[0];
}

int main() {
    double t0=1, tf=30, y0[NVAR] = {1, 0};
    double t[POINTS], y[NVAR][POINTS];
    double mu = 2;

    oderk(func, t0, tf, y0, &mu, t, y);
    plotxy(t, y, "The solution for the van der Pol equation", "t (seconds)", "y1 and y2");
}
```

プログラム 24.6: Van der Pol 方程式の解を求めるプログラム

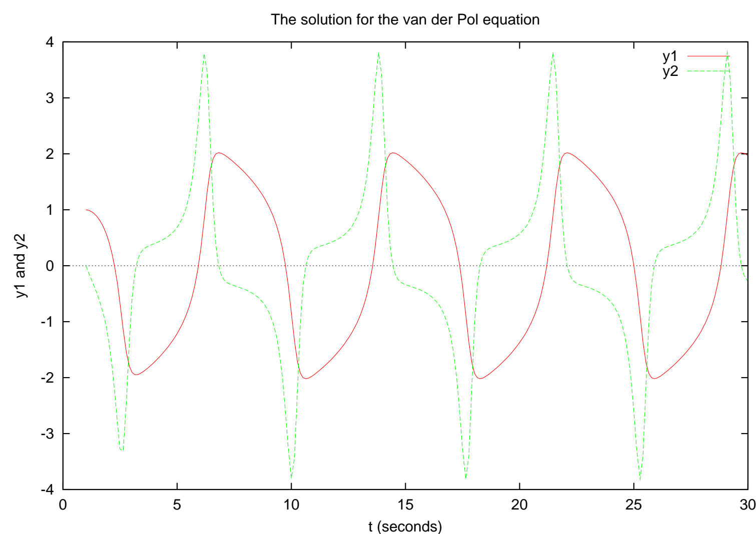


図 24.8: 関数 oderk() の出力結果

別の例として、自由度 2 の動的システムが、次のような連立微分方程式でモデル化できると仮定します。

$$
\begin{aligned}
(1.5 + q_2)\ddot{q}_1 - \dot{q}_1\dot{q}_2 - q_1 &= 0 \\
(1 + q_1)\ddot{q}_2 - \dot{q}_1 - q_2 &= 0
\end{aligned}
$$

上記の方程式は、

$y_0 = q_1, y_1 = \dot{q_1}, y_2 = q_2, y_3 = \dot{q_2}$

と定義することにより、関数 **oderungekutta**() を使用した、数値計算を簡単に行えるような標準形式で次のように再定式化できます。

$$\begin{aligned}\frac{dy_0}{dt} &= y_1\\ \frac{dy_1}{dt} &= \frac{y_0 + y_1 y_3}{1.5 + y_2}\\ \frac{dy_2}{dt} &= y_3\\ \frac{dy_3}{dt} &= \frac{y_1 + y_2}{1 + y_0}\end{aligned}$$

24.9 数値積分

24.9.1 1 次元積分

関数 **integral1**() のプロトタイプは

```
double integral1(double (*func)(double x),
                 double x1, double x2, ... /* [double tol] */);
```

であり、次の積分式を数値積分します。

$$\int_{x_1}^{x_2} f(x)dx$$

引数 *func*(関数へのポインタ) は、積分される関数です。引数の *x1* と *x2* は、区間の端点です。関数は、整数の値を返します。オプション引数 *tol* を代入すると、引数は、繰り返し停止を決めるのに使用されます。代入値がない場合、既定値の $10 * \mathrm{FLT_EPSILON}$ が使用されます。たとえば、次の積分式

$$\int_0^{\frac{\pi}{2}} x^2(x^2 - 2)\sin(x)dx$$

は、次のコマンドで計算できます。

```
> double func(double x) { return x*x*(x*x-2.0)*sin(x);}
> integral1(func, 0, 3.1416/2);
-0.4792
```

24.9.2 2 次元積分

次の 2 次元積分式

$$I = \int_{x_1}^{x_2} dx \int_{y_1}^{y_2} dy f(x, y),$$

の計算には、次のプロトタイプを持つ関数 **integral2**() を使用できます。

```
double integral2(double (*func)(double x, double y),
                 double x1, double x2, double y1, double y2);
```

引数 *func*(関数へのポインタ) は、積分される関数です。引数 *x1* と *x2* は x の端点で、引数 *y1* と *y2* は y の端点です。たとえば、次の積分式

$$I = \int_0^{\pi} \int_{-\pi}^{\pi} (\sin(x)\cos(y) + 1) dxdy$$

は、次のコマンドで計算できます。

```
> double func(double x,double y) { return sin(x)*cos(y)+1;}
> integral2(func,0, 3.14, -3.14, 3.14)
19.7392
```

下限値 $y_1(x)$ および上限値 $y_2(x)$ が変数 x を持つ関数の場合、次のプロトタイプを持つ関数 **integration2()** を使用できます。

```
double integration2(double (*func)(double x, double y), double x1,
        double x2, double (*y1)(double x), double (*y2)(double x));
```

関数 **integral2()** と異なり、関数 **integration2()** では引数 `y1` と `y2` が関数へのポインタになります。たとえば、次のような $r = 2$ の半径を持つ円形の r^2 の積分

$$I = \int_{-r}^{r} \int_{-\sqrt{r^2-x^2}}^{\sqrt{r^2-x^2}} (x^2 + y^2) dxdy$$

は、プログラム 24.7を使って計算できます。

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

double func(double x,double y) {
   return x*x+y*y;
}
double y1(double x) {
   return -sqrt(4-x*x);
}
double y2(double x) {
   return sqrt(4-x*x);
}

int main() {
  double x1=-2, x2=2;
  double s;
  s=integration2(func,x1,x2,y1,y2);
  printf("integration2() = %.3f\n", s);
}
```

プログラム 24.7: 関数 **integration2()** を使用したプログラム

プログラム 24.7の出力結果は、`integration2() = 25.156` となります。

24.9.3 3次元積分

同様に、次の 3 次元積分式

$$I = \int_{x_1}^{x_2} dx \int_{y_1}^{y_2} dy \int_{z_1}^{z_2} dz f(x,y,z),$$

の値は、次のプロトタイプを持つ関数 **integral3()** を使用して計算できます。

```
double integral3(double (*func)(double x, double y, double y),
   double x1, double x2, double y1, double y2, double z1, double z2);
```

引数 *func*(関数へのポインタ) は、積分される関数です。引数 *x1* と *x2* は x の端点、*y1* と *y2* は y の端点、*z1* と *z2* は z の端点です。たとえば、次の積分式

$$I = \int_0^{\pi} \int_{-\pi}^{\pi} \int_0^{\pi} (\sin(x)\cos(y)\sin(z) + 1) dxdydz$$

の値は、次のコマンドで計算できます。

```
> double func(double x,double y, double z) \
      { return sin(x)*cos(y)*sin(z)+1;}
> integral3(func,0, 3.14, -3.14, 3.14, 0, 3.14)
62.0126
```

下限値 $y_1(x)$ および上限値 $y_2(x)$ が変数 x を持つ関数の場合、または下限値 $z_1(x,y)$ および上限値 $z_2(x,y)$ が x と y の変数を持つ関数の場合、次のプロトタイプを持つ関数 **integration3()** を使用できます。

```
double integration3(double (*func)(double x, double y), double x1,
   double x2, double (*y1)(double x), double (*y2)(double x));
   double (*z1)(double x, double y), double (*z2)(double x, double z));
```

関数 **integral3()** と異なり、関数 **integration3()** では引数 `y1`、`y2`、`z1`、および `z2` が関数へのポインタとなります。たとえば、次のような $r = 2$ の半径を持つ球体の r^2 の積分

$$I = \int_{-r}^{r} \int_{-\sqrt{r^2-x^2}}^{\sqrt{r^2-x^2}} \int_{-\sqrt{r^2-x^2-y^2}}^{\sqrt{r^2-x^2-y^2}} (x^2 + y^2 + z^2) dxdydz$$

は、プログラム 24.8で計算できます。

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

double func(double x,double y,double z) {
   return x*x+y*y+z*z;
}
double y1(double x) {
   return -sqrt(4-x*x);
}
double y2(double x) {
   return sqrt(4-x*x);
}
double z1(double x,double y) {
   return -sqrt(4-x*x-y*y);
}
double z2(double x,double y) {
   return sqrt(4-x*x-y*y);
}

int main() {
   double x1=-2, x2 =2, s;
   s=integration3(func,x1,x2, y1,y2, z1,z2);
   printf("integration3() = %.3f\n", s);
}
```

プログラム 24.8: 関数 **integration3()** を使用したプログラム

プログラム 24.8の出力結果は、`integration3() = 80.487` となります。

24.10 行列関数

このセクションでは、n × n の正方行列のみに適用される初歩の関数を説明します。

24.10.1 行列の特性

関数 **charpolycoef()** を使用する行列の固有多項式の係数の計算方法は、セクション 24.5.6で説明されています。このセクションでは、それ以外の行列の機能を利用する関数について説明します。多次元配列の様々な特性に関して、セクション 24.2で説明したいくつかの関数も、n × n 正方行列に適用できます。

行列式

関数 **determinant()** のプロトタイプは

```
double determinant(array double complex a[&][&]);
```

であり、行列 *a* の行列式を返します。行列 *a* が正方行列でない場合は、NaN が返されます。複素行列の場合は、プロトタイプ

```
double complex cdeterminant(array double complex a[&][&]);
```

を持つ関数 **cdeterminant()** を使用して、行列式を計算します。次に例を示します。

```
> array double a[2][2] = {2, 4, 3, 7}
> determinant(a)
2.0000
> cdeterminant(a)
complex(2.0000,-0.0000)
```

条件数

行列の条件数は、連立一次方程式の解のデータ誤差に対する感度を測定します。条件数によって、行列の逆行の結果の精度および連立一次方程式の解の数値結果の精度が示されます。条件数が 1 に近づく (小さい) ほど、行列の条件が良いことを示します。悪条件の行列では、上限は無限に大きくなります。関数 **condnum()** のプロトタイプは

```
double condnum(array double complex a[&][&]);
```

であり、行列 a の条件数を返します。関数 **rcondnum()** のプロトタイプは

```
double rcondnum(array double complex a[&][&]);
```

であり、1 ノルムで行列の条件の逆数の見積りを計算します。**condnum()** と比べ、関数 **rcondnum()** は行列の条件を見積もるのに効率的な方法ですが、信頼性は高くありません。次に例を示します。

```
> array double a[2][2] = {2, 4, 3, 7}
> array double b[2][2] = {2, 4, 2.001, 4.001}
> condnum(a)
38.9743
> condnum(b)
20006.0010
> rcondnum(a)
0.0182
> 1/condnum(a)
0.0257
> rcondnum(b)
0.0000
> 1/condnum(b)
0.0001
```

トレース

トレースは行列の対角要素の合計として定義されます。関数 **trace()** のプロトタイプは

```
double trace(array double a[&][&]);
```

であり、行列 a のトレースを返します。複素行列の場合は、次のプロトタイプを持つ関数 **ctrace()** を使用してトレースを計算します。

```
double complex ctrace(array double complex a[&][&]);
```

次に例を示します。

```
> array double a[2][2] = {2, 4, 3, 7}
> trace(a)
9.0000
> ctrace(a)
complex(9.0000,0.0000)
> array double b[2][3] = {1, 1, 1, 1, 1, 1}
> trace(b)
2.0000
```

対角

関数 **diagonal()** のプロトタイプは

```
double diagonal(array double a[&][&], ... /* [int k] */)[:];
```

であり、行列 a の k 番目の要素で形成された列ベクトルを求めます。オプション引数 k が指定されていない場合、関数は行列の対角を返します。複素行列の場合は、次のプロトタイプを持つ関数 **ctrace()** を使用して対角を計算します。

```
double complex cdiagonal(array double complex a[&][&],
                         ... /* [int k] */)[:];
```

次に例を示します。

```
> array double a[4][3] = {1, 2, 3, \
                          4, 5, 6, \
                          7, 8, 9, \
                          4, 4, 4}
> diagonal(a)
1.0000 5.0000 9.0000
> diagonal(a, -1)
4.0000 8.0000 4.0000
> cdiagonal(a, 1)
complex(2.0000,0.0000) complex(6.0000,0.0000)
```

ランク

ランクは、行列の連続する独立の行または列の数として定義されます。関数 **rank()** のプロトタイプは

```
int rank(array double complex a[&][&]);
```

であり、行列 a のランクを返します。関数内で使用されるアルゴリズムは、特異値分解に基づきます。ランクは、$tol = \max(m, n) \times \max(S) \times$ `DBL_EPSILON`. の許容値を持つ、ゼロ以外の特異値の数です。次に例を示します。

```
> array double a[2][2] = {2, 4, 3, 7}
> array double b[2][3] = {1, 2, 3, 2, 4, 6}
> rank(a)
2
> rank(b)
1
```

24.10.2 行列の操作

行列の転置に使用する汎用関数 **transpose()** に加え、関数 **fliplr()**、**flipud()**、**rot90()** を使用して行列を操作できます。

行列の左右の反転

関数 **fliplr()** のプロトタイプは

```
int fliplr(array double complex y[&][&], array double complex x[&][&]);
```

であり、行列 x を左右に反転し、行は変更せず、列のみを入れ替えます。x と同じデータ型とサイズを持つ行列 y には、入力された行列 x の反転した結果が含まれます。次に例を示します。

```
> array double y[2][4], x[2][4]={1, 2, 3, 4, \
                                 5, 6, 7, 8}
> fliplr(y, x)
> y
4.0000 3.0000 2.0000 1.0000
8.0000 7.0000 6.0000 5.0000
```

行列の上下の反転

関数 **fliplr()** と同様に、次のプロトタイプ

```
int flipud(array double complex y[&][&], array double complex x[&][&]);
```

を持つ関数 **flipud**() は、行列 X を上下に反転し、列は変更せず、行のみを入れ替えます。X と同じデータ型とサイズを持つ行列 y には、入力された行列 x の反転した結果が含まれます。次に例を示します。

```
> array double y[4][2], x[4][2]={1, 2, \
                                 3, 4, \
                                 5, 6, \
                                 7, 8}
> flipud(y, x)
> y
7.0000 8.0000
5.0000 6.0000
3.0000 4.0000
1.0000 2.0000
```

行列の回転

関数 **rot90**() のプロトタイプは

```
int rot90(array double complex y[&][&], array double complex x[&][&],
          ... /* [int  k] */);
```

であり、行列 x を $k*90$ 度回転させます。k が正の値の場合、行列は反時計方向に回転します。k が負の値の場合、行列は時計方向に回転します。配列の引数 y は、行列 x と同じデータ型とサイズの 2 次元行列です。行列 y には、入力された行列 x の回転が含まれます。次に例を示します。

```
> array double y[4][2], x[2][4]={1, 2, 3, 4, \
                                 5, 6, 7, 8}
> rot90(y, x)
> y
4.0000 8.0000
3.0000 7.0000
2.0000 6.0000
1.0000 5.0000
> rot90(x, x, 2)
> x
8.0000 7.0000 6.0000 5.0000
4.0000 3.0000 2.0000 1.0000
```

24.10.3 特殊な行列

このセクションでは、Ch の特殊な行列を使用したプログラミングを説明します。

単位行列

関数 **identitymatrix()** のプロトタイプは

```
array double identitymatrix(int n)[:][:];
```

であり、$n \times n$ の単位行列を返します。次に例を示します。

```
> identitymatrix(3)
1.0000 0.0000 0.0000
0.0000 1.0000 0.0000
0.0000 0.0000 1.0000
```

対角行列

関数 **diagonalmatrix()** のプロトタイプは

```
array double diagonalmatrix(array double v[&], ... /*[int k]*/)[:][:];
```

であり、k 番目の対角要素を v とする、$n + abs(k)$ の正方行列を返します。$k = 0$ の場合は主対角線を、$k > 0$ の場合は主対角線以上を、$k < 0$ の場合は主対角線以下をそれぞれ表します。既定では、k は 0 です。関数 **diagonalmatrix()** は、主対角行列を作成します。複素数型の対角行列の場合は、次のプロトタイプを持つ関数 **cdiagonalmatrix()** を使用する必要があります。

```
array double cdiagonalmatrix(array double complex v[&], ...
                             /* [int k] */)[:][:];
```

次に例を示します。

```
> array double v[2] = {1, 2}
> diagonalmatrix(v)
1.0000 0.0000
0.0000 2.0000
> diagonalmatrix(v, 1)
0.0000 1.0000 0.0000
0.0000 0.0000 2.0000
0.0000 0.0000 0.0000
> diagonalmatrix(v, -1)
0.0000 0.0000 0.0000
1.0000 0.0000 0.0000
0.0000 2.0000 0.0000
```

三角行列

関数 **triangularmatrix**() のプロトタイプは

```
array double triangularmatrix(string_t pos, array double a[&][&],
                                  ... /* [int k] */)[:][:];
```

であり、行列 *a* の三角行列を返します。入力された行列 *a* と同じ次元の行列が返されます。引数 *pos* が`"upper"`の場合、関数は行列 *a* の上三角部 (行列の *k* 番目の対角線の上側 (対角線も含む)) を返します。オプション引数 *k* を指定すると、三角行列を行列の上側 k 番目の対角線へオフセットします。引数 *pos* が`"lower"`の場合、関数は行列 *a* の下三角部 (行列の *k* 番目の対角線の下側 (対角線も含む)) を返します。オプション引数 *k* を指定すると、三角行列を行列の下側 *k* 番目の対角線へオフセットします。既定では、*k* は 0 です。関数 **diagonalmatrix**() は、主対角行列を作成します。複素数型の三角行列の場合は、次のプロトタイプを持つ関数 **ctriangularmatrix**() を使用する必要があります。

```
array double complex ctriangularmatrix(string_t pos,
        array double complex a[&][&], ... /* [int k] */)[:][:];
```

次に例を示します。

```
> array double a[4][3] = {1,2,3, \
                          4,5,6, \
                          7,8,9, \
                          6,3,5}
> triangularmatrix("upper", a)
1.0000 2.0000 3.0000
0.0000 5.0000 6.0000
0.0000 0.0000 9.0000
0.0000 0.0000 0.0000
> triangularmatrix("upper", a, 1)
0.0000 2.0000 3.0000
0.0000 0.0000 6.0000
0.0000 0.0000 0.0000
0.0000 0.0000 0.0000
> triangularmatrix("upper", a, -1)
1.0000 2.0000 3.0000
4.0000 5.0000 6.0000
0.0000 8.0000 9.0000
0.0000 0.0000 5.0000
> triangularmatrix("lower", a)
1.0000 0.0000 0.0000
4.0000 5.0000 0.0000
7.0000 8.0000 9.0000
6.0000 3.0000 5.0000
```

```
> triangularmatrix("lower", a, 1)
1.0000 2.0000 0.0000
4.0000 5.0000 6.0000
7.0000 8.0000 9.0000
6.0000 3.0000 5.0000
> triangularmatrix("lower", a, -1)
0.0000 0.0000 0.0000
4.0000 0.0000 0.0000
7.0000 8.0000 0.0000
6.0000 3.0000 5.0000
```

随伴行列

関数 **companionmatrix()** のプロトタイプは

```
array double companionmatrix(array double v[&])[:][:];
```

であり、多項式の係数の配列 v から随伴行列を返します。サイズ n の配列 v の場合、随伴行列の先頭の行は-**v**[1:n]/v[0] になります。随伴行列の固有値は多項式を左辺とする代数方程式の根です。複素数型の三角行列の場合は、次のプロトタイプを持つ関数 **ccompanionmatrix()** を使用する必要があります。

```
array double complex ccompanionmatrix(array double complex v[&])[:][:];
```

たとえば、多項式 $2x^3 + 3x^2 + 4x + 5$ の随伴行列は、次のコマンドで得られます。

```
> #define N 4
> array double v[N] = {2,3,4,5}
> array double a[N-1][N-1]
> a = companionmatrix(v)
-1.5000 -2.0000 -2.5000
1.0000 0.0000 0.0000
0.0000 1.0000 0.0000
```

Householder 行列

ベクトル **x** に対して、Householder 行列 **H** は次のように定義されます。

$$\mathbf{H} = \mathbf{I} - \beta \mathbf{v}\mathbf{v}^{\mathbf{T}}$$

ここで、**I** は単位行列で、ベクトル **v** のサイズはベクトル **x** と同じです。Householder 行列 **H** は、次の方程式を満たします。

$$\mathbf{Hx} = -\mathrm{sign}(\mathbf{x}[0]) * \mathrm{norm}(\mathbf{x}) * \mathbf{E}$$

ここで、ベクトル $\mathbf{E} = [1, 0, 0, \cdots, 0]$ は、ベクトル $\mathbf{x}$ と同じサイズです。$\mathbf{x}$ が複素数ベクトルの場合、Householder 行列 $\mathbf{H}$ は、次のように定義されます。

$$\mathbf{H} = \mathbf{I} - \beta \mathbf{v}\mathbf{v}^{\mathbf{H}}$$

また、($\mathbf{x}[0]$) は次のように定義されます。

$$\mathrm{sign}(\mathbf{x}[0]) = \frac{\mathbf{x}[0]}{\mathrm{abs}(\mathbf{x}[0])}$$

関数 **householdermatrix**() のプロトタイプは

```
int householdermatrix(array double complex x[&],
                      array double complex v[&], ... /* [double *beta] */);
```

であり、入力引数としてベクトル *x* を渡すと、Householder 行列のベクトル *v* とオプションの出力値 *beta* が得られます。たとえば、実数型ベクトルの Householdermatrix は、次のコマンドで計算できます。

```
> array double x[5] = {-0.3, 54, 25.3, 25.46, 83.47}
> array double e[5]={1,0,0,0,0}
> array double v[5], h[5][5]
> double beta
> householdermatrix(x,v,&beta)
> h = identitymatrix(5) - beta*v*transpose(v)
> v
-105.9959 54.0000 25.3000 25.4600 83.4700
> beta
0.0001
> h*x+sign(x[0])*norm(x,"2")*e
0.0000 0.0000 0.0000 -0.0000 -0.0000
```

複素数型ベクトルの Householder 行列は、以下のコマンドで計算できます。

```
> array double complex x[3] = {complex(-0.3, 0.5), 54, 25.3}
> array double e[3]={1,0,0}
> array double complex v[3], h[3][3]
> double beta
> householdermatrix(x,v,&beta)
> h = identitymatrix(3) - beta*v*conj(transpose(v))
> printf("%.3f", v)
complex(-30.982,51.637) complex(54.000,0.000) complex(25.300,0.000)
> beta
0.0003
> v = h*x+x[0]/abs(x[0])*norm(x,"2")*e
> printf("%.3f", v)
complex(-30.682,51.137) complex(0.000,0.000) complex(0.000,0.000)
```

特殊行列

関数 **specialmatrix**() のプロトタイプは

```
array double specialmatrix(string_t name, ...
             /* [type1 arg1, type2 arg2, ...] */)[:][:];
```

であり、特殊行列を返します。引数 *name* には、表 24.4にリストした特殊行列のいずれかを指定できます。

表 24.4: 特殊行列

Cauchy	ChebyshevVandermonde	Chow	Circul	Celement
Dramadah	DenavitHartenberg	DenavitHartenberg2	Fiedler	Frank
Gear	Hadamard	Hankel	Hilbert	InverseHilbert
Magic	Pascal	Rosser	Toeplitz	Vandermonde
Wilkinson				

特殊行列では、オプション引数 *arg1* および *arg2* の代入が必要になる場合があります。特殊行列の種類によって、引数の数とデータ型が異なります。

たとえば、Hilbert 行列の **H** は、$\mathbf{H}[i][j] = 1/(i+j+1)$ の要素を持ちます。これは、悪条件な行列の例です。Hilbert 行列は、`specialmatrix("Hilbert", n)` の関数呼び出しで生成されます。ここで引数 *n* には行列の次数を指定します。すなわち、Hilbert 行列のサイズは $n \times n$ となります。4 × 4 の Hilbert 行列は、次のコマンドで作成できます。

```
> specialmatrix("Hilbert", 4)
1.0000 0.5000 0.3333 0.2500
0.5000 0.3333 0.2500 0.2000
0.3333 0.2500 0.2000 0.1667
0.2500 0.2000 0.1667 0.1429
```

各特殊行列の詳細については、『*Ch* **言語環境リファレンスガイド**』の数値解析の章を参照してください。

24.10.4 行列解析

配列の各要素に関数を適用する関数 **fevalarray**() や関数 **cfevalarray**() とは異なり、このセクションで説明する Ch 関数を使用して、行列型の変数を含む数学関数を評価することができます。関数 **funm**() のプロトタイプは

```
int funm(array double y[&][&], double (*func)(double),
         array double x[&][&]);
```

であり、引数 *func* で指定された、行列バージョンの関数を評価します。この関数では、double 型の正方行列 *x* を入力し、次のようにプロトタイプ化された関数を指定する必要があります。

```
double func(double);
```

行列 *x* と同じサイズを持つ正方行列 *y* の出力は、必要に応じて、実数型または複素数型にできます。関数 **funm()** は、成功すると 0 を返し、失敗すると-1 を返します。複素数型の行列の場合は、次のプロトタイプを持つ関数 **cfunm()** を使用する必要があります。

```
int cfunm(array double complex y[&][&],
          double complex (*func)(double complex),
          array double complex x[&][&]);
```

たとえば、次のような多項式

$$\mathbf{Y} = \mathbf{X}^3 + 2\mathbf{X}^2 + 3\mathbf{X} + 4$$

の次の行列

$$\mathbf{X} = \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix}$$

に対する **Y** の値は、次のコマンドで計算できます。

```
> array double p[4] = {1,2,3,4}
> array double x[2][2] = {5,6,7,8}, y[2][2]
> double func(double x) { return polyeval(p, x);}
> funm(y, func, x)
> y
1034.0000 1200.0000
1400.0000 1634.0000
```

Ch では、行列を解析するための関数 **polyevalm()**、**sqrtm()**、**expm()**、および **logm()** が実装されており、それぞれ、様々な変数を持つ多項式関数、平方根関数、指数関数、および自然対数関数を扱うことができます。これらの関数のプロトタイプは

```
int polyevalm(array double complex y[&][&],
              array double complex p[&],
              array double complex x[&][&]);
int sqrtm(array double complex y[&][&], array double complex x[&][&]);
int expm(array double complex y[&][&], array double complex x[&][&]);
int logm(array double complex y[&][&], array double complex x[&][&]);
```

であり、たとえば、前述の行列の多項式は、関数 **polyevalm()** を使って計算できます。また、**expm()** 対 **logm()** および **sqrtm()** 対 X^2 の相補的な機能の適用結果は、次のコマンドで得られます。

```
> array double p[4] = {1,2,3,4}
> array double x[2][2] = {5,6,7,8}, y[2][2]
> polyevalm(y, p, x)
> y
1034.0000 1200.0000
1400.0000 1634.0000
> expm(y, x)
> y
199464.8244 232291.9543
271007.2800 315610.8016
> logm(y, y)
> y
5.0000 6.0000
7.0000 8.0000
> array double complex z[2][2]
> sqrtm(z, x)
> z
complex(1.4044,0.2389) complex(1.6355,-0.1759)
complex(1.9081,-0.2052) complex(2.2222,0.1510)
> array double p2[3] = {1, 0, 0}
> polyevalm(y, p2, z)
> y
5.0000 6.0000
7.0000 8.0000
```

24.11 行列分解

24.11.1 LU 分解

ludecomp()

関数 **ludecomp()** では、部分的なピボット操作で行を交換することで、正方行列 **A** の LU 分解を計算できます。分解は次の形式を持ちます。

$$\mathbf{A} = \mathbf{PLU},$$

ここで、**P** は置換行列を表し、**L** は単位対角要素を持つ下三角行列、**U** は上三角行列をそれぞれ表します。関数 **ludecomp()** のプロトタイプは

```
int ludecomp(array double complex a[&][&],array double complex l[&][&],
        array double complex u[&][&], ... /*[array int p[&][&]] */);
```

であり、上記の LU 分解公式に従って、一般的な n x n 正方行列を分解します。次に例を示します。

```
> array double l[3][3], u[3][3], a[3][3] = {2, 1, -2, \
                                            4, -1, 2, \
                                            2, -1, 2}
> array int p[3][3]
> ludecomp(a, l, u)
> l
0.5000 1.0000 0.0000
1.0000 0.0000 0.0000
0.5000 -0.3333 1.0000
> u
4.0000 -1.0000 2.0000
0.0000 1.5000 -3.0000
0.0000 0.0000 0.0000
> l*u
2.0000 1.0000 -2.0000
4.0000 -1.0000 2.0000
2.0000 -1.0000 2.0000
> ludecomp(a, l, u, p)
> l
1.0000 0.0000 0.0000
0.5000 1.0000 0.0000
0.5000 -0.3333 1.0000
> u
4.0000 -1.0000 2.0000
0.0000 1.5000 -3.0000
0.0000 0.0000 0.0000
> p
0 1 0
1 0 0
0 0 1
> p*l*u
2.0000 1.0000 -2.0000
4.0000 -1.0000 2.0000
2.0000 -1.0000 2.0000
```

24.11.2 特異値分解

特異値分解は次のように定義されます。

$$\mathbf{A} = \mathbf{U}\mathbf{S}\mathbf{V}^{\mathbf{T}}$$

ここで、S は、サイズ min(m,n) の対角要素を除き、各要素にゼロを持つ m $\times$ n の行列を表します。U

と V は、それぞれ、m × m の直交行列と n × n の直交行列を示します。S の対角要素は行列 **A** の特異値です。実数を取り、負の値は取りません。降順に並べられています。**U** と **V** の先頭の min(m,n) 列は、行列 A の左と右の特異ベクトルとなります。関数 **svd**() のプロトタイプは

```
int svd(array double complex a[&][&], array double s[&],
        array double complex u[&][&], array double complex vt[&][&]);
```

であり、数型または複素数型の n × m 行列の特異値、および左右の特異ベクトルを計算します。たとえば、n × n 行列の特異値分解は、次のコマンドで計算できます。

```
> array double a[2][2] = {1, 2, \
                          3, 4}
> array double s[2], u[2][2], v[2][2]
> svd(a, s, u, v)
> s
5.4650 0.3660
> u
-0.4046 -0.9145
-0.9145 0.4046
> v
-0.5760 0.8174
-0.8174 -0.5760
> u*diagonalmatrix(s)*transpose(v)
1.0000 2.0000
3.0000 4.0000
```

n x m 行列の特異値分解は、次のコマンドで実行できます。

```
> array double a[2][3] = {7, 8, 1, \
                          3, 6, 4}
> int m=2, n=3, mn=min(m,n), i
> array double s[mn], sm[m][n], u[m][m], v[n][n]
> svd(a, s, u, v)
> for(i=0; i<mn; i++) sm[i][i] = s[i]
> s
12.8515 3.1367
> u
-0.8189 -0.5739
-0.5739 0.8189
> v
-0.5800 -0.4976 0.6450
-0.7777 0.1027 -0.6202
-0.2424 0.8613 0.4465
> u*sm*transpose(v)
```

```
7.0000 8.0000 1.0000
3.0000 6.0000 4.0000
```

24.11.3 Cholesky 分解

Cholesky 分解は、対称正定値行列を 2 つの行列に分解します。実数型の対称正定値行列 **A** では、Cholesky 分解は

$$\mathbf{A} = \mathbf{L}^{\mathbf{T}}\mathbf{L}$$

を満たす上三角行列 **L** または

$$\mathbf{A} = \mathbf{L}\mathbf{L}^{\mathbf{T}}$$

を満たす下三角行列 **L** を生成します。ここで、$\mathbf{L}^T$ は、行列 **L** の転置です。複素数型の対称正定値行列 **A** の場合は、$\mathbf{L}^H$ の代わりに、行列 **L** のエルミート $\mathbf{L}^H$ を使用します。

関数 **choldecomp()** のプロトタイプは

```
int choldecomp(array double complex a[&][&],
               array double complex l[&][&], ... /* [char mode] */);
```

であり、Cholesky 分解を実行します。配列引数 **l** には、対称正定値行列 **a** の上三角と下三角が含まれます。**string_t** *mode* で、上三角行列または下三角行列の計算を指定します。下三角の因数分解には'L' または'l' の文字を指定します。指定しない場合は、上三角行列が計算されます。

既定で、上三角の行列が計算されます。関数 **choldecomp()** は、成功すると 0 を返し、失敗すると負の数を返します。正の値 i は、順位 i の首座小行列が正の定符号ではなく、分解が完了できないことを示します。

たとえば、実数型の対称正定値行列を上三角行列へ Cholesky 分解する計算は、次のコマンドで実行できます。

```
> array double l[2][2], a[2][2] = {1, 1, \
                                   1, 2}
> choldecomp(a, l);
> l
1.0000 1.0000
0.0000 1.0000
> transpose(l)*l
1.0000 1.0000
1.0000 2.0000
```

複素数型の対称正定値行列を下三角行列へ Cholesky 分解する計算は、次のコマンドで実行できます。

```
> array double complex l[2][2],
          a[2][2] = {complex(2,0), complex(0,-1), \
                     complex(0,1), complex(2,0)}
```

```
> choldecomp(a, l, 'L');
> l
complex(1.4142,0.0000) complex(0.0000,0.0000)
complex(0.0000,0.7071) complex(1.2247,0.0000)
> l*conj(transpose(l))
complex(2.0000,0.0000) complex(0.0000,-1.0000)
complex(0.0000,1.0000) complex(2.0000,0.0000)
```

24.11.4 QR 分解

QR 分解は、次のように行列 **A** を 2 つの行列に分解します。

$$\mathbf{A} = \mathbf{QR},$$

ここで、**Q** は実数型の直交行列または複素数型のユニタリ行列を表し、**R** は上三角行列を表します。行列 **A** のサイズが $m \times n$ であるとします。$m \leq n$ では、行列 **Q** および行列 **R** は 1 種類です。行列 **Q** のサイズは $m \times m$ で、行列 R のサイズは $m \times n$ です。$m \leq n$ では、行列 **Q** と行列 R は 2 つの種類になります。1 つはフルサイズで、行列 **Q** は $m \times m$、行列 R は $m \times n$ となります。もう 1 つは無駄のないサイズで、行列 **Q** は $m \times n$、行列 **R** は $n \times n$ となります。関数 **qrdecomp()** のプロトタイプは

```
int qrdecomp(array double complex a[&][&],
    array double complex q[&][&], array double complex r[&][&]);
```

であり、QR 分解を実行します。行列 a からユニタリ行列 q および上三角行列 q が得られます。サイズ $m \times n$ の配列 a では、$m > n$ の場合、関数 **qrdecomp()** によって引数 q と r の配列サイズがチェックされ、対応する出力の型が自動的に選択されます。たとえば、実数行列の QR 分解は、次のコマンドで計算できます。

```
> array double q1[3][3], r1[3][2], q2[3][2], r2[2][2],
> array double a[3][2] = {1, 5, \
                          -7, 4, \
                          3, 2}
> qrdecomp(a, q1, r1)
> q1
-0.1302 -0.8351 -0.5345
0.9113 -0.3132 0.2673
-0.3906 -0.4523 0.8018
> r1
-7.6811 2.2132
0.0000 -6.3326
0.0000 0.0000
> transpose(q1)*q1
1.0000 -0.0000 -0.0000
```

```
-0.0000 1.0000 -0.0000
-0.0000 -0.0000 1.0000
> q1*r1
1.0000 5.0000
-7.0000 4.0000
3.0000 2.0000
> qrdecomp(a, q2, r2)
> q2
-0.1302 -0.8351
0.9113 -0.3132
-0.3906 -0.4523
> r2
-7.6811 2.2132
0.0000 -6.3326
> transpose(q2)*q2
1.0000 -0.0000
-0.0000 1.0000
> q2*r2
1.0000 5.0000
-7.0000 4.0000
3.0000 2.0000
```

複素数型の行列の QR 分解は、次のコマンドで計算できます。

```
> array double complex q[2][2], r[2][2]
> array double complex a[2][2] = {complex(1,2), 5, \
                                  3, 3}
> qrdecomp(a, q, r)
> q
complex(-0.2673,-0.5345) complex(0.7171,-0.3587)
complex(-0.8018,0.0000) complex(-0.0000,0.5976)
> r
complex(-3.7417,0.0000) complex(-3.7417,2.6726)
complex(0.0000,0.0000) complex(3.5857,0.0000)
> conj(transpose(q))*q
complex(1.0000,0.0000) complex(-0.0000,0.0000)
complex(-0.0000,-0.0000) complex(1.0000,0.0000)
> q*r
complex(1.0000,2.0000) complex(5.0000,0.0000)
complex(3.0000,-0.0000) complex(3.0000,0.0000)
```

24.11.5 Hessenberg 分解

実数型の正方行列 **A** の Hessenberg 行列 **H** は、次のように定義されます。

$$\mathbf{H} = \mathbf{P}^T\mathbf{A}\mathbf{P}$$

ここで、行列 **P** は、$\mathbf{P}^T\mathbf{P}=\mathbf{I}$ の値を持つ直交行列です。複素数型の行列 **A** の場合、Hessenberg 行列 **H** は次のように定義されます。

$$\mathbf{H} = \mathbf{P}^\mathbf{H}\mathbf{A}\mathbf{P}$$

ここで、行列 **P** は $\mathbf{P}^H\mathbf{P}=\mathbf{I}$ の値を持つユニタリ行列です。先頭の副対角線の下に位置する Hessenberg 行列の各要素はゼロです。行列が対称行列またはエルミート行列の場合は、三重対角の形式になります。この行列は、元の行列と同じ固有値を持っていますが、元の行列より少ない計算で求めることができます。関数 **hessdecomp()** のプロトタイプは

```
int hessdecomp(array double complex a[&][&],
               array double complex h[&][&], ...
               /* [array double complex p[&][&]] */);
```

であり、行列 *a* を、Hessenberg 行列 *h* と直交行列またはユニタリ行列 *p* に分解します。正方行列 *a* は、サポートされている任意の算術データ型を使用できます。行列 h の出力は、*a* の入力値と同じ次元とデータ型になります。*a* の入力値が実数型の場合、オプション出力 *p* は double 型のみになります。*a* の代入値が複素数型の場合、*p* は **complex** 型または **double complex** 型になります。たとえば、Hessenberg 行列は、次のコマンドで計算できます。

```
> array double a[3][3] = {0.8, 0.2, 0.1,\
                          0.1, 0.7, 0.3,\
                          0.1, 0.1, 0.6}
> array double h[3][3], p[3][3]
> hessdecomp(a, h, p)
> h
1.0000 0.0000 0.0000
0.0000 -0.7071 -0.7071
0.0000 -0.7071 0.7071
> p
0.8000 -0.2121 -0.0707
-0.1414 0.8500 -0.0500
0.0000 0.1500 0.4500
> transpose(p)*a*p
1.0000 0.0000 0.0000
0.0000 -0.7071 -0.7071
0.0000 -0.7071 0.7071
```

24.11.6 Schur 分解

実数型の正方行列 **A** の Schur 行列 **T** は、次のように定義されます。

$$\mathbf{A} = \mathbf{QTQ}^T$$

ここで、行列 **Q** は、$\mathbf{Q}^T\mathbf{Q}=\mathbf{I}$ の値を持つ直交行列です。複素数型の行列 **A** の場合、Schur 行列 T は次のように定義されます。

$$\mathbf{A} = \mathbf{QTQ}^H$$

ここで、行列 **Q** は $\mathbf{Q}^H\mathbf{Q}=\mathbf{I}$ の値を持つユニタリ行列です。関数 **schurdecomp()** のプロトタイプは

```
int schurdecomp(array double complex a[&][&],
    array double complex q[&][&], array double complex t[&][&])
```

であり、行列 a を、Schur 行列 t と直交行列またはユニタリ行列 q に分解します。正方行列 a は、サポートされている任意の算術データ型を使用できます。行列 t と行列 q の出力は、a の入力値と同じ次元とデータ型になります。たとえば、Schur 行列は、次のコマンドで計算できます。

```
> array double t[2][2], q[2][2], a[2][2] = {8, -3, \
                                           -5, 9}
> schurdecomp(a, q, t)
> t
4.5949 2.0000
0.0000 12.4051
> q
0.6611 -0.7503
0.7503 0.6611
> transpose(q)*q
1.0000 0.0000
0.0000 1.0000
> q*t* transpose(q)
8.0000 -3.0000
-5.0000 9.0000
```

24.12 線形方程式

24.12.1 連立一次方程式

次のような連立一次方程式

$$\mathbf{Ax} = \mathbf{b}$$

は、Ch では、さまざまな関数で解を求めることができます。行列 **A** が double 型の n × n の正方行列の場合は、関数 **linsolve()** でその解を求めることができます。関数 **linsolve()** は、LU 分解を使用して

部分的なピボット操作で行を交換し、行列 **A** を $\mathbf{A} = \mathbf{PLU}$ と分解します。ここで、P は置換行列を表し、L は単位下三角行列、U は上三角行列を表します。**A** の分解後の式は、**x** の連立方程式の解を求めるのに使用されます。関数 **linsolve()** のプロトタイプは

```
int linsolve(array double x[:], array double a[:][:], array double b[:])
```

であり、$\mathbf{Ax} = \mathbf{b}$ の連立一次方程式の **x**、**A**、および **b** のそれぞれに対応する、x、a、および b の 3 つの引数を受け取ります。これらの引数は、double 型の配列にする必要があります。関数 **linsolve()** は、方程式の解に成功すると 0 を返します。失敗すると-1 を返します。複素数型の連立一次方程式の解を求める場合は、関数 **clinsolve()** を使用します。次に例を示します。

```
> array double a[3][3] = {3, 0, 6,\
                          0, 2, 1,\
                          1, 0, 1}
> array double x[3], b[3] = {2, 13, 25}
> linsolve(x, a, b)
> x
49.3333 18.6667 -24.3333
> a*x
2.0000 13.0000 25.0000
```

24.12.2 過剰決定または過少決定の連立一次方程式

次のような連立一次方程式

$$\mathbf{Ax} = \mathbf{b},$$

で、m x n の次元を持つ A が正方行列でない場合、$(\mathbf{A} * \mathbf{x} - \mathbf{b})^T(\mathbf{A} * \mathbf{x} - \mathbf{b})$ の 2 乗誤差の最小値を求める直線の最小 2 乗法で、解を求めることができます。関数 **llsqsolve()** のプロトタイプは

```
int llsqsolve(array double complex x[&],
    array double complex a[&][&], array double complex b[&]);
```

であり、$\mathbf{Ax} = \mathbf{b}$ の連立一次方程式の **x**、**A**、および **b** のそれぞれに対応する、x、a、および b の 3 つの引数を受け取ります。これらの引数は、複素数の配列にもできます。x の要素の数は、行列 a の列数と同じにする必要があります。b の要素の数は、行列 a の行数と同じにする必要があります。関数 **llsqsolve()** は、方程式の解に成功すると 0 を返します。失敗すると-1 を返します。たとえば、最小 2 乗法による連立一次方程式の解は、次のコマンドで計算できます。

```
> array double a[2][3] = {3, 5, 6,\
                          7, 2, 1}
> array double x[3], b[2] = {1, 2}
> llsqsolve(x, a, b)
> x
```

```
0.2742 0.0440 -0.0070
> a*x
1.0000 2.0000
> array double a2[3][2] = {3, 5, \
                           6, 7, \
                           2, 1}
> array double x2[2], b2[3] = {1, 2, 3}
> llsqsolve(x2, a2, b2)
> x2
1.4278 -0.8299
> a2*x2
0.1340 2.7577 2.0258
```

関数 **llsqnonnegsolve()** のプロトタイプは

```
int llsqnonnegsolve(array double x[&], array double a[&][&],
       array double b[&], ... /* [double tol, array double w[&]] */);
```

であり、最小 2 乗法を使用した連立一次方程式 $\mathbf{Ax} = \mathbf{b}$ の解を求めるのに使用できます。ただし、解ベクトル x が持つ要素は、負にはならないという制約があります。すなわち、$x[i] \geq 0$ for $i = 0, 1, \cdots, n{-}1$ に対して、$\mathbf{Ax} = \mathbf{b}$ となります。関数 **llsqnonnegsolve()** は、$\mathbf{Ax} = \mathbf{b}$ の連立一次方程式の **x**、**A**、および **b** のそれぞれに対応する x、a、および b の 3 つの引数を受け取ります。オプション引数 *tol* には、解の許容値を指定します。ユーザーがこの引数を指定しないか、またはゼロを指定すると、既定で、$tol = 10 * max(m, n) * norm(u, "1") *$ FLT_EPSILON が使用されます。FLT_EPSILON は、ヘッダーファイル **float.h** で定義されます。n 個の要素を持つオプションの配列引数 w には、ベクトル ($x[i] = 0$ の場合は $w[i] < 0$。$x[i] > 0$ の場合は $w[i] \cong 0$) が含まれます。たとえば、最小 2 乗法に基づく連立一次方程式の非負の解は、次のコマンドで計算できます。

```
> array double a[2][3] = {3, 5, 6,\
                          7, 2, 1}
> array double x[3], w[3], b[2] = {1, 2}
> llsqnonnegsolve(x, a, b)
> x
0.2821 0.0000 0.0257
> a*x
1.0000 2.0000
> array double a2[3][2] = {3, 5, \
                           6, 7, \
                           2, 1}
> array double x2[2], b2[3] = {1, 2, 3}
> llsqnonnegsolve(x2, a2, b2)
> x2
0.4286 0.0000
```

```
> a2*x2
1.2857 2.5714 0.8571
```

関数 **llsqcovsolve**() のプロトタイプは

```
int llsqcovsolve(array double x[&], array double a[&][&],
       array double b[&], array double v[&][&], ...
       /* [array double p[&]] */);
```

であり、最小 2 乗法を使用して、平均値ゼロの正規分布した誤差と共分散 *v* 連立一次方程式 $\mathbf{A}\mathbf{x} = \mathbf{b}$ の解を求めることができます。これは過剰決定の線形最小 2 乗問題です。したがって、行 m の数は列 n の数より大きくなければなりません。オプションの出力ベクトル *p* を指定すると、*x* の標準誤差が渡されます。次に例を示します。

```
> array double a[3][2] = {3, 5, \
                          6, 7, \
                          2, 1}
> array double x[2], b[3] = {1, 2, 3}
> array double v=[3][3] = {1, 1, 3, \
                           1, 2, 5, \
                           3, 5, 6}
> llsqcovsolve(x, a, b, v)
> x
0.3402 -0.3521
> a*x
-0.7396 -0.4231 0.3284
```

24.12.3 逆行列と疑似逆行列

正方行列 **A** の逆行列 A^{-1} は、次のように定義されます。

$$\mathbf{A^{-1}A} = \mathbf{AA^{-1}} = \mathbf{I}.$$

逆行列を得るには、行列 **A** は特異的であってはなりません。実数型の行列 **A** の逆行列 A^{-1} は、次のプロトタイプを持つ関数 **inverse**() で計算できます。

```
array double inverse(array double a[:][:], ...
                     /* [int *status] */)[:][:];
```

逆行列の計算は、元の行列の LU 分解に基づきます。オプション引数 *status* に、計算の状態が示されます。計算に成功すると、*status* は 0 を示し、失敗した場合は負の値を示します。複素数型の行列 **A** の逆行列 A^{-1} は、次のプロトタイプを持つ関数 **cinverse**() で計算できます。

```
array double complex cinverse(array double complex a[:][:], ...
                              /* [int *status] */)[:][:];
```

たとえば、逆行列を使用して **x** = A^{-1}**b** の連立一次方程式の解を求めるには、次のコマンドを使用できます。

```
> array double a[3][3] = {3, 0, 6,\
                          0, 2, 1,\
                          1, 0, 1}
> array double ai[3][3], b[3] = {2, 13, 25}
> ai=inverse(a)
-0.3333 -0.0000 2.0000
-0.1667 0.5000 0.5000
0.3333 0.0000 -1.0000
> ai*b
49.3333 18.6667 -24.3333
```

行列 **A** の Moore_Penrose 疑似逆行列 **B** は、次の 4 つの条件を満たさなければなりません。

$$\mathbf{ABA} = \mathbf{A},$$
$$\mathbf{BAB} = \mathbf{B},$$
$$\mathbf{AB} \text{ is Hermitian}$$
$$\mathbf{BA} \text{ is Hermitian}$$

（**AB** と **BA** はエルミート行列）ここで、A には、特異の正方行列または非正方行列であっても構いません。関数 **pinverse()** のプロトタイプは

```
array double pinverse(array double a[:][:])[:][:];
```

であり、実数型の関数 *a* の Moore_Penrose 疑似逆行列を計算します。次に例を示します。

```
> int M = 2, N = 3
> array double a[2][3] = {7, 8, 1, \
                          3, 6, 4}
> array double p[3][2]
> p = pinverse(a)
0.1280 -0.1040
0.0308 0.0615
-0.1422 0.2357
> a*p*a
7.0000 8.0000 1.0000
3.0000 6.0000 4.0000
> p*a*p
0.1280 -0.1040
0.0308 0.0615
-0.1422 0.2357
```

24.12.4 線形空間

関数 **orthonormalbase()** のプロトタイプは

```
int orthonormalbase(array double complex orth[&][&],
                    array double complex a[&][&]);
```

であり、次元 m × n を持つ行列 *a* の正規直交基底を計算します。*orth* の列は正規直交し、**a** の列と同じ空間を持ちます。配列 *orth* と配列 *a* の行数は同じです。配列 *orth* の列数は配列 *a* のランクです。関数 **orthonormalbase()** は、成功すると 0 を返し、失敗すると-1 を返します。行列 **A** の null 空間 *S* は、次の条件を満たします。

$$\mathbf{S^T S} = I$$
$$\mathbf{AS} = \mathbf{0}$$

関数 **nullspace()** のプロトタイプは

```
int nullspace(array double complex null[&][&],
              array double complex a[&][&]);
```

であり、次元 m × n を持つ配列 **a** の null 空間の正規直交基底を計算します。*null* の列は正規直交です。配列 null と配列 *a* の行数は同じです。配列 *null* の列数は、配列 *a* 列数から *a* のランクの値を引いた数です。関数 **nullspace()** は、成功すると 0 を返し、失敗すると-1 を返します。たとえば、正規直交基底およびランク 2 の特異行列の null 空間の正規直交基底は、次のコマンドで計算できます。

```
> #define M 3
> #define N 3
> array double a[M][N] = {1, 2, 3, \
                          4, 5, 6, \
                          7, 8, 9}
> int r
> r = rank(a)
2
> array double orth[M][r]
> orthonormalbase(orth, a)
> orth
-0.2148 0.8872
-0.5206 0.2496
-0.8263 -0.3879
> transpose(orth)*orth
1.0000 -0.0000
-0.0000 1.0000
> array double null[M][N-r]
```

```
> nullspace(null, a)
> null
-0.4082
0.8165
-0.4082
> transpose(null)*null
1.0000
```

24.13 固有値と固有ベクトル

正方行列 **A** の固有値 λ と固有ベクトル **V** は次のように定義されます。

$$\mathbf{AV} = \lambda\mathbf{V}$$

行列 A が対称でない場合、またはすべての要素が実数型でない場合は、固有値 λ と固有ベクトル **V** は複素数になることがあります。

関数 **eigen**() のプロトタイプは

```
int eigen(... /* double [complex] a[:][:],
                 double [complex] evalues[:],
                 double [complex] evectors[:][:],
                 [char *mode] */);
```

であり、n × n の次元を持つ行列 *a* の固有値 `evalues` と固有ベクトル `evectors` を計算します。この関数は、次の構文で呼び出します。

```
eigen(a, evalues);
eigen(a, evalues, evectors);
eigen(a, evalues, evectors, mode);
```

計算された固有値と固有ベクトルは、それぞれ、関数の引数 `evalues` と `evectors` として渡されます。`a`、`evalues`、および `evectors` は、double 型または double complex 型にする必要があります。計算された固有ベクトルは正規化されるため、各固有ベクトルのノルムは 1 になります。引数 *mode* を使用して、計算前に仮バランスを取るステップを実行するかどうかを指定します。通常、バランスを取ることで指定された行列の条件が改善され、固有値と固有ベクトルを正確に計算できるようになります。ただし、いくつかの特別な場合において、固有ベクトルが不正確になることがあります。既定では、仮バランスを取るステップが取られます。たとえば、以下のような行列があります。

$$\mathbf{A} = \begin{bmatrix} 0.8 & 0.2 & 0.1 \\ 0.2 & 0.7 & 0.3 \\ 0.1 & 0.3 & 0.6 \end{bmatrix}, \mathbf{B} = \begin{bmatrix} 0.8 & 0.2 & 0.1 \\ 0.1 & 0.7 & 0.3 \\ 0.1 & 0.1 & 0.6 \end{bmatrix}, \mathbf{C} = \begin{bmatrix} 3 & 9 & 23 \\ 2 & 2 & 1 \\ -7 & 1 & -9 \end{bmatrix},$$

行列 **A** は実数の固有値を持つ対称行列です。行列 **B** は実数の固有値を持つ非対称行列です。行列 **C** は複素数の固有値を持つ非対称行列です。これら 3 つの行列の固有値と固有ベクトルは、次のコマンドで計算できます。

```
> array double a[3][3] = {0.8,0.2,0.1, 0.2,0.7,0.3, 0.1,0.3,0.6}
> array double b[3][3] = {0.8,0.2,0.1, 0.1,0.7,0.3, 0.1,0.1,0.6}
> array double c[3][3] = {3, 9, 23, 2, 2, 1, -7, 1, -9}
> array double evalues[3], evectors[3][3]
> array double complex zvalues[3], zvectors[3][3]
> eigen(a, evalues)
> evalues
1.1088 0.6526 0.3386
> eigen(b, evalues, evectors)
> evalues
1.0000 0.6000 0.5000
> evectors
-0.7448 -0.7071 0.4082
-0.5793 0.7071 -0.8165
-0.3310 0.0000 0.4082
> eigen(c, evalues)
> evalues
NaN NaN 3.2417
> eigen(c, zvalues, zvectors)
> printf("%.3f", zvalues)
complex(-3.621,10.647) complex(-3.621,-10.647) complex(3.242,0.000)
> printf("%.3f", zvectors)
complex(0.854,0.000) complex(0.854,0.000) complex(0.604,0.000)
complex(-0.024,-0.125) complex(-0.024,0.125) complex(0.744,0.000)
complex(-0.236,0.445) complex(-0.236,-0.445) complex(-0.285,0.000)
```

24.14 高速 Fourier 変換

離散 Fourier 変換および逆変換のペアは次のように定義されます。

$$Y(k) = \sum_{j=0}^{N-1} x(j)\omega_N^{jk}, (k = 0, 1, \cdots, N-1);$$

$$x(j) = \sum_{k=0}^{N-1} Y(k)\omega_N^{-jk}, (j = 0, 1, \cdots, N-1);$$

ここで、$\omega_N = e^{(-2\pi i)/N}$ は、n 番目の単位根です。関数 **fft()** と関数 **ifft()** のプロトタイプは

```
int fft(array double complex &y, array double complex &x, ...
        /* [int n [int dim [&]]] */);
int ifft(array double complex &x, array double complex &y, ...
         /* [int n [int dim [&]]] */);
```

であり、上記の変換および逆変換の評価には、高速 Fourier 変換 (FFT) アルゴリズムが使用されます。このアルゴリズムは、1 次元、2 次元、および 3 次元の高速 Fourier 変換に使用できます。多次元配列 (最大 3 次元) の *x* と *y* には、サポートされている任意の算術データ型およびサイズを使用できます。データは、内部処理で、double complex 型へ変換されます。配列 *x* と同じ次元を持つ配列 *y* には、高速 Fourier 変換の結果が含まれます。int 型のオプション引数 *n* を使用して、1 次元データの高速 Fourier 変換のデータ点の数を指定します。2 番目の引数に入力された配列の長さが n より短い場合、入力された配列データには、n の長さになるまで末尾にゼロが埋め込まれます。2 番目の引数に入力された配列の長さが n より長い場合、代入された配列のデータは切り詰められます。多次元のデータの場合、int 型のオプション配列引数 *dim* には、ユーザーが指定した高速 Fourier 変換のデータ点が含まれます。2 番目の引数に入力された配列の次元は、配列 *dim* に含まれます。

たとえば、3 次元データ *x[m][n][l]* の場合、配列の高速 Fourier 変換のデータ点は m、n、l で指定されます。そこで、配列 *dim* に、dim[0] $= m$、dim[1] $= n$ および dim[2] $= l$ の値を指定します。1 次元配列の場合と同様、2 番目の引数に入力された配列の長さが配列 dim に指定された値より小さい場合、代入された配列は、配列 *dim* に指定された長さになるまで末尾にゼロが埋め込まれます。2 番目の引数に入力された配列の長さが配列 dim に指定された値より大きい場合、指定されたデータは切り詰められます。オプション引数が渡されない場合、高速 Fourier 変換のデータ点は、2 番目の引数に入力された配列から取得されます。関数 **fft**() と関数 **ifft**() では、n(配列のサイズ) を 2 の乗数にしなければばならないという制約はありません。関数 **fft**() と関数 **ifft**() は、成功すると 0 を返し、失敗すると-1 を返します。関数 **unwrap**() は、π より大きい絶対ジャンプを $2*\pi$ 補正へ変更することで、高速 Fourier 変換と逆高速 Fourier 変換で得られた複素数の位相角を調整するのに役立ちます。

たとえば、1 次元配列 $\mathbf{x} = (0, 0.25, 0.5, 0.75, 1)$ の高速 Fourier 変換と逆高速 Fourier 変換は、次のコマンドで計算できます。

```
> array double x[5]
> array double complex x1[5], yy1[5], x2[3], y2[3]
> lindata(0, 1, x)
> x
0.0000 0.2500 0.5000 0.7500 1.0000
> fft(yy1,x)
> printf("%.3f",yy1)
complex(2.500,0.000) complex(-0.625,-0.860) complex(-0.625,-0.203)\
complex(-0.625,0.203) complex(-0.625,0.860)
> ifft(x1,yy1)
> printf("%.3f",x1)
complex(0.000,0.000) complex(0.250,0.000) complex(0.500,0.000) \
complex(0.750,0.000) complex(1.000,0.000)
> fft(y2,x,3)
> printf("%.3f",y2)
complex(0.750,0.000) complex(-0.375,-0.217) complex(-0.375,0.217)
> ifft(x2,y2,3)
> printf("%.3f",x2)
complex(0.000,0.000) complex(0.250,0.000) complex(0.500,0.000)
> fft(yy1,x2,5)
```

```
> printf("%.3f",yy1)
complex(0.750,0.000) complex(-0.327,0.532) complex(-0.048,-0.329) \
complex(-0.048,0.329) complex(-0.327,-0.532)
> ifft(x1,yy1)
> printf("%.3f",x1)
complex(0.000,0.000) complex(0.250,0.000) complex(0.500,0.000)
complex(0.000,0.000) complex(0.000,0.000)
```

次の 2 次元配列

$$\mathbf{x} = \begin{bmatrix} 0 & 0.2 \\ 0.4 & 0.6 \\ 0.8 & 1 \end{bmatrix}$$

の高速 Fourier 変換と逆高速 Fourier 変換は、次のコマンドで計算できます。

```
> array double x[3][2]
> array double complex yy1[3][2], x1[3][2]
> array double complex y2[2][2], x2[2][2]
> int dim[2] = {2, 2}
> lindata(0, 1, x)
> x
0.000000 0.200000
0.400000 0.600000
0.800000 1.000000
> fft(yy1, x)
> printf("%.3f",yy1)
complex(3.000,0.000) complex(-0.600,0.000)
complex(-1.200,-0.693) complex(0.000,-0.000)
complex(-1.200,0.693) complex(0.000,0.000)
> ifft(x1, yy1)
> printf("%.3f",x1)
complex(0.000,0.000) complex(0.200,0.000)
complex(0.400,0.000) complex(0.600,0.000)
complex(0.800,0.000) complex(1.000,0.000)
> fft(yy1, x, dim)
> printf("%.3f",yy1)
complex(1.200,0.000) complex(-0.400,0.000)
complex(-0.800,0.000) complex(0.000,0.000)
complex(0.000,0.000) complex(0.000,1.200)
> fft(y2, x, dim)
> printf("%.3f",y2)
complex(1.200,0.000) complex(-0.400,0.000)
complex(-0.800,0.000) complex(0.000,0.000)
```

```
> ifft(x2, y2, dim)
> printf("%.3f",x2)
complex(-0.000,0.000) complex(0.200,0.000)
complex(0.400,0.000) complex(0.600,0.000)
> ifft(x2, y2)
> printf("%.3f",x2)
complex(-0.000,0.000) complex(0.200,0.000)
complex(0.400,0.000) complex(0.600,0.000)
```

24.15 畳み込みとフィルタリング

2 つの関数 $x(t)$ および $y(t)$ の畳み込み ($x * y$ と表記される) は、次のように定義されます。

$$x * y \equiv \int_{-\infty}^{\infty} x(\tau)y(t-\tau)d\tau$$

定義域全体では、$x * y$ と $y * x$ は等しくなります。畳み込みの定理によれば、$X(f)$ および $Y(f)$ をそれぞれ $x(t)$ および $y(t)$ の Fourier 変換としたとき、すなわち、

$$x(t) \Longleftrightarrow X(f) \qquad \text{AND} \qquad y(t) \Longleftrightarrow Y(f)$$

のとき、次の関係が成り立ちます。

$$x * y \Longleftrightarrow X(f)Y(f)$$

関数が 2 つの配列 (サイズ n の x とサイズ m の y) として数値化される場合、2 つの配列 x と y のサイズは $m+n-1$ の大きさになるまで拡張され、内部処理でゼロが埋め込まれます。高速 Fourier 変換のアルゴリズムを使用して、x と y の離散型 Fourier 変換を計算できます。2 つの変換を成分ごとに乗算し、次に逆高速 Fourier 変換のアルゴリズムを使用して、積の逆離散型 Fourier 変換を行うと、配列 x と y の畳み込みが求められます。関数 **conv()** のプロトタイプは

```
int conv(array double complex c[&],
         array double complex x[&], array double complex y[&]);
```

であり、サイズ n の配列 x とサイズ m の配列 y の畳み込みを計算します。配列¿x と y の両方が実数型の場合、結果 c はサイズ $n+m-1$ の 1 次元配列になります。x と y のどちらかが複素数型の場合、結果 c は複素数型になります。x と y を多項式係数の 2 つのベクトルとした場合、x と y の畳み込みはこの 2 つの多項式の積と等しくなります。逆重畳は畳み込みの逆の演算です。関数 **deconv()** のプロトタイプは

```
int deconv(array double complex u[&], array double complex v[&],
      array double complex q[&], ... /* array double complex r[&]);
```

であり、長除法を使ってベクトル u からベクトル v の逆重畳を求めます。u と v が多項式係数の 2 つのベクトルとした場合、v からの u の逆重畳は多項式の商と等しくなります。商はベクトル q に格納され、剰余はオプション引数 r に格納され、u = conv(v, q)+r が得られます。サイズ n の配列 u と v サイズ m の配列 v の両方が実数型の場合、サイズ $n-m+1$ の商 q とサイズ n の剰余 r は実数型です。u と v のどちらかが複素数型の場合、結果の q と r は複素数型になります。たとえば、次のような $x(t)$ と $y(t)$ の多項式関数があります。

$$\begin{aligned} x(t) &= t^5 + 2*t^4 + 3*t^3 + 4*t^2 + 5*t + 6; \\ y(t) &= 6*t + 7; \end{aligned}$$

$x(t)*y(t)$ の畳み込みまたは 2 つの多項式の $x(t)$ と $y(t)$ の乗算結果は、次のようになります。

$$c(t) = x(t)*y(t) = 6*t^6 + 19*t^5 + 32*t^4 + 45*t^3 + 58*t^2 + 71*t + 42$$

$c(t)$ からの $y(t)$ の逆重畳または $c(t)$ からの $x(t)$ の逆重畳は、多項式 $c(t)$ を $y(t)$ で、または $c(t)$ を $x(t)$ で乗算した結果となります。これらの計算は、次のコマンドで実行できます。

```
> array double x[6]={1,2,3,4,5,6},y[2]={6,7}, c[6+2-1]
> conv(c,x,y)
> printf("%.2f", c)
6.00 19.00 32.00 45.00 58.00 71.00 42.00
> deconv(c,y,x)
> printf("%.2f", x)
1.00 2.00 3.00 4.00 5.00 6.00
> deconv(c,x,y)
> printf("%.2f", y)
6.00 7.00
```

2 次元畳み込みは、1 次元畳み込みと類似しています。すなわち、$f(x,y)$ と $g(x,y)$ の 2 つの関数の畳み込みは、次のように定義されます。

$$f(x,y)*g(x,y) = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} f(\alpha,\beta)g(x-\alpha,y-\beta)d\alpha d\beta$$

$f(x,y)$ および $g(x,y)$ を 2 次元 Fourier 変換したものを、それぞれ $F(u,v)$ および $G(u,v)$ とすると、畳み込みの定理に従って、以下の関係が成り立ちます。

$$f(x,y)*g(x,y) \Longleftrightarrow F(u,v)G(u,v)$$

関数 **conv2()** のプロトタイプは

```
int conv2(array double complex c[&][&], array double complex f[&][&],
          array double complex g[&][&], ... /* [string_t method] */);
```

であり、行列 f と g の 2 次元畳み込みを計算します。この計算では、f のサイズを n_a x n_b、g のサイズを m_a x n_b とした場合、行列 f と g はそれぞれ $(n_a + m_a - 1)$ x $(n_b + m_b - 1)$ のサイズまで拡張され、内部処理でゼロが埋め込まれます。高速 Fourier 変換のアルゴリズムを使用して、x と y の離散型 Fourier 変換を計算できます。2 つの変換を成分ごとに乗算し、次に逆高速 Fourier 変換のアルゴリズムを使用して、積の逆離散型 Fourier 変換を行うと、行列 f と g の畳み込みが求められます。配列 c のサイズはオプション引数 *method* の値に依存します。オプション引数 *method* の値が`"full"`の場合、各次元の c のサイズは、入力された行列の対応する次元の合計から 1 を引いたものに等しくなります。オプション引数 *method* の値が`"same"`の場合、c には、行列 f と同じサイズの 2 次元畳み込みの中央部が含まれます。オプション引数 *method* の値が`"valid"`の場合、c には、計算された 2 次元畳み込みのみが含まれ、ゼロは埋め込まれません。c のサイズは $(m_a - m_b + 1) \times (n_a - n_b + 1)$ で、f のサイズは g のサイズより大きくする必要があります。既定では、引数 *method* の値は`"full"`になります。

たとえば、画像処理の Sobel フィルタは、エッジ検出の概念と似ています。元の 2 次元画像データを、次のように Sobel フィルタと畳み込みします。

$$g_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

これにより、画像の x 方向のエッジが検出されます。同様に、元の 2 次元画像データを、行列 g_x の転置行列である次の Sobel フィルタと畳み込みします。

$$g_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

これにより、Y 方向のエッジが検出されます。プログラム 24.9を使用して、関数 **conv2()** を使ったエッジ検出を計算できます。出力結果は図 24.9に示すとおりです。

```
#include <math.h>
#include <chplot.h>
#include <numeric.h>

int main() {
   int i, j;
   array double g[3][3]={{-1,0,1},{-2,0,2},{-1,0,1}};
   array double x[16], y[16], z1[256], f[16][16], H[18][18], V[18][18],Z[18][18], Z1[256];

   linspace(x,0,16);
   linspace(y,0,16);
   for(i=3; i<13; i++)
      for(j=3; j<13; j++) {
         z1[i*16+j]=1;
         f[i][j] = 1;
   }
   plotxyz(x,y,z1);                    /* original image */
   conv2(H,f,g);
   conv2(V,f,transpose(g));
   Z = H .* H + V .* V;                /* magnitude of the pixel value */
   for(i = 0; i<16; i++)
      for(j=0; j<16; j++)
         Z1[i*16+j] = Z[i+1][j+1];
   plotxyz(x,y,Z1);                    /* edge finded image */
}
```

プログラム 24.9: **conv2**() を使用したプログラム例

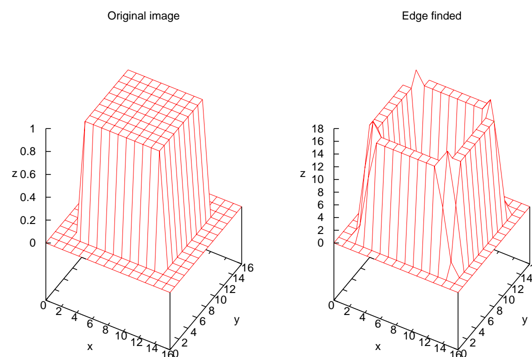


図 24.9: 2 次元畳み込み関数 **conv2**() の出力結果

関数 **filter**() のプロトタイプは

```
int filter(array double complex v[&], array double complex u[&],
     array double complex x[&], array double complex y[&], ...
     /* [array double complex zi[&], array double complex zf[&]] */);
```

であり、関数 **filter**() は、ベクトル u と v で表されるフィルタでベクトル x のデータをフィルタし、フィルタ後のデータ y を作成します。フィルタは、次の標準の差分式を直接型 II 転置で実装したものです。

$$\begin{aligned} y(n) &= v_0 * x(n) + v_1 * x(n-1) + ... + v_{nb} * x(n-nb-1) \\ &\quad -u_1 * y(n-1) - ... - u_{na} * y(n-na-1) \end{aligned}$$

z 変換でのこのフィルタ演算の入出力を記述するのは、次のような有理伝達関数です。

$$Y(z) = \frac{v_0 + v_1 z^{-1} + \cdots + v_{nb-1} z^{-nb-1}}{1 + u_1 z^{-1} + \cdots + u_{na-1} z^{-na-1}} X(z)$$

システム伝達関数 *v* にゼロを含む分子係数、システム伝達関数 *u* に極を含む分母係数、および代入データのベクトル *x* では、サポートされている任意の算術データ型およびサイズを使用できます。データは、内部処理で、double complex 型へ変換されます。ベクトル *y*(ベクトル x と同じサイズ) には、フィルタ後の出力結果が含まれます。オプション引数 *zi* に遅延の初期値を設定すると、オプション引数 *zf* にフィルタの最終遅延が得られます。これらの引数は double complex データ型にする必要があります。ベクトル *u* と *v* のサイズを、それぞれ、n_a と n_b とした場合、ベクトル *zi* と *zf* のサイズは、それぞれ $(\max(n_a, n_b) - 1)$ と $\max(n_a, n_b)$ になります。他の係数が分母 u_0 で除算されるため、分母 u_0 の先頭係数はゼロ以外でなければなりません。

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

#define N 512

int main() {
   array double t[N], x[N], y[N], Pyy[N/2], f[N/2], u[7], v[7];
   array double complex Y[N];
   int i;
   class CPlot plot;

   u[0]=1;u[1]=-5.66792131;u[2]=13.48109005;u[3]=-17.22250511;
   u[4]=12.46418230;u[5]=-4.84534157;u[6]=0.79051978;
   v[0]=0.00598202; v[1]=-0.02219918; v[2]=0.02645738; v[3]=0;
   v[4]=-0.02645738;v[5]=0.02219918;v[6]=-0.00598202;

   linspace(t,0,N-1);
   t = t/N;
   for (i=0; i<N; i++) {
      x[i] = sin(2*M_PI*5*t[i]) + sin(2*M_PI*15*t[i]) + sin(2*M_PI*t[i]*30);
      x[i]=x[i]+3*(urand(NULL)-0.5);
   }

   filter(v,u,x,y);
   plotxy(t,x,"Time domain original signal","Time (second)","Magnitude ");
   plotxy(t,y,"Time domain filtered signal","Time (second)","Magnitude ");

   fft(Y,x);
   for (i=0; i<N/2; i++)
      Pyy[i] = abs(Y[i]);
   linspace(f,0,N/2);
   plotxy(f,Pyy,"Frequency domain original signal","frequency (Hz)","Magnitude (db)");
   fft(Y,y);
   for (i=0; i<N/2; i++)
      Pyy[i] = abs(Y[i]);
   linspace(f,0,255);
   plotxy(f,Pyy,"Frequency domain filtered signal","frequency (Hz)","Magnitude (db)");
}
```

プログラム 24.10: **filter**() を使用したプログラム例

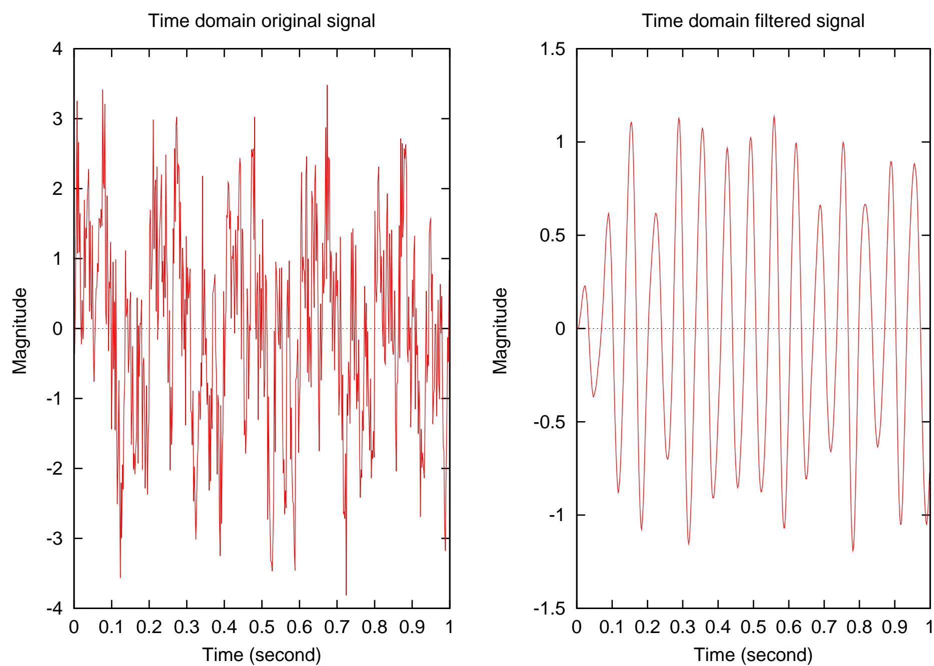


図 24.10: フィルタ処理前の信号と処理後の信号 (時間領域)

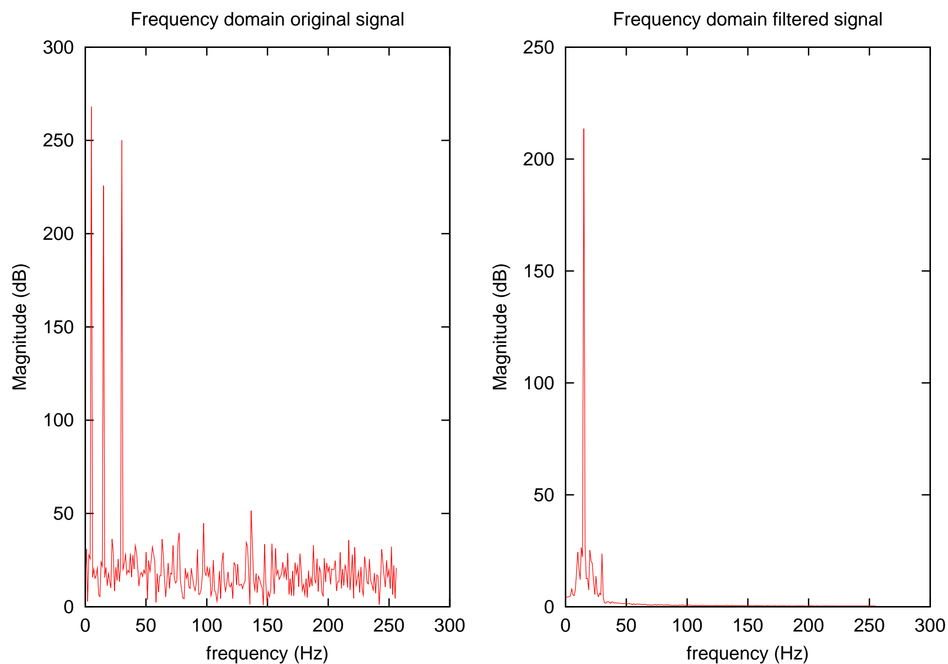


図 24.11: フィルタ処理前の信号と処理後の信号 (周波数領域)

関数 **filter**() の使用例をプログラム 24.10に示します。この例は、雑音を含む信号のスペクトラムを検出し、高速 Fourier 変換アルゴリズムを使用して不要な信号をフィルタする方法を示しています。図 24.10は、フィルタ処理前の信号と処理後の信号を時間領域で示しています。左側の図は、周波数

5Hz、15Hz、30Hz の 3 つの正弦波成分を含むフィルタ処理前の信号を示しています。これらの信号は、一様乱数ジェネレータ関数 **urand**() でシミュレートされたホワイトノイズを含んでいます。右側の図は、10Hz〜20Hz の通過帯域を持つ 6 次 IIR フィルタで処理された信号を示しています。プログラム 24.10に記述されている係数 *u* と *v* を持つフィルタは、15Hz の正弦波信号を保持し、5Hz と 30Hz の正弦波信号とその他のホワイトノイズを除去します。周波数領域の信号は、関数 **fft**() を使用する高速 Fourier 変換で得られます。図 24.11は、フィルタ処理前の信号と処理後の信号を周波数領域で示しています。左側の図は、3 つの主要周波数スペクトラムといくつかのノイズ周波数を含むフィルタ処理前の信号を示しています。右側の図は、フィルタ処理後の信号を示しています。主に 15Hz の周波数といくつかのノイズ成分が含まれています。関数 **filter**2() のプロトタイプは

```
int filter2(array double complex y[&][&], array double complex u[&],
        array double complex x[&], ...  /* [string_t method] */);
```

であり、関数 **filter**2() は、サイズ $(n_x \times m_x)$ の入力行列 *x* を持つ FIR フィルタの完全な 2 次元畳み込みを計算します。関数 **filter**2() 内で、まず入力フィルタ u が 180 度回転され、次に関数 **conv2**() が呼び出されてフィルタリングの演算が実行されます。フィルタ行列 *u* のサイズを $(n_x \times m_x)$ とすると、フィルタ処理後のデータの配列 *y* のサイズはオプション引数 *method* の値によって決まります。オプション引数 method の値が`"same"`の場合、*y* には、行列 *x* と同じサイズの 2 次元畳み込みの中央部が含まれます。

既定では、引数 *method* の値は`"same"`になります。元の引数 *method* の値が`"full"`の場合、各次元の *y* のサイズは、入力された行列の対応する次元の合計から 1 を引いたものに等しくなります。オプション引数 *method* の値が`"valid"`の場合、*y* には、計算された 2 次元畳み込みのみが含まれ、ゼロは埋め込まれません。*y* のサイズは $(m_x - m_u + 1) \times (n_x - n_u + 1)$ で、*x* のサイズは *u* のサイズより大きくする必要があります。たとえば、次の画像処理の Sobel フィルタ

$$g_x = \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

は、画像のスムージングに使用できます。プログラム 24.11は、上記の Sobel マスクを使用して画像を畳み込みする方法を示しています。図 24.12に、フィルタ処理前の画像と関数 **filter**2() を使ったフィルタ処理後の画像を示します。

```
#include <math.h>
#include <chplot.h>
#include <numeric.h>

int main() {
   int i, j;
   array double u[3][3]={{1,2,1},{2,4,2},{1,2,1}};
   array double s[3][3]={{1,2,1},{2,4,2},{1,2,1}};
   array double x[16],y[16],z1[256],z[16][16],Z[18][18],Z1[256];

   linspace(x,0,16);
   linspace(y,0,16);
   for(i=3; i<13; i++)
      for(j=3; j<13; j++) {
         z1[i*16+j]=1;
         z[i][j] = 1;
   }
   plotxyz(x,y,z1);
   filter2(Z,u,z,"full");
   for(i = 0; i<16; i++)
      for(j=0; j<16; j++)
         Z1[i*16+j] = Z[i+1][j+1];
   plotxyz(x,y,Z1);
}
```

プログラム 24.11: **filter2**() を使用したプログラム例

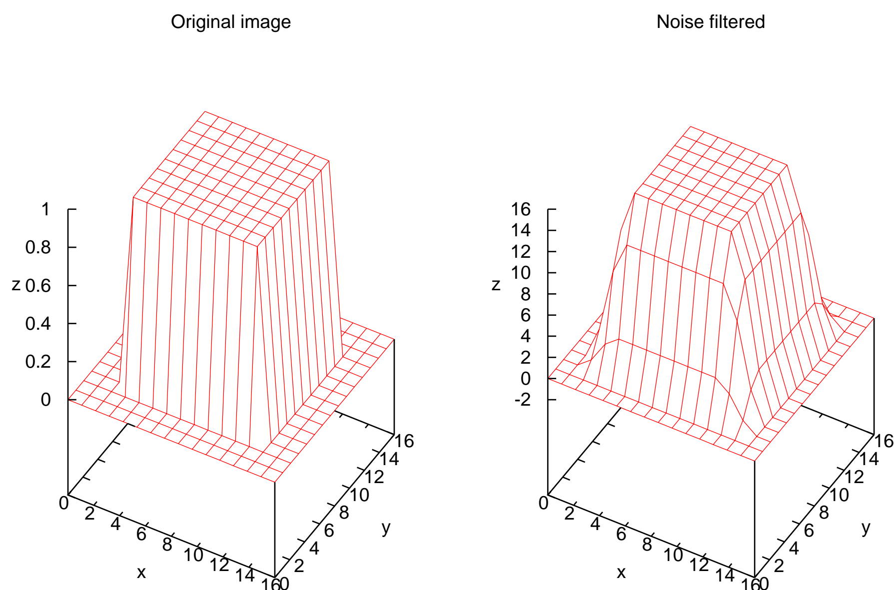


図 24.12: フィルタ処理前の画像と関数 **filter2**() を使ったフィルタ処理後の画像

24.16 相互相関

2 つの関数 $x(t)$ および $y(t)$ に対して、それぞれ対応する Fourier 変換を $X(f)$ および $Y(f)$ とした場合、これら 2 つの関数の相互相関 ($xcorr(x,y)$) は、次のように定義されます。

$$xcorr(x,y) \equiv \int_{-\infty}^{\infty} x(t+\tau)y(t)d\tau$$

$$xcorr(x,y) \iff X(f)Y^*(f)$$

アスタリスクは複素共役を示します。関数とそれ自身との相関は、自己相関と呼ばれます。Fourier 変換との対応は次のようになります。

$$xcorr(x,x) \Longleftrightarrow |X(f)^2|$$

数値処理では、同じサイズ n の 2 つのデータシーケンス x のサイズは、$(2n-1)$ のサイズまで拡張され、内部処理でゼロが埋め込まれます。高速 Fourier 変換アルゴリズムを使用して、まず、x と y の離散 Fourier 変換が計算されます。次に、y の変換の複素共役と x の変換が成分ごとに掛け合わされます。最後に、その積を逆 Fourier 変更すると、配列 x と y の相互相関 $xcorr(x,y)$ が得られます。関数 **xcorr**() のプロトタイプは

```
int xcorr(array double complex c[&],
          array double complex x[&], array double complex y[&]);
```

であり、同じサイズ n を持つ 2 つの配列 x と y の相互相関を計算します。配列 x と y の両方が実数型であるなら、相互相関 c はサイズ $(2n-1)$ の 1 次元配列になります。x と y のどちらかが複素数型の場合、結果 c は複素数型になります。たとえば、2 つのデータ列 $x[n]$ と $y[n]$ の解析では、x と y の相互相関は次の式で計算できます。

$$c[k] = \sum_{j=\max\{1,k-n+1\}}^{\min\{k,m\}} y[j]x[n+j-k]; \qquad k = 1, 2, \cdots, n+m-1$$

ここで、$x = \{1,2,3,4\}$ および $y = \{3,2,0,1\}$ とすると、次のようになります。

$$\begin{aligned}
c[1] &= y[1]x[4] = 12 \\
c[2] &= y[1]x[3] + y[2]x[4] = 17 \\
c[3] &= y[1]x[2] + y[2]x[3] + y[3]x[4] = 12 \\
c[4] &= y[1]x[1] + y[2]x[2] + y[3]x[3] + y[4]x[4] = 11 \\
c[5] &= y[2]x[1] + y[3]x[2] + y[4]x[3] = 5 \\
c[6] &= y[3]x[1] + y[4]x[2] = 2 \\
c[7] &= y[4]x[1] = 1
\end{aligned}$$

上記の相互相関は、次のコマンドで計算できます。

```
> #define N 4
> array double x[1:N] = {1, 2, 3, 4}
> array double y[1:N] = {3, 2, 0, 1}, c[1:2*N-1]
> xcorr(c,x,y)
> printf("%.2f", c)
12.00 17.00 12.00 11.00 5.00 2.00 1.00
> c[1]
12.000000
> c[7]
1.000000
```

第25章　参考文献

1. ANSI, *ANSI Standard X3.9-1978, Programming Language FORTRAN* (revision of ANSI X2.9-1966), American National Standards Institute, Inc., NY, 1978.

2. Cheng, H. H., The Ch Language Environment Homepage, `http://www.softintegration.com/docs/ch/` .

3. Cheng, H. H., CGI Programming in C, *C/C++ Users Journal*, Vol. 14, No. 11, November, 1996, pp. 17-21.

4. Cheng, H. H., Scientific Computing in the Ch Programming Language, *Scientific Programming*, Vol. 2, No. 3, Fall, 1993, pp. 49-75.

5. Cheng, H. H., Handling of Complex Numbers in the Ch Programming Language, *Scientific Programming*, Vol. 2, No. 3, Fall, 1993, pp. 76-106.

6. Cheng, H. H., Extending C with Arrays of Variable Length, *Computer Standards and Interfaces*, Vol. 17, 1995, pp. 375-406.

7. Cheng, H. H., Extending C and FORTRAN for Design Automation, *ASME Trans., Journal of Mechanical Design*, Vol. 117, No. 3, 1995, pp. 390-395.

8. Cheng, H. H., Programming with Dual Numbers and its Applications in Mechanisms Design, *Engineering with Computers, An International Journal for Computer-Aided Mechanical and Structural Engineering*, Vol. 10, No. 4, 1994, pp. 212-229.

9. Cheng, H. H., Adding References and Nested Functions to C for Modular and Parallel Programming, The ANSI C Standard Committee X3J11.1 Meeting, NCEG, X3J11.1/93-044, October 22, 1993.

10. Churchill, R. V. and Brown, J. W., Churchill, R. V. and Brown, J. W., *Complex Variables and Applications*, Fourth edition, McGraw-Hill Book Co., NY, 1984.

11. IEEE, *ANSI/IEEE Standard 754-1985, IEEE Standard for Binary Floating-Point Arithmetic*, Institute of Electrical and Electronic Engineers, Inc., Piscataway, NJ, 1985.

12. ISO, *ISO/IEC Standard 9899:1990, Programming Language C*, International Standards Organization, Geneva, 1990.

13. ISO/IEC, *Information Technology, Programming Languages - FORTRAN*, 1539:1991E, ISO, Geneva, Switzerland.

14. Joy, W., *An Introduction to the C Shell*, Department of Electrical Engineering and Computer Science, University of California at Berkeley, 1980.

15. Kahan, W., Branch Cuts for Complex Elementary Functions, or Much Ado about Nothing's Sign Bit, *The State of the Art in Numerical Analysis* (ed. Iserles & Powell), 1987, Oxford Univ. Press; *Proc. of the Joint IMA/SIAM Conference*, April 14-18, 1986.

16. Kernighan, B. W. and Ritchie, D. M., *The C Programming Language*, Prentice-Hall, Inc., Englewood Cliffs, NJ, second edition, 1988.

17. Marsden, J. E., *Basic Complex Analysis*, W. H. Freeman and Company, San Francisco, 1973.

18. The MathWorks, Inc., *MATLAB Function Reference*, Version 6, 2001.

19. Stroustrup, B., *The C++ Programming Language*, Addison-Wesley, Publishing Company, Inc., 1987.

付録A　既知の問題とプラットフォーム特有の機能

A.1　プラットフォーム固有の機能

Ch のほとんどの関数は、異なる複数のプラットフォームでサポートされています。特定のプラットフォームでサポートされていない関数は、『*Ch* **言語環境 - リファァレンスガイド**』に記載されています。他のプラットフォーム固有の機能と問題について以下に説明します。

A.1.1　Solaris

1. システム変数`_ignoreeof`は true に設定できません。EOF は起動セッションでログインシェルを終了するために使用されるからです。

A.1.2　Windows NT/2000/XP/Vista/Windows 7

1. Ch 言語環境のユーザーは、Windows NT/2000/XP/Vista/Windows 7 で Unix 環境と同じように作業することができます。

   スラッシュ/がディレクトリ表記として扱われます。パスの前に "ドライブ名:"を付けることによって、ディスクドライブを指定できます。たとえば、ドライブ C は **C:/Ch/bin/ch** とのように指定できます。Windows コマンドシェルのすべての DOS 固有コマンドが Ch シェルでサポートされています。

2. 表 A.1に示す MS-DOS 固有コマンドは、Ch シェルで Windows とまったく同じように使用できます。たとえば、スラッシュ(/) を使用してコマンドのオプションを指定できます。

3. 表 A.2に示す MS-DOS 固有コマンドは、Windows NT/2000/XP/Vista/Windows 7 の Ch シェルで Windows NT/2000/XP/Vista/Windows 7 とまったく同じように使用できます。

   たとえば、スラッシュ(/) を使用してコマンドのオプションを指定できます。

A.2　特定のプラットフォームでサポートされていない関数

『*The Ch Language Environment — Reference Guide*』の付録にある「Functions Not Supported in Specific Platforms」を参照してください。

表 A.1: Ch シェルでの DOS コマンド

コマンド	説明
cls	画面をクリアします。
copy	1 つまたは複数のファイルを別の場所にコピーします。
del	1 つまたは複数のファイルを削除します。
dir	ディレクトリ内のファイルおよびサブディレクトリを一覧表示します。
endlocal	バッチファイル内での環境変数の一時的な変更を終了し、 setlocal コマンドを実行する前の設定に復元します。
erase	1 つまたは複数のファイルを削除します。
ren	1 つまたは複数のファイルの名前を変更します。 rd はコマンドプロンプトでのみ有効です。
setlocal	バッチファイル内で環境変数を一時的に変更することが できるようにします。endlocal コマンドが実行されると、 setlocal コマンドを実行する前の状態に戻ります。
time	システム時間を表示または設定します。
title	コマンドプロンプトウィンドウのタイトルを設定します。
type	ファイルの内容を表示します。
ver	Windows バージョン番号を表示します。
verify	ファイルがディスクに正しく書き込まれたことを 検証するかどうかを Windows に指示します。
vol	ディスクボリュームラベルと通し番号を表示します。

表 A.2: Windows NT の Ch シェルでの DOS コマンド

コマンド	説明
move	ファイルまたはディレクトリを移動します (Windows NT 内でのみ)。
start	別のウィンドウを起動してプログラムまたはコマンドを実行します (Windows NT 内でのみ)。

付 録B　Cの動作と実装で定義される動作の比較

B.1　Ch でサポートされている C99 の新機能

Ch は C89 規格の機能をすべてサポートしています。Ch は ISO C99 規格で追加された以下の主な新機能もサポートしています。

1. 浮動小数点演算に関する IEEE 754 規格の機能は、ユーザーに対して透過的です。実数は、±0.0、±Inf、および NaN というメタ数値を使用して 実数線全体で表現されます。実数を使用する数学関数は実数域全体に定義されています。

2. float 複素数、double 複素数、long double 複素数のデータ型。

3. long long および符号なし long long のデータ型。

4. 可変長配列 (VLA)。

5. ポリモーフィックな汎用関数。

6. 以下のような実行可能なコードと宣言の混在。

```
x = 4;
int n = 2*x;
int a[n];
```

7. C++ 形式のコメントシンボルである//が追加されています。

8. クラスの関数またはメンバ関数の内部にある識別子__func__は、関数の名前を含みます。The identifier __func__

9. 変数の引数リストマクロでは、引数で省略記号の表記を使用し、置換リストで識別子**__VA_ARGS__**を使用します。次に例を示します。

```
#define debug(...)    printf(__VA_ARGS__)
debug("x = %d\n", x);
```

   結果は、次のようになります。

```
printf("x = %d\n", x);
```

10. ブール型のヘッダーファイル `stdbool.h` と、`true` および `false` のマクロをサポートしています。

11. ヘッダーファイル **wchar.h** および **wctype.h** はもちろん、**complex.h**、**fenv.h**、**inttypes.h**、**iso646.h**、**stdbool.h**、**stdin.h**、**tgmath.h** もサポートしています。

12. 暗黙の int 宣言は削除されています。

```
fun(int i) { // ERROR: implicit int type for fun
    ...
}
int fun2(i) { // ERROR: implicit int type for i
    ...
}
```

13. ヘッダーファイル `stdarg.h.` に `va_copy()` が追加されています。

14. キーワード `restrict` および `inline` を認識します。

15. 16 進浮動小数点定数。たとえば、

```
> 0X2P3
16.0000
> 0x1.1p0
1.0625
> 0x1.1p1F
2.12
```

B.2　C に対する拡張の要約

1. C インタープリタ。Ch は統合言語環境です。異なるプラットフォームにおいて C プログラムをそのまま Ch で実行でき、コンパイル、リンク、実行、デバッグという長いサイクルは不要です。

2. C++のクラス。

3. コマンドインタープリタ。Ch のコマンドモードは C シェルに似ています。

4. セーフ Ch は、複数プラットフォームにまたがるネットワークコンピューティングに使用します。

5. すべての数学関数、入出力関数、およびその他の汎用関数はポリモーフィックです。

6. 複素演算と複素関数はポリモーフィックです。これらは複素数のメタ数値である ComplexInf および ComplexNan を使用して、 複素数域全体に定義されています。省略可能な引数を指定した数学関数によって、複数の値を持つ複素関数のさまざまな分岐を取得できます。複素数の無限数は *1* **つ**だけであり、複素数の非数も *1* **つ**だけです。

7. char、符号なし char、short、符号なし short、int、符号なし int、float、double、float complex、double complex の間での (さらに各型のメタ数値間での)、一貫した暗黙的および明示的なデータ型変換。

8. 組み込みの演算と関数により、正しい数値または NaN、ComplexNaN を確実に取得できます。

9. 文字列型 **string_t** の文字列はファーストクラスオブジェクトであり、シンボリック計算に使用できます。新規の文字列関数 **str2ascii()**、**str2mat()**、**stradd()**、**strgetc()**、**strputc()**、および **strrep()** が追加されました。

10. 関数は入れ子にすることができ、再帰的な入れ子も可能です。関数はまた相互に再帰的に実行できます。つまり、再帰的な関数は相互に呼び出し可能です。

11. Ch では、クラス、構造体、共用体、列挙体のタグと、クラス、構造体、共用体、列挙体の変数は同じ名前空間を共有します。

12. goto ステートメントを使用して、入れ子にされた関数から、ラベルが定義されている上位つまり外側の関数にプログラムの実行を移行できます。

13. 整数定数に加えて文字列も switch ステートメントのインデックスにすることができます。

14. Fortran 90 と同様に、計算配列はファーストクラスオブジェクトとして処理されます。

15. 形状無指定配列、形状引継ぎ配列、および形状引継ぎ配列へのポインタを含む可変長配列がサポートされます。

16. 調整可能な範囲の配列。配列のインデックスを表す添字の範囲は調整できます。

17. クラス、構造体、共用体のメンバを形状引継ぎ配列へのポインタとして使用できます。

18. 単純データ型である char、short、int、float、double、ならびに signed、unsigned、long、complex で修飾されたデータ型に対する参照を宣言できます。関数を参照によって呼び出すことができます。

19. さまざまなデータ型の変数を、変数または配列を介した参照によって関数の引数に渡すことができます。

20. 参照の配列がサポートされています。参照型の配列を使用する関数の引数に、さまざまな形状とデータ型の配列を渡すことができます。

21. 関数および変数は、ある特定のスコープとレキシカルレベルに 1 回だけ定義できます。関数へのすべての関数呼び出しがプロトタイプによって統制されること、同じ関数のすべてのプロトタイプに互換性があること、数多くのファイルに分割されたプログラムの場合でも、すべてのプロトタイプが関数定義を満たすことが保証されます。外部変数についても同様です。すべての宣言内のデータ型と外部変数の定義内のデータ型は、異なるファイル間で互換性がなければなりません。配列の変数の場合、異なる複数の宣言とその定義において、外部配列の形状とエクステントの両方に互換性がなければなりません。

22. 記憶クラス指定子の `__declspec(local)` と `__declspec(global)` が追加されています。

23. C シェル形式の foreach ループがサポートされています。

24. メモリ割り当て関数によって割り当てた変数とメモリは自動的に初期化されます。

25. *exclusive or* の関係演算子`^^`がサポートされています。

26. float(3) のような関数の型キャスト演算がサポートされています。

27. シフト演算の rvalue が負である場合、lvalue のシフト方向が逆になります。

28. クラスのメンバ関数の内部にある識別子`__class__`は、クラスの名前を含みます。クラスのメンバ関数の内部にある識別子`__class_func__`は、クラスおよびメンバ関数の両方の名前を含みます。

29. `_path`や`_fpath`などの複数のシステム変数。

30. 整数演算では最長のデータ型を使用しません。前処理コマンドでの演算と標準コマンドでの演算は、同じ実行環境を使用します。すべての整数および浮動小数点数の演算と汎用関数は、プログラムを実行するときに使用するのと同様に前処理で使用できます。

31. 演算を安全に行うために、配列の要素がアクセスされるときに配列境界がチェックされます。文字配列が文字列として更新される場合、文字列長がチェックされます。

32. `int func(double a, b, c)` のような関数の略式パラメータリストがサポートされています。型定義された識別子を関数パラメータリストの識別子として使用する場合は、型指定子を前に付ける必要があります。

33. この言語には多数の組み込み汎用関数があります。

34. ツールキットと`plotxy()`、`plotxyz()`、`plotxyf()`、`plotxyzf()` などの多数の関数ファイルがあります。

35. 汎用関数 **free**() によって解放されたポインタは NULL にリセットされます。

36. NULL は既定のキーワードです。int データ型と void データ型へのポインタの両方の状態があります。int としては値 0 を持ちます。void へのポインタとしては、どこもポイントしません。前処理ディレクティブ内で使用できます。

```
ifdef NULL is true
ifndef NULL is false
defined(NULL) is true
```

37. C では、アリティ可変の関数には少なくとも 1 つの引数が指定されている必要があります。C++ および Ch ではその必要はありません。

```
int func(...); // ok in C++/Ch, bad in C.
```

38. 関数 **printf()** および **scanf()** が制御文字列なしに使用可能です。セクション 20.4 で説明された既定の I/O 形式が使用されます。たとえば、以下のとおりです。

```
int i = 90;
printf(i, " ", 2*i, "\n");
printf("Please input a number\n");
scanf(&i);
```

39. プログラムは、関数 main() のないスクリプトとして記述可能です。

B.3 実装に関する補足

B.3.1 無制限のプロパティ

以下のプロパティは、 Ch では制限されていませんが、Ch を実行する特定のコンピュータシステムの使用可能なメモリ量によって制限されます。

1. 配列の次元。
2. 複合ステートメントの入れ子のレベル数、for ループ、do ループ、while ループの繰り返し構造の入れ子のレベル数、および選択を制御する switch ステートメントの入れ子のレベル数。
3. 関数の入れ子のレベル数。
4. クラス、構造体、共用体およびスコープの入れ子の深さ。
5. 条件付き組み込みの入れ子のレベル数。
6. 完全な式を含み、かっこで囲まれた式の入れ子のレベル数。
7. マクロ識別子の数。
8. 1 つの関数定義でのパラメータの数。
9. 1 回の関数呼び出しでの引数の数。
10. 1 つのマクロ定義でのパラメータの数。
11. 1 回のマクロ呼び出しでの引数の数。
12. 配列、クラス、構造体、共用体のオブジェクトのサイズ。
13. 1 つのクラス、構造体、共用体でのメンバの数。
14. 1 つの列挙体での要素の数。
15. 1 つの構造体宣言リストで入れ子にしたクラス、構造体、共用体定義のレベル数。
16. 関数呼び出しスタックの深さの数。
17. 型定義された定義の数。
18. クラス、構造体、列挙体、共用体の定義の数。

B.3.2　定義済みプロパティ

Ch では、実装に依存する C の動作の多くがさまざまなプラットフォームを対象に定義されており、最大の移植性を実現しています。実装依存プロパティと呼ばれるこれらのプロパティは、次のように定義されています。

1. データ型 *char* は 1 バイトを使用する符号付きです。
2. データ型 *short* は 2 バイトを占有します。
3. データ型 *int* は 4 バイトを占有します。
4. データ型 *long* は *int* と同じです。
5. データ型 *long long* は 64 ビットを占有します。
6. データ型 *float* は 4 バイトを占有します。
7. データ型 *double* は 8 バイトを占有します。
8. データ型 *complex* は 8 バイトを占有します (実部と虚部に 4 バイトずつ)。
9. データ型 *double complex* は 16 バイトを占有します (実部と虚部に 8 バイトずつ)。
10. `#included` でインクルードするファイルの入れ子のレベル数は 32 です。
11. 文字列の最大文字数は 5119 です。
12. 論理ソース行の最大文字数は 5119 です。
13. ファイル名の最大文字数は 5119 です。
14. 内部識別子またはマクロ名で有意な先頭からの最大文字数は 5119 です。
15. 外部識別子で有意な先頭からの最大文字数は 5119 です。
16. 外部識別子の最大数は 32767 です。
17. 1 つのブロック内に宣言するブロックスコープを持つ識別子の最大数は、32767 です。
18. switch ステートメントの case ラベルの最大数は、INT_MAX-1 です。
19. 配列の次元の最大エクステントは INT_MAX の値です。
20. 配列の次元の上限は INT_MAX-1 の値です。
21. *if-else if-else* ステートメントは入れ子にできます。各層に最大 127 個の分岐を指定できます。
22. 最大 3 つの間接ポインタを宣言できます。

B.3.3 一時的な機能

以下の機能は、C コードの Ch への移植を容易にするための一時的な機能です。今後、これらの機能は C に準拠する予定です。

1. ISO C の **sizeof** 演算子は、Ch では汎用関数として処理されます。sizeof 演算子のオペランドが複数のオペランドを持つ式である場合、または sizeof 演算の結果が式の最後のオペランドとして使用されない場合は、sizeof 演算子のオペランドを 1 対のかっこで囲む必要があります。次に例を示します。

   ```
   double a[3];
   int l = sizeof a/sizeof a[1]*4-sizeof a;
   ```

   これは次のように変更する必要があります。

   ```
   double a[3];
   int l = sizeof(a)/sizeof(a[1])*4-sizeof a;
   ```

2. '#' で始まる前処理ディレクティブは C 準拠ですが、以下の例外があります。

   (1) マクロ展開の中にそのマクロ自体が出現した場合は、再展開されます。つまり、次のコードは無効です。

   ```
   #define sqrt(x) ((x)<0? sqrt(-x) : sqrt(x))  // Error

   enum {
     _MACRO1,
   #define _MACRO1 _MACRO1   // Error
     _MACRO2
   #define _MACRO2 _MACRO2   // Error
   };
   ```

   (2) エスケープ文字'\0' は、トークンの文字列への変換時に正しく機能しない場合があります。次に例を示します。

   ```
   #define print2(a, b) printf(#a "<" #b "=%d\n", (a)<(b) )
   int main() {
     print2('\0', 10);
     print2('\\',10);
     print2('\n',10);
   }
   ```

   Ch の出力は次のとおりです。

```
'<10=1
'\'<10=0
'\n'<10=0
```

C では、出力は次のようになります。

```
'\0'<10=1
'\\'<10=0
'\n'<10=0
```

3. 型修飾子 **register** および **volatile** は無視されます。関数の引数リスト内では、型修飾子 **restrict** は無視されます。

4. 汎用関数の fscanf()、sscanf() は、割り当てステートメントの rvalue として使用できません。また、関数へのポインタの関数引数として渡すことはできません。たとえば、 Ch では次のコードは無効です。

```
int (*fp)(char *, char *, ...);
void func(int (*fp)(char *, char *, ...));
fp = fscanf;    /* Bad in Ch, ok in C */
func(fscanf);   /* Bad in Ch, ok in C */
```

5. **long double** のサイズは 8 バイトです。**long double** 型は double と同様に扱われます。**long double complex** のサイズは 16 バイトです。**long double complex** 型は **double complex** と同様に扱われます。今後、**long double** のサイズは 16 バイトに、**long double complex** のサイズは 32 バイトになる予定です。

B.3.4　Ch と C の非互換性

Ch は C のスーパーセットとなるように設計されています。ただし、以下に示す C の機能は、現在のバージョンの Ch ではサポートされていません。これは Ch 言語環境の制限ではなく、現行アプリケーションのプログラミング上のニーズを反映したものです。また、 Ch プログラミング言語の現在の実装はインタープリタとしての実装であるため、コンパイラおよびリンカ向けに設計されている C の機能は、Ch では必要ありません。サポートされていない C の機能および Ch と ISO C の非互換性を以下に示します。

1. **関数プロトタイプ**

   K&R C と呼ばれる以前の形式の関数定義がサポートされています。ただし、関数定義は、main() 関数を除いて、int を返す関数であっても Ch では型識別子で開始する必要があります。C では、型修飾子を省略すると、関数の戻り値のデータ型は既定で int です。理由は、関数の戻り値のデータ型に関する限り、Ch の型検査の方が厳密なためです。厳密な型検査は、プログラムの隠れたバグを検出するのに役立ちます。また、 Ch はスクリプトファイルであるため、関数宣言の既定の int をあいまいさなく実装することはできません。次に例を示します。

```
fun();    /* function fun() invoked or declaration of int fun()? */
fun() {
     ...
}
```

2. Ch には C よりも多くのキーワードがあります。Ch には、以下の C++のキーワードが追加されています。

   class delete new private public this

   Ch には、以下の追加キーワードが加えられています。

   ComplexInf ComplexNaN Inf NaN NULL foreach fprintf printf scanf string_t

3. C では、構造体のタグと変数は別々の名前空間を使用します。C++と同様、Ch では構造体タグを型定義して変数の名前空間に置くことにより、オブジェクト指向プログラミングを実現しています。

```
struct tag2 {
     int i;
     /* enum tag1 is local to tag2 in C++/CH, global in C */
     enum tag1 {rich, poor, thief} e;
     int f();
};
```

4. 4 次元以上の配列の初期化。しかしながら、Ch は、宣言時に初期化されない 4 次元以上の配列をサポートします。

5. 非常に複雑で頭痛のタネの（平均的な人間が理解不可能な）宣言は、Ch では無効になる場合があります。

6. C では、ポインタの間接参照の最大数は 12 です。Ch では、ポインタの多重間接参照は最大 3 個まで許可されます。つまり、単一ポインタ、二重ポインタ、および三重ポインタのみを定義できます (例: `int *ptr1`、`*ptr2`、`***ptr3`)。Ch の考え方では、2 個より多いポインタの多重間接参照は、実際のアプリケーションでは必要ないということです。複雑な四重またはそれ以上のポインタ間接参照を使用する場合、同じプログラミング上の目的を果たすための、より優れた方法が常に存在します。

7. 推奨:

```
char c, s1[] = "ABC";
c = s1[i];
```

非推奨

```
char c;
c = "ABC"[i];
```

8. IEEE 演算

```
Function              C          Ch
-----------------------------------
hypot(+/-Inf, NaN)    Inf        NaN
hypot(NaN, +/-Inf)    Inf        NaN
pow(1, NaN)           1.0        NaN
pow(NaN, +/-0.0)      1.0        NaN
-----------------------------------
```

9. C9x では、複素数を暗黙的に浮動小数点実数にキャストする場合、複素数の実部が使用されます。Ch では、複素数を暗黙的に浮動小数点実数にキャストすると、虚部がゼロでない場合、結果の実数は NaN になります。複素数の実部は汎用関数 **real()** によって取得できます。次にコード例を示します。

```
#include <complex.h>
int main() {
    double d;
    complex z;

    z = complex(1, 0);
    d = z; // d is 1 in both C9x and Ch
    z = complex(1, 2);
    d = z; // d is 1 in C9x, d is NaN in Ch
}
```

10. C では、‘c’ のような文字定数は **int** 型です。Ch では、C++と同様に、文字定数は **char** 型です。

11. ドル記号文字 `'$'` は Ch では識別子に使用することはできません。しかし、これは C では使用可能です。

B.4 C を Ch に移植する場合のヒント

Ch は C のスーパーセットとして設計されています。C に対する拡張を考慮し、プログラムを C と Ch の両方で実行できるように、C で非推奨である機能の使用は避けてください。

1. 移植性のないコード:

```
funct()
{ ...}
```

移植性のあるコード:

```
int funct()
{ ...}
```

2. 移植性のないコード:

```
funct(a, b, c)
int a;
int b;
char *c;
{...}
```

移植性のあるコード:

```
int funct(a, b, c)
int a;
int b;
char *c;
{...}
```

3. 移植性のないコード:

```
funct(int a, int b, char *c)
{...}
```

移植性のあるコード:

```
int funct(int a, int b, char *c)
{...}
```

4. インクルードするヘッダーファイルに対して、以下を追加してください。

```
#ifndef FILENAME_H
#define FILENAME_H
...
#endif
```

5. 関数の **fscanf()**、**sscanf()** を割り当てステートメントの rvalue として使用しないでください。また、関数へのポインタの関数引数として渡さないでください。たとえば、 Ch では次のコードは無効です。

```
int (*fp)(char *, ...);
void func(int (*fp)(char *, ...));
fp = fscanf;   /* Bad in Ch, ok in C */
func(fscanf);  /* Bad in Ch, ok in C */
```

6. Ch では、新しいキーワードである **ComplexInf**、**ComplexNaN**、**Inf**、**NULL**、**NaN**、**array**、**class**、**delete**、**foreach**、**fprintf**、**new**、**printf**、**private**、**public**、**scanf**、**string_t**、**this** を変数として使用しないでください。識別子がキーワードであるかどうかをチェックするには、`'which identifier'` を使用します。場合によっては、次のコードのような方法を使用して、名前空間の競合を処理できます。

```
#if defined(__cplusplus) || defined(c_plusplus) || defined(_CH_)
    int c_class;
#else
    int class;
#endif
```

7. 三重より多い間接参照のポインタを使用しないでください。

コードの移植性を確保するために、当面は次の C 機能は使用しないでください。

1. 3 次元より多くの次元を持つ配列を初期化しないでください。

2. ヘッダーファイル wchar.h に定義されている次の関数は、使用しないでください。

```
extern int fwprintf(FILE *, const wchar_t *, ...);
extern int fwscanf(FILE *, const wchar_t *, ...);
extern int swprintf(wchar_t *, size_t, const wchar_t *, ...);
extern int swscanf(const wchar_t *, const wchar_t *, ...);
extern int vfwprintf(FILE *, const wchar_t *, __va_list);
extern int vwprintf(const wchar_t *, __va_list);
extern int vswprintf(wchar_t *, size_t, const wchar_t *, __va_list);
extern int wprintf(const wchar_t *, ...);
extern int wscanf(const wchar_t *, ...);
```

3. `"string"[i]` は使用しないでください。

4. `sizeof expr` は、使用しないでください。代わりに `sizeof(expr)` を使用してください。

付録C　C++との比較

C.0.1　C++と Ch の両方にある機能

1. メンバ関数。
2. コードと宣言の混在。
3. `this->`ポインタ。
4. 参照型と参照渡し。
5. 関数形式の型変換。
6. クラス。
7. プライベートおよびパブリックのデータと関数。`public` 宣言が指定されていない限り、クラス定義のメンバはプライベートと想定されます。
8. クラス、構造体、共用体の静的メンバ。
9. const メンバ関数。
10. new 演算子と delete 演算子。
11. コンストラクタとデストラクタ。
12. ポリモーフィックな関数。
13. メンバ関数定義、静的メンバ、およびグローバル変数に使用するスコープ解決演算子`::` (ローカルな変数 g がありグローバル変数 g を参照できない場合にグローバル変数 g を示す`::g` など)。
14. 入出力 cout、cerr、cin での endl および ends の使用。
15. cout、cerr、cin、endl、および ends に対する次のような using ディレクティブ。

```
using std::cout;
using std::cin;
using std::cerr;
using std::endl;
using std::ends;
```

または、

```
using namespace std;
```

16. アリティ可変関数の引数は省略可能です。

```
int func(...); // ok in C++/Ch, bad in C.
```

C.0.2 C++のクラスに対する Ch での拡張

1. メンバ関数の内部のクラス。
2. クラスを持つ入れ子の関数。
3. 関数へのポインタ型の引数でメンバ関数を関数に渡します。

C.0.3 Ch でサポートされていない C++の機能

1. メンバ関数定義を持つクラス。

   クラスのメンバ関数の定義はクラス宣言の外側になければなりません。たとえば、次のコードは動作しません。

```
class tag {
    int i;
  public:
    tag() {
      i=0;
    }
    void f() {
      i++;
    }
};
```

   次のように変更する必要があります。

```
class tag {
    int i;
  public:
    tag();
    void f();
};
tag::tag() {
   i =0;
}
```

```
void tag::f(void) {
   i++;
}
```

2. 仮想関数と純粋仮想関数。
3. フレンド関数。
4. 継承。
5. 多重継承。
6. 保護されたデータと関数。
7. 関数のオーバーロード。
8. 演算子のオーバーロード。
9. テンプレート。
10. 例外。
11. ファイルストリーム入出力。
12. 関数の戻り値の参照型。
13. 既定の引き数値を持つ関数。
14. コピーコンストラクタ。
15. 変換関数。
16. asm キーワード。
17. 'extern C' のような結合指定子。
18. 演算子`::*` 及び `::->`。
19. 略式の初期化。

```
...
myclass ob1 = myclass(i);    // ok in  both C++ and Ch
myclass ob2 = myclass(2);    // ok in  both C++ and Ch
myclass ob3(i);              // ok in C++, bad in Ch
myclass ob4(2);              // ok in C++, bad in Ch
```

20. 式の中での宣言。次に例を示します。

```
if(char *c = malloc(8)) {    // ok in C++, bad in Ch
   /* use c here */
}.
for(int i = 0; i<10; i++) { // ok in C++, bad in Ch
   /* use i here */
}
```

21. 名前空間。

22. 実行時の型情報。

23. 新しいキャスト表記。

24. ヘッダーファイルをインクルードするとき、後ろに.*h*を付けない新しい形式。次に例を示します。

```
#include <stdio>
```

C.0.4 C++と Ch の相違点

1. Ch では、クラス型のメンバのコンストラクタが別のクラスの内部にある場合、コンストラクタはインスタンス化された時点で自動的には呼び出されません。関数が返すクラスを使用してクラスの値を割り当てることができます。次に例を示します。

```
#include <stdio.h>

class tag1 {
public:
        tag1();
        class tag1 fun();
};
tag1::tag1() {
   printf("hello from tag1 constructor\n");
}
class tag1 tag1::fun() {
  class tag1 t; /* constructor is called */

  return t;
}

class tag2 {
public:
        class tag1 document; /* constructor is NOT called */
        tag2();
```

```
};
tag2::tag2() {
   printf("hello from tag2 constructor\n");
}

int main(){
  class tag2 window;
  window.document = window.document.fun();
}
```

この例では、main() 関数の実行が始まるとまず、class tag2 window; によって tag2 クラスのオブジェクト window が作成され、このとき tag2 クラスのコンストラクタ tag2::tag2() によって文字列 hello from tag2 constructor が表示された後改行されます。tag2 クラスには tag1 クラスのオブジェクト document が含まれていますが、この段階では自動的に tag1 クラスのコンストラクタが呼ばれることはありません。main() 関数の window.document.fun() により tag1 クラスの別のオブジェクトが作成され、このときに tag1 クラスのコンストラクタ tag1::tag1() が呼ばれて文字列 hello from tag1 constructor\verb がその後の改行を伴って表示されます。なお fun() 関数が返した tag1 クラスのオブジェクトは main() 関数の window.document オブジェクトに格納されます。

付 録D　Cシェルとの比較

D.1　構文

Ch および C シェルの両方に、環境変数以外のローカルシェル変数があります。Ch と C シェルの簡単な構文比較を表 D.1に示します。表 D.1に示す Ch コマンドの意味は、対応する C シェルのコマンドと同じです。Ch の **alias**、**history**、**remvar**、および **unalias** コマンドは、対話型コマンドシェルでのみ有効です。ディレクティブ`#pragma remvar(var)`は、Ch プログラムの内部でのみ有効です。C シェルと Ch の制御フローの比較を表 D.2に示します。

表 D.1: C シェルと Ch の構文比較の一部

C シェル	Ch シェル
$#argv	_argc
$argv[*]	strjoin(" ", _argv[1], _argv[2], ..., _argv[_argc])
$argv	strjoin(" ", _argv[1], _argv[2], ..., _argv[_argc])
$*	strjoin(" ", _argv[1], _argv[2], ..., _argv[_argc])
$argv[1-n]	strjoin(" ", _argv[1], _argv[2], ..., _argv[n])
$0	_argv[0]
$argv[n]	_argv[n]
$1 $2 ... $9	_argv[1] _argv[2] ... _argv[9]
$argv[$#argv]	_argv[_argc]
set prompt = "ch> "	_prompt = "ch> "
set path = (/usr/bin /bin)	_path = "/usr/bin /bin"
umask 022	umask(022)
setenv PATH "/usr/bin /bin"	putenv("PATH=/usr/bin /bin")
echo $PATH	printf("%s\n", getenv("PATH"))
echo $PATH	echo $(getenv("PATH"))
echo $PATH	echo $PATH
echo ${PATH}	echo ${PATH}
echo $path	printf("%s\n", _path)
echo $path	echo $_path
unsetenv PATH	remenv("PATH")
printenv PATH	getenv("PATH")
unset path	remvar _path
unset path	#pragma remvar(_path)
unset i	remvar i
unset i	#pragma remvar(i)
set i =90	int i = 90
set i =91	i = 91
`cmd $var`	`cmd $var`
`cmd $ENVVAR`	`cmd $ENVVAR`
`cmd $ENVVAR`	`cmd $(getenv("ENVVAR"))`
set hostnam =`hostname`	string_t hostnam =`hostname`
set host =`hostname`	_host =`hostname`

表 D.1: C シェルと Ch の構文比較の一部 (続き)

C シェル	Ch シェル
alias rm	alias rm
alias	alias
alias rm "rm -i"	alias("rm", "rm -i")
alias f "find . -name \!:1 -print"	alias("f", "find . -name _argv[1] -print")
alias e "echo \!* \!$ \!#"	alias("e", "echo _argv[*] _argv[$] _argv[#]")
alias rm	alias("rm")
alias	alias()
unalias rm	unalias rm
unalias rm	alias("rm", NULL)
eval ls	streval("`ls`")
eval ls	system("ls")
eval setenv NAME value	streval("putenv(\"NAME=value\")")
./cmd -option	./cmd -option
/usr/bin/ls *	/usr/bin/ls *
"/path with space/cmd" option	"/path with space/cmd" option
$cmd option	$cmd option
status	_status
ls ~ *	ls ~ *
ls > output	ls > output
ls | tar -cf tarfile	ls | tar -cf tarfile
ls $PATH	ls $(getenv("PATH"))
ls $PATH	ls $PATH
ls $path	ls $_path
history = 100	_histsize = 100
history	history
!l	!l
!l -agl	!l -agl
!3	!3
!-1	!-1
!!	!!
!!	!
vi `grep -l "str1 str2" *.c`	vi `grep -l "str1 str2" *.c`
more .cshrc .login .logout	more .chrc .chlogin .chlogout
more .cshrc .login .logout	more .chsrc .chlogin .chlogout

D.2　制御フロー

表 D.2: C シェルと Ch の制御フローの比較

説明	C シェル	Ch シェル

while ループ	`while(expr)`
	`    commands`
	`end`
		`while(expr) {`
		`    commands`
		`}`

foreach ループ	`foreach token(wordList)`
	`#use $token`
	`    commands`
	`end`
		`foreach(token; wordList) {`
		`// use token`
		`    commands`
		`}`

if	`if(expr)`
	`    commands`
	`endif`
		`if(expr) {`
		`    commands`
		`}`

if-else if
-else
	`if(expr1) then`
	`    commands1`
	`else if(expr2) then`
	`    commands2`
	`else`
	`    commands3`
	`endif`
		`if(expr1) {`
		`    commands1`
		`}`
		`else if(expr2) {`
		`    commands2`
		`}`
		`else {`
		`    commands3`
		`}`

goto	`goto label` `label: statements`	`goto label` `label: statements`

switch	`switch(expr)`
	`    case pattern1:`
	`        commands1`
	`        breaksw`
	`    case pattern2:`
	`        commands2`
	`        breaksw`
	`    default:`
	`        defaultCommands`
	`endsw`
		`switch(expr) {`
		`    case pattern1:`
		`        commands1`
		`        break;`
		`    case pattern2:`
		`        commands2`
		`        break;`
		`    default:`
		`        defaultCommands`
		`        break;`
		`}`

付 録E　MATLABとの比較

表 E.1: MATLAB と Ch の比較に使用するシンボル

シンボル	データ型
x	実数型または複素数型のスカラか計算配列。
A, *Ai*, *B*, *Bi*	実数型または複素数型の計算配列。
Av, *Avi*, *Bv*, *Bvi*	実数型または複素数型の 1 次元計算配列。
R, *Ri*	実数型の計算配列。
Rv, *Rvi*	実数型の 1 次元計算配列。
I, *Ii*	整数型の計算配列。
Iv, *Ivi*	整数型の 1 次元計算配列。
C, *Ci*	char 型の配列。
Z, *Zi*	複素数型の計算配列。
Zv, *Zvi*	複素数型の 1 次元計算配列。
s	実数型または複素数型のスカラ。
z	スカラの複素数型。
r	スカラの実数型。
f	スカラの浮動小数点型。
i	スカラの整数型。
p	ポインタ型。
str	文字列。

E.1 演算子

表 E.2: MATLAB と Ch での演算子の比較

演算子	MATLAB	Ch
$\sim$	$\sim A$	$\mathbf{!}A$
	$\sim s$	$\mathbf{!}s$
$+$	$+A$	$+A$
	$+s$	$+s$
$-$	$-A$	$-A$
	$-s$	$-s$
$<$	$A<B$	$A<B$
	$A<s$	$A<r$
	$s<A$	$r<A$
$\sim=$	$A\sim=B$	$A!=B$
	$A\sim=s$	$A!=s$
	$s\sim=A$	$s!=A$
$\vert$	$A\vert B$	$A\vert\vert B$
	$A\vert s$	$A\vert\vert r$
	$s\vert A$	$r\vert\vert A$
&	$A\& B$	$A\&\& B$
	$A\& s$	$A\&\& s$
	$s\& A$	$s\&\& A$
$+$	$A+B$	$A+B$
	$A+s$	$A+s$
	$s+A$	$s+A$
	$A=A+B$	$A=A+B$
	$A=A+B$	$A+=B$
$-$	$A-B$	$A-B$
	$A-s$	$A-s$
	$s-A$	$s-A$
	$A=A-B$	$A-=B$
$*$	$A*B$	$A*B$
	$A*s$	$A*s$
	$s*A$	$s*A$
	$A=A*B$	$A=A*B$
	$A=A*B$	$A*=B$
$/$	$A=A1/B1$	$A=A1*$**inverse**$(B1)$
	A/s	A/s
	$A=A/s$	$A=A/s$
	$A=A/s$	$A/=s$

表 E.2: MATLAB と Ch での演算子の比較 (続き)

演算子	MATLAB	Ch
^	*i1^i2*	pow(*i1*,*i2*)
	s1^s2	pow(*s1*,*s2*)
	I^i	pow(*I*,*i*)
	A^i	pow(*A*,*i*)
\	*A*=*A1**B1*	*A*=**inverse**(*A1*)∗*B1*
\	*Av*=*A**Bv*'	**linsolve**(*Av*,*A*,*Bv*)
		Av=**inverse**(*A*)∗*Bv*
		llsqsolve(*Bv*, *A*,*Av*)
'	*A*'	**transpose**(*A*)
'	*Z*'	**transpose**(**conj**(*Z*))
.∗	*A*.∗*B*	*A*.∗*B*
./	*A*./*B*	*A*./*B*
./	*s*./*B*	*s*./*A*
.^	*A*.^*B*	**pow**(*A*, *B*)
.^	*A*.^*s*	**pow**(*A*, (**array double** `[n]`) *s*)
.^	*s*.^*A*	**pow**((**array double** `[n]`) *s*, *A*)
.\	*A*.*B*	(無効)

E.2 関数および定数

表 E.3: MATLAB と Ch での関数の比較

関数	MATLAB	Ch
abs	*r*=**abs**(*s*)	*r*=**abs**(*s*)
	r=**abs**(*A*)	*r*=**abs**(*A*)
	Iv=**abs**('*str*')	**array** int *Iv*[**strlen**("*str*")]= {'*s*','*t*','*r*'}
acos	*x*=**acos**(*x*)	*x*=**acos**(*x*)
acosh	*x*=**acosh**(*x*)	*x*=**acosh**(*x*)
addpath	**addpath**('/new/dir',/new/dir2')	_path=**stradd**("/new/dir1;/new/dir2;", _path)
all	*i*=**all**(*A*)	*i*=**sum**(**!***A*=0)==0
angle	*r*=**angle**(*z*)	*r*=**carg**(*z*)
any	*i*=**any**(*A*)	*i*=**sum**(*A*! =0)! =0
asin	*x*=**asin**(*x*)	*x*=**asin**(*x*)
asinh	*x*=**asinh**(*x*)	*x*=**asinh**(*x*)
atan	*x*=**atan**(*x*)	*x*=**atan**(*x*)
atan2	*r*=**atan2**(*r1*,*r2*)	*r*=**atan2**(*r1*,*r2*)
atanh	*x*=**atanh**(*x*)	*x*=**atanh**(*x*)
balance	[*A*, *B*] = **balance**(*A1*)	**balance**(*A1*, *A*, *B*)
base2dec	*i*=**base2dec**('*str*',*i2*)	*i*=**strtol**("*str*", NULL, *i2*)
bin2dec()	**bin2dec**('*01010*')	
blanks	*str*=**blank**(*n*)	" "
break	**break**	**break**
ceil	*x*=**ceil**(*x*)	*s*=**ceil**(*s*)
choly	*A*= **chol**(*A1*)	**choldecomp**(*A1*, *A*)
clear	**clear** `name`	**remvar** `name`
	clear `name`	#pragma remvar(`name`)
compan	*A*=**compan**(*Av*)	*R*=**companionmatrix**(*Rv*)
		Z=**ccompanionmatrix**(*Zv*)
cond	*r*=**cond**(*A*)	*r*=**condnum**(*A*)
condest	*r*=**condest**(*A*)	*r*=**condnum**(*A*)
conj	*x*=**conj**(*x*)	*x*=**conj**(*x*)
conv	*Av*= **conv**(*Av1*, *Av2*)	**conv**(*Av*, *Av1*, *Av1*)
conv2	*A*= **conv2**(*A1*, *A2*)	**conv2**(*A*, *A1*, *A2*)
corrcoef	*R*= **corrcoef**(*R1*)	**corrcoef**(*R*, *R1*)
corr2	*r*= **corr2**(*R1*, *R2*)	*r*= **correlation2**(*R1*, *R2*)
cos	*x*=**cos**(*x*)	*x*=**cos**(*x*)
cosh	*x*=**cosh**(*x*)	*x*=**cosh**(*x*)
cov	*R*= **cov**(*R1*)	**covariance**(*R*, *R1*)
cross	*Rv*=**cross**(*Rv1*,*Rv2*)	*Rv*=**cross**(*Rv1*,*Rv2*)

表 E.3: MATLAB と Ch での関数の比較 (続き)

関数	MATLAB	Ch
cumprod	*A*= **cumprod**(*A1*')	**cumprod**(*A*, *A1*)
	A= **cumprod**(*A1*)	**cumprod**(*A*, **transpose**(*A1*))
	Av=**cumprod**(*Av1*)	**cumprod**(*Av*, *Av1*)
cumsum	*A*= **cumsum**(*A1*')	**cumsum**(*A*, *A1*)
	A= **cumsum**(*A1*)	**cumsum**(*A*, **transpose**(*A1*))
	Av= **cumsum**(*Av1*)	**cumsum**(*Av*, *Av1*)
dec2base()		
dec2bin	*i*=**dec2bin**(*i2*)	**printf**("*%b*", *i2*)
		printf("*%10b*", *i2*)
dec2hex	*str*=**dec2hex**(*x*)	**sprintf**(*str*,"*%x*", *r*)
deconv	*A*=**deconv**(*A1*,*B1*)	**deconv**(*A*,*A1*,*B1*)
	[*A*, *B*]=**deconv**(*A1*,*B1*)	**deconv**(*A*,*A1*,*B1*,*B*)
deblank()	**deblank**('*str*')	(無効)
det	*s*=**det**(*A*)	*r*=**determinant**(*A*)
		z=**cdeterminant**(Z)
diag	*Av*=**diag**(*A*)	*Rv*=**diagonal**(*R*)
		Zv=**diagonal**(*Z*)
	A=diagb(*Av*)	*R*=**diagonalmatrix**(*Rv*)
		Z=**cdiagonalmatrix**(*Zv*)
diff	*A*=**diff**(*A*)	*Rv*=**difference**(*Rv*)
		r=**derivative**(*func*,*r*)
		R=**derivatives**(*func*,*R*)
disp	**disp**(*i*)	**printf**("%d", *i*) or **printf**(i)
	disp(*f*)	**printf**("%f", *f*) or **printf**(f)
	disp(*str*)	**printf**("%s", *str*) or **printf**(str)
	disp(*A*)	**printf**("%d", *A*) or **printf**(A)
display	**display**(*i*)	**printf**("%d", *i*) or **printf**(i)
	display(*f*)	**printf**("%f", *f*) or **printf**(f)
	display(*str*)	**printf**("%s", *str*) or **printf**(str)
	display(*A*)	**printf**("%d", *A*) or **printf**(A)
dot	*x*=**dot**(*Av1*,*Av2*)	*r*=**dot**(*Rv1*,*Rv2*)
eig	*Av*=**eig**(*A1*)	**eigensystem**(*Av*, *NULL*,*A1*)
	[*Av*, *B*]=**eig**(*A1*)	**eigensystem**(*Av*, *B*,*A1*)
	[*Av*, *B*]=**eig**(*A1*,'nobalance')	**eigensystem**(*Av*, *B*, *A1*, "nobalance")
eps	**eps**	**#include<float.h>**
		FLT_EPSILON, DBL_EPSILON

表 E.3: MATLAB と Ch での関数の比較 (続き)

関数	MATLAB	Ch
eval	**eval**('cmd')	**system**("cmd")
	eval('expr')	**streval**("expr")
	eval(*try*,*catch*)	
eye	*R*=**eye**(*i*)	*R*=**identitymatrix**(*i*)
exp	*x*=**exp**(*x*)	*x*=**exp**(*x*)
expm	*A*= **expm**(*A1*)	**expm**(*A*,*A1*)
fclose	**fclose**	**fclose**() を参照
feval	**feval**('*fun*')	**streval**("*fun*")
feof	**feof**	**feof**() を参照
ferror	**ferror**	**perror**() および他の I/O 関数を参照
fft	*Av*= **fft**(*Av1*)	**fft**(*Av*, *Av1*)
	Av= **fft**(*Av1*, *i*)	*A*= **fft**(*A1*, *i*)
fft2	*A*=**fft2**(*A1*)	**fft**(*A*, *A1*)
	A= **fft**(*A1*, *i1*, *i2*)	*Iv*[0]=*i1*, *Iv*[*1*] =*i2*, *A*= **fft**(*A1*, *Iv*)
fftn	*A*=**fftn**(*A1*)	**fft**(*A*, *A1*) /* 3D only */
	A= **fftn**(*A1*, *i*)	*Iv*[0]=*Iv*[*1*]=*Iv*[*2*]=*i*, **fft**(*A*,*A1*,*Iv*) /* 3D only */
fftshift		
fgetl	*str*= **fgetl**(*fid*)	*i*=**strlen**(*str*)
		getline(*fid*, *str*, *i*)
fgets	**fgets**	**fgets**() を参照
fix	*i*=**fix**(*r*)	*i*=*r*
filter	*Av*= **filter**(*Bv1*, *Bv2*, *Av1*)	**filter**(*Bv1*, *Bv2*, *Av1*, *Av*)
filter2	*A*= **filter2**(*A1*, *B1*)	**filter2**(*A*, *A1*, *B1*)
finite	*i*=**finite**(*s*)	*i*=**isfinite**(*s*)
	I=**finite**(*A*)	**fevalarray**(*I*,**isfinite**,*R*);
find()	*i*=**find**(*x*)	
	[*r*,*c*]=**find**(*x*)	
	I= **find**(*A*)	*i*=**findvalue**(*I*, *A*) /* i is # of values found */
findstr()	*i*=**findstr**('*str1*', '*str2*')	*p*=**strstr**(*str1*, *str2*)
fliplr	*A*= **fliplr**(*A1*)	**fliplr**(*A*, *A1*)
flipud	*A*= **flipud**(*A1*)	**flipud**(*A*, *A1*)
floor	**floor**(*x*)	**floor**(*x*)
flops	**flops**	(無効)
fmin	*r*= **fmin**('*fun*', *r1*, *r2*)	**fminimum**(*r*3, *r*, *fun*, *r1*, *r2*)
fmins	*Rv*= **fmins**('*fun*', *Rv1*)	**fminimums**(*r*3, *Rv*, *fun*, *Rv1*)
fplot	**fplot**('*fun*', [*r1r2*])	**fplotxy**(*fun*,*r1*,*r2*)
		fplotxyz()
		CPlot:MemberFunctions()

表 E.3: MATLAB と Ch での関数の比較 (続き)

関数	MATLAB	Ch
fprintf	**fprintf**()	**fprintf**() を参照
fread	**fread**()	**fread**() を参照
frewind	**frewind**()	**frewind**() を参照
fscanf	**fscanf**()	**fscanf**() を参照
fseek	**fseek**()	**fseek**() を参照
ftell	**ftell**()	**ftell**() を参照
funm	*R*= **funm**(*R1*,'*fun*')	**funm**(*R*, *fun*, *R1*)
	Z= **funm**(*Z*,'*fun*')	**cfunm**(*Z*, /* complex */ cfun, *Z1*)
fwrite	**fwrite**()	**fwrite**() を参照
fsolve	*R*= **fsolve**('*fun*', *Ri*)	**fsolve**(*R*, *fun*, *Ri*)
fzero	*r*= **fzero**('*fun*', *r1*)	**fzero**(*r*, *fun*, *r1*)
gallery	*A*=**gallery**(*name*,*arg1*)	*A*=**specialmatrix**(Name,*arg1*)
	A=**gallery**(*name*,*arg1*,*arg2*)	*A*=**specialmatrix**(*Name*,*arg1*,*arg2*)
	A=**gallery**(*name*,*arg1*,*arg2*,*arg3*)	*A*=**specialmatrix**(*Name*,*arg1*,*arg2*,*arg3*)
	A=**gallery**('*caychy*',*Av1*)	**specialmatrix**("*Cauchy*",*Av1*)
	A=**gallery**('*caychy*',*Av1*,*Av2*)	**specialmatrix**("*Cauchy*",*Av1*,*Av2*)
	A=**gallery**('*chebvand*',*Av1*)	**specialmatrix**("*ChebyshevVandemonde*",*Av1*)
	A=**gallery**('*chebvand*',*i*,*Av1*)	**specialmatrix**("*ChebyshevVandemonde*",*Av1*,*i*)
	A=**gallery**('*chow*',*i*)	**specialmatrix**("*Chow*",*i*)
	A=**gallery**('*chow*',*i*,*r1*)	**specialmatrix**("*Chow*",*i*,*r1*)
	A=**gallery**('*chow*',*i*,*r1*,*r2*)	**specialmatrix**("*Chow*",*i*,*r1*,*r2*)
	A=**gallery**('*circul*',*Av1*)	**specialmatrix**("*Circul*",*Av1*)
	A=**gallery**('*clement*',*i1*,*i2*)	**specialmatrix**("*Clement*",*i1*,*i2*)
		specialmatrix("*DenavitHartenberg*",*r1*,*r2*,*r3*,*r*4)
		specialmatrix("*DenavitHartenberg2*",*r1*,*r2*,*r3*,*r*4)
	A=**gallery**('*dramadah*',*i1*)	**specialmatrix**("*Dramadah*",*i1*)
	A=**gallery**('*dramadah*',*i1*,*i2*)	**specialmatrix**("*Dramadah*",*i1*,*i2*)
	A=**gallery**('*fiedler*',*Av1*)	**specialmatrix**("*Fiedler*",*Av1*)
	A=**gallery**('*frank*',*i1*)	**specialmatrix**("Frank",*i1*)
	A=**gallery**('*frank*',*i1*,*i2*)	**specialmatrix**("*Frank*",*i1*,*i2*)
	A=**gallery**('*gearmat*',*i1*)	**specialmatrix**("Gear",*i1*)
	A=**gallery**('*wilk*',*i1*)	**specialmatrix**("*Wilkinson*",*i1*)
	AV=**gallery**('*house*',*Av1*)	**householdermatrix**(*Av1*,*Av*)
	[*AV*,*r*]=**gallery**('*house*',*Av1*)	**householdermatrix**(*Av1*,*Av*,*r*)
gcd()	*I*= **gcd**(*I1*, *I2*)	**gcd**(*I1*, *I2*, *I*)
	[*I*,*I*3,*I*4]=**gcd**(*I1*,*I2*)	**gcd**(*I1*, *I2*, *I*, *I*3, *I*4)
hadamard	*A*=**hadamard**(*i1*)	**specialmatrix**("*Hadamard*",*i1*)

表 E.3: MATLAB と Ch での関数の比較 (続き)

関数	MATLAB	Ch
isglobal	*i*=**isglobal**(*x*)	**#include**<chshell.h >
		isvar("*x*")==**CH_SYSTEMVAR**
ishold	*i*=**ishold**	(無効)
isieee	*i*=**isieee**	(無効)
isinf	*i*=**isinf**(*s*)	*i*=**isinf**(*s*)
	I=**isinf**(*A*)	**fevalarray**(*I*,**isinf**,*R*);
isletter	*i*=**isletter**(*s*)	*i*=**isalpha**(*s*)
	I=**isletter**(*A*)	**fevalarray**(*I*,**isalpha**,*R*);
isnan	*i*=**isnan**(*s*)	*i*=**isnan**(*s*)
	I=**isnan**(*A*)	**fevalarray**(*I*,**isnan**,*R*);
isreal	*i*=**isreal**(*s*)	*i*=**elementtype**(*x*) != **elementtype**(**complex**) &&
		elementtype(*x*) != **elementtype**(**double complex**)
isspace	*i*=**isspace**(*s*)	*i*=**isspace**(*s*)
	I=**isspace**(*A*)	**fevalarray**(*I*,**isspace**,*R*);
issparse	*i*=**issparse**(*x*)	
isstr	*i*=**isstr**(*x*)	*i*=**elementtype**(*x*) == **elementtype**(*string_t*)
isstudent	*i*=**isstudent**	*i*=**isstudent**()
isunix	*i*=**isunix**	**#ifndef _WIN32_**
isvms	*i*=**isvms**	(無効)
invhilb()		
j	**j**	**#include**<**complex.h**>
		I
lasterr	**lasterr**('')	**perror**()、**strerror**() を参照
lcm()	*I*= **lcm**(*I1*,*I2*)	**lcm**(*I*,*I1*,*I2*)
linspace	*x*=**linspace**(*first*,*last*,*n*)	**lindata**(*first*,*last*, *x*)
loglog	*x*=**loglog**(*x*,*y*)	plot.loglog(*x*,*y*)
loglog	*x*=**loglog**(*x*,*y*)	plot.data2D(*x*,*y*)
		plot.scaleType(PLOT_AXIS_X, PLOT_SCALETYPE_LOG)
		plot.scaleType(PLOT_AXIS_Y, PLOT_SCALETYPE_LOG)
logspace	*x*=**logspace**(*first*,*last*,*n*)	**logdata**(*first*,*last*,*x*)
log	*x*=**log**(*x*)	*x*=**log**(*x*)
log10	*x*=**log10**(*x*)	*x*=**log10**(*x*)
log2	*x*=**log2**(*x*)	*x*=**log**(*x*)/**log**(*2*)
	r=**log2**(*r*)	*r*=**log2**(*r*)
logm	*A*=**logm**(*A1*)	**logm**(*A*,*A1*)
lower	*str*=**lower**('*str*')	**tolower**() を参照
lscov	*R*= **lscov**(*A1*,*R1*,*A2*)	**llsqcovsolve**(*R*, *A1*,*R1*,*A2*)
	[*R*,*R2*]=**lscov**(*A1*,*R1*,*A2*)	**llsqcovsolve**(*R*, *A1*,*R1*,*A2*,*R2*)

表 E.3: MATLAB と Ch での関数の比較 (続き)

関数	MATLAB	Ch
lu	[*R1*,*R2*]=**lu**(*A*)	**ludecomp**(*A*,*R1*,*R2*)
	[*R1*,*R2*,*I*]=**lu**(*A*)	**ludecomp**(*A*,*R1*,*R2*,*I*)
magic()	*A*=**magic**(*i1*)	**specialmatrix**(”*Magic*”,*i1*)
	Zv= **mean**(*Z*)	**cmean**(*Z*, *Zv*)
max	*r*=**max**(*s1*,*s2*)	*r*=**max**(*r1*,*r2*)
	r=**max**(*Av*)	*r*=**max**(*R*)
	Rv=**max**(*A*)	*Rv*=**maxv**(*R*)
		Rv=**transpose**(**maxv**(**transpose**(*R*))
min	*r*=**min**(*s1*,*s2*)	*r*=**min**(*r1*,*r2*)
	r=**min**(*Av*)	*r*=**min**(*R*)
	Rv=**min**(*A*)	*Rv*=**minv**(*R*)
		Rv=**transpose**(**minv**(**transpose**(*R*))
mean	*r*=**mean**(*R*)	*r*=**mean**(*R*)
	Rv=**mean**(*R*)	*r*=**mean**(*R*, *Rv*)
		mean(**transpose**(*R*), *Rv*)
	z= **mean**(*Z*)	*z*= **cmean**(*Z*)
median	*r*=**median**(*R*)	*r*=**median**(*R*)
	Rv=**median**(*R*)	*r*=**median**(*R*, *Rv*)
		median(**transpose**(*R*), *Rv*)
mod	*i*=**mod**(*i1*,*i2*)	*i*=*i1*%*i2*
	r=**mod**(*r1*,*r2*)	*r*=**fmod**(*r1*,*r2*)
NaN	**NaN**	**NaN**
nargin	**nargin**	(無効)
nargout	**nargout**	(無効)
nextpow2	*i*=**nextpow2**(*r*)	*i*=**ceil**(**log2**(*r*))
	i=**nextpow2**(*z*)	*i*=**ceil**(**log2**(**abs**(*z*)))
nnls()	*R*=**nnls**(*A1*,*R1*)	**llsqnonnegsolve**(*R*, *A1*,*R1*)
	R=**nnls**(*A1*,*R1*,*r*)	**llsqnonnegsolve**(*R*, *A1*,*R1*,*R*)
	[*R*, *R2*]=**nnls**(*A1*,*R1*)	**llsqnonnegsolve**(*R*, *A1*,*R1*,0.0, *R2*)
	[*R*, *R2*]=**nnls**(*A1*,*R1*,*r*)	**llsqnonnegsolve**(*R*, *A1*,*R1*,*r*, *R2*)
norm	**norm**(*A*)	**norm**(*A*,”*2*”)
	norm(*A*,*1*)	**norm**(*A*,”*1*”)
	norm(*A*,*2*)	**norm**(*A*,”*2*”)
	norm(*A*,*inf*)	**norm**(*A*,”*i*”)
	norm(*A*,’*fro*’)	**norm**(*A*,”*f*”)
	norm(*Av*)	**norm**(*Av*,”*2*”)
	norm(*Av*,*inf*)	**norm**(*Av*,”*i*”)
	norm(*Av*,-*inf*)	**norm**(*Av*,”-*i*”)
	norm(*Av*,p)	**norm**(*Av*,”*p*”)

表 E.3: MATLAB と Ch での関数の比較 (続き)

関数	MATLAB	Ch
numtwostr	*str*=**numtwostr**(*s*)	**sprintf**(*str*,"%", *s*)
null	*A*=**null**(*Ai*)	**nullspace**(*A*, *Ai*)
ones	**ones**(*i1*)	(**array int a**[*i1*][*i1*])*1*
	ones(*i1*,*i2*)	(**array int a**[i*i1*][*i2*])*1*
		(**array int a**[dim[*0*]][dim[*1*]])*1*
ode23	[*Rv1*,*Rv2*]= **ode23**('*fun*',*r1*,*r2*,*Rv*)	**oderungekutta**(*Rv1*,*Rv2*, NULL, *fun*,*r1*,*r2*,*Rv*)
ode45	[*Rv1*,*Rv2*]= **ode45**('*fun*',*r1*,*r2*,*Rv*)	**oderungekutta**(*Rv1*,*Rv2*, NULL, *fun*,*r1*,*r2*,*Rv*)
orth	*A*= **orth**(*Ai*)	**orthonormalbase**(*A*, *Ai*)
pascal()	*A*=**pascal**(*i1*)	**specialmatrix**("*Pascal*",*i1*)
	A=**pascal**(*i1*,*i2*)	**specialmatrix**("*Pascal*",*i1*,*i2*)
pi	**pi**	**#include<math.h>**
		M_PI
pinv	*A1*=**pinv**(*A*)	*R1*=**pinverse**(*R*)
poly	*Av*=**poly**(*A*)	**charpolycoef**(*Av*,*A*)
	Bv=**poly**(*Av*)	**polycoef**(*Bv*,*Av*)
polyder	*Av*= **polyder**(*Bv*)	**polyder**(*Av*, *Bv*)
polyder2	*Av*= **polyder**(*Av1*, *Bv1*)	**polyder2**(*Av*, NULL, *Av1*, *Bv1*)
	[*Av*,*Bv*] = **polyder**(*Av1*,*Bv1*)	**polyder2**(*Av*,*Bv*,*Av1*,*Bv1*)
polyfit	*Rv*=**polyfit**(*Rv1*, *Rv2*, *i*)	**polyfit**(*Rv*, *Rv1*, *Rv2*)
polyval	*r*= **polyval**(*Rv1*, *r1*)	*r*= **polyeval**(*Rv1*, *r1*)
	r= **polyval**(*Rv1*,*r1*,*Rv*)	*r*= **polyeval**(*Rv1*, *r1*, *Rv*)
	z= **polyval**(*Av1*, *s*)	*z*= **cpolyeval**(*Av1*, *s*)
	z= **polyval**(*Av1*, *s*, *Av*)	z = **cpolyeval**(*Av1*, *s*, *Av*)
	Av= **polyval**(*Bv*, *Av1*)	**polyevalarray**(*Av*, *Bv*, *Av1*)
	A=**polyval**(*Av*,*A1*)	**polyevalm**(*A*,*Av*,*A1*)
polyvalm()	*A*=**polyvalm**(*Av*,*A1*)	**polyevalm**(*A*,*Av*,*A1*)
plot	**plot**(Rva,Rvb)	**plotxy**(*Rv1*,*Rv2*)
		plotxyf(*file*)
		CPlot:MemberFunctions()
plot3	**plot**3(*Rv1*,*Rv2*,*Rv*3)	**plotxyz**(*Rv1*,*Rv2*,*Rv*3)
		plotxyzf(*file*)
		CPlot:MemberFunctions()

表 E.3: MATLAB と Ch での関数の比較 (続き)

関数	MATLAB	Ch
prod	*s*=**prod**(*A*)	*r*=**product**(*R*)
		z=**cproduct**(Z)
	Av=**prod**(*A*)	*r*=**product**(*R*, *Rv*)
	Av=**prod**(*A*, 1)	*r*=**product**(*R*, *Rv*)
	Av=**prod**(*A*, 2)	*r*=**product**(**transpose**(*R*), *Rv*)
		z=**cproduct**(*Z*,*Zv*)
		product(**transpose**(*R*), *Rv*)
		cproduct(**transpose**(*Z*), *Zv*)
quad	*r*=**quad**('*fun*',*r1*,*r2*)	*r*=**integral1**(*fun*,*r1*,*r2*)
quadeight	*r*=**quadeight**('*fun*',*r1*,*r2*)	*r*=**integral1**(*fun*,*r1*,*r2*)
qr	[*A1*, *A2*] = **qr**(*A*)	**qrdecomp**(*A*, *A1*, *A2*)
	[*A1*, *A2*, *A3*] = **qr**(*A*)	
	[*A1*, *A2*]=**qr**(*A*)	**qrdecomp**(*A*,*A1*,*A2*)
	[*A1*,*A2*]=**qr**(*A*,**zero**)	**qrdecomp**(*A*,*A1*,*A2*)
qrdelete	[*A1*, **itBa**] = **qrdelete**(*A*, *B*, *i2*)	**qrdelete**(*A1*, **itBa**, *A*, *B*, *i2*)
qrinsert	[*A1*, **itBa**] = **qrdelete**(*A*, *B*, *i2*, *Av*)	**qrdelete**(*A1*, **itBa**, *A*, *B*, *i2*, *Av*)
rank	*i*= **rank**(*A*)	*i*= **rank**(*A*)
	i= **rank**(*A*, *r*)	*i*= **rank**(*A*)
real	*r*=**real**(*s*)	*r*=**real**(*s*)
	R=**real**(*A*)	*R*=**real**(*A*)
realmax	**realmax**	**#include<float.h>**
		FLT_MAX, DBL_MAX
realmin	**realmin**	**#include<float.h>**
		FLT_MIN, DBL_MIN
rem	*r*=**rem**(*r1*,*r2*)	*r*=**fmod**(*r1*,*r2*)
		r=**remainder**(*r1*,*r2*)+*r2*
residue	[*Av*,*Bv*,Rv]=**residue**(*Rv1*,*Rv2*)	**residue**(*Rv1*, *Rv2*, *Av*, *Bv*, *Rv*)
round	*i*=**round**(*r*)	*i*=**round**(*r*)
roots	*Av*=**roots**(*Bv*)	**roots**(*Av*,*Bv*)
rosser	*A*=**rosser**()	**specialmatrix**("*Rosser*")
rot90(*A*)		
	A= **rot90**(*A1*)	**rot90**(*A*,*A1*)
	A= **rot90**(*A1*,*i*)	**rot90**(*A*,*A1*,*i*)
rand	*r*=**rand**()	*r*=**urand**()
randn	*R*=**rand**(*i1*, *i2*)	**urand**(*R*)
rcond	*r*= **rcond**(*A*)	*r*= **rcondnum**(*A*)
reshape	**reshape**(*A*, *m*, *n*)	(**array** type [m][n])*A*
rsf2csf	[*A1*, **itBa**] = **rsf2csf**(*A*, *B*)	**rsf2csf**(*A1*, **itBa**, *A*, *B*)

表 E.3: MATLAB と Ch での関数の比較 (続き)

関数	MATLAB	Ch
schur	*A1*= **schur**(*A*)	**schurdecomp**(*A*, *A1*, *NULL*)
	[*A1*, *A2*] = **schur**(*A*)	**schurdecomp**(*A*, *A1*, *A2*)
semilogx	*x*=**semilogx**(*x*,*y*)	plot.semilogx(*x*,*y*)
semilogx	*x*=**semilogx**(*x*,*y*)	plot.data2D(*x*,*y*)
		plot.scaleType(PLOT_AXIS_X,
		PLOT_SCALETYPE_LOG)
semilogy	*x*=**semilogy**(*x*,*y*)	plot.semilogy(*x*,*y*)
semilogy	*x*=**semilogy**(*x*,*y*)	plot.data2D(*x*,*y*)
		plot.scaleType(PLOT_AXIS_Y,
		PLOT_SCALETYPE_LOG)
sign	*i*=**sign**(*r*)	*i*=**sign**(*r*)
	z=**sign**(*z*)	*z*=*z*/**abs**(*z*)
sqrt	*x*=**sqrt**(*x*)	*x*=**sqrt**(*x*)
sqrtm	*A*= **sqrtm**(*A1*)	**sqrtm**(*A*,*A1*)
sort	*Av*=**sort**(*Av1*)	**sort**(*Av*, *Av1*)
		sort(*Av*, *Av1*, ”array”)
	A=**sort**(*A1*)	**sort**(*A*, *A1*, ”column”)
	A=**sortrows**(*A1*)	**sort**(*A*, *A1*, ”row”)
	[*Av*, I]=**sort**(*Av1*)	**sort**(*Av*, *Av1*, ”array”, I)
		sort(*Av*, *Av1*, NULL, I)
	[*A*, *I*]=**sort**(*A1*)	**sort**(*A*, *A1*, ”column”, I)
	[*A*, *I*]=**sortrows**(*A1*)	**sort**(*A*, *A1*, ”rows”, I)
spline	*Ri*= **spline**(*R1*, *R2*, *Ri1*)	**interp1**(*Ri*, *R1i*, *R2i*, ”*spline*”)
	ri= **spline**(*R1*, *R2*, *r*)	**CSpline**::**Interp**(r)
	Ri= **spline**(*R1*, *R2*, *Ri1*)	**CSpline**::**Interpm**(*Ri1*, *Ri*)
sprintf	**sprintf**()	**sprintf**() を参照
sscanf	**sscanf**()	**sscanf**() を参照
std	*r*=**std**(*R*)	*r*=**std**(*R*)
	Rv=**std**(*R*)	*r*=**std**(R, Rv)
		std(**transpose**(*R*), *Rv*)
str2mat	C=**str2mat**(’*str1*’, ’*str2*’, ...)	**str2mat**(*C*, *str1*, *str2*)
str2num	*i*=**str2num**(’*str*’)	*i*=**atol**(*str*)、**strtod**() を参照
strcmp	**strcmp**(’*str1*’,’*str2*’)	!**strcmp**(”*str1*”,”*str2*”)
strrep	*str*= **strrep**(’*str1*’, ’*str2*’, ’*str3*’)	*str*=**strrep**(*str1*, *str2*, *str3*)
strtok	**strtok**	**strtok**()、**strtok_r**() を参照
subplot	**subplot**()	**CPlot**::**subplot**()
		CPlot::**getSubplot**()

表 E.3: MATLAB と Ch での関数の比較 (続き)

関数	MATLAB	Ch
sum	*r*=**sum**(*A*)	*r*=**sum**(*R*)
		z=**csum**(*Z*)
	Av=**sum**(*A*)	*r*=**sum**(*R*, *Rv*)
		z=**csum**(*Z*,*Zv*)
		sum(**transpose**(*R*), *Rv*)
		csum(**transpose**(*Z*), *Zv*)
svd	*R*= **svd**(*A*)	**svd**(*A*, *R*, *NULL*, *NULL*)
	[*A1*, *R*, *A2*] = **svd**(*A*))	**svd**(*A*, *R*, *A1*, *A2*)
tan	*x*=**tan**(*x*)	*x*=**tan**(*x*)
tanh	*x*=**tanh**(*x*)	*x*=**tanh**(*x*)
toeplititz	*A*=**toeplititz**(*Av1*)	**specialmatrix**("*Toeplititz*",*Av1*)
	A=**toeplititz**(*Av1*,*Av2*)	**specialmatrix**("*Toeplititz*",*Av1*,*Av2*)
trace	*s*=**trace**(*A*)	*r*=**trace**(*R*)
		z=**ctrace**(Z)
trapz	*r*=**trapz**(Rva,Rvb)	**integral1**() を参照
tril	*A1*=**tril**(*A*)	*A1*=**triangularmatrix**("*lower*",*A*)
	Z1=**tril**(Z)	*Z1*=**ctriangularmatrix**("*lower*",*Z*)
	A1=**tril**(*A*,i)	*A1*=**triangularmatrix**("*lower*",*A*,*i*)
	Z1=**tril**(Z,i)	*Z1*=**ctriangularmatrix**("*lower*",*Z*,*i*)
triu	*A1*=**triu**(*A*)	*A1*=**triangularmatrix**("*upper*",*A*)
	Z1=**triu**(Z)	*Z1*=**ctriangularmatrix**("*upper*",*Z*)
	A1=**triu**(*A*,i)	*A1*=**triangularmatrix**("*upper*",*A*,*i*)
	Z1=**triu**(*Z*,*i*)	*Z1*=**ctriangularmatrix**("*upper*",*Z*,*i*)
vander	*R*=**vander**(*Rv*)	*R*=**vandermatrix**(*Rv*)
	A=**vander**(*Av*)	**specialmatrix**("*Vandermonde*", *Av*)
unwrap()	*A*=**unwrap**(*A1*)	**unwrap**(*A*,*A1*)
	A=**unwrap**(*A1*,r)	**unwrap**(*A*,*A1*,*r*)
upper	*str*=**upper**('str')	**toupper**() を参照
who	**who**	**showvar**
xcorr	*Av*= **xcorr**(*Av1*, *Av2*)	**xcorr**(*Av*, *Av1*, *Av2*)
xor	**xor**(*A*,*B*)	*A* ^^ *B*
	xor(*A*,*s*)	*A* ^^ *r*
	xor(s,*A*)	*r* ^^ *A*
zeros	**zeros**(*i1*)	(**array int**[*i1*][*i1*]) *0*
	zeros(*i1*,*i2*)	(**array int**[*i1*][*i2*]) *0*
	zeros(**size**(*A*))	**array int** dim[*2*]=shape(*A*);
		(**array int** [dim[*0*]][dim[*1*]]) *0*
2D/3D プロット関数	**CPlot** クラスを参照	

E.3　制御フロー

表 E.4: MATLAB と Ch の制御フローの比較

説明	MATLAB	Ch シェル

for-loop	for i=n1:n2:n3
	    commands
	end
	
	for i=n1:n3
	    commands
	end
		for (i=n1; i<=n3; i+=n2) {
		    commands
		}
		
		for(i=n1; i<=n3; i++) {
		    commands
		}

while-loop	while expr
	    commands
	end
		while(expr) {
		    commands
		}

if	if expr
	    commands
	end
		if (expr) {
		    commands
		}

if-else	if expr1
	    commands1
	
	else
	    commands2
	end
		if (expr1) {
		    commands1
		}
		else {
		    commands2
		}

if-else if-else	if expr1
	    commands1
	
	elseif expr2
	    commands2
	
	else
	    commands3
	end
		if (expr1) {
		    commands1
		}
		else if(expr2) {
		    commands2
		}
		else {
		    commands3
		}

付 録F　Fortranとの比較

Chでは、科学の分野での数値計算に関するCとFORTRANの差異を補うために、多くの機能が追加されています。Chの参照、複素数、汎用関数、および可変長配列はFORTRANと同様です。たとえば、Chの参照の言語的な特徴は、FORTRANのequivalenceステートメント、サブルーチン、および関数に密接に関連しています。以前にFORTRANの経験があるユーザーは、簡単にChのプログラミングパラダイムに適応できます。この付録では、FORTRANコードのChへの移植に関連する問題について説明します。

F.1　ChでのZ参照とFORTRANでのequivalence

Chでの参照の言語的な特徴は、equivalenceステートメントに密接に関連しています。次のようなFORTRANのequivalenceステートメントがあるとします。

```
real f1, f2
equivalence (f1, f2), (i, j, k)
```

これは次のようにChで記述できます。

```
float f1, &f2 = f1
int i, &j = i, &k = i
```

ここで、float変数である`f1`と`f2`は同じメモリ領域を共有し、int変数である`i`、`j`、および`k`は同じ領域を共有します。FORTRANでの2つの配列のequivalenceは、Chではポインタによって実現できます。たとえば、次のようなFORTRANのコードがあるとします。

```
dimension A(10), B(10), C(20, 20), D(20, 20)
equivalence (A[1], B[1]), (C, D)
```

これは次のようにChに移植できます。

```
float A[10], *B, C[20,20], (*D)[20]
B = A; D = C;
```

ここで、`B`はfloatへのポインタであり、`D`は20個のfloatで構成される配列へのポインタです。FORTRANでの単一の変数と配列の1つの要素のequivalenceをChに移植する場合は、より複雑になります。たとえば、次のようなFORTRANのコードがあるとします。

```
int a[2][3], b[4][5];
a[1][2] = 5;
void funct(int (*c)[3], d[:][:])
{
  d[2][3] = c[1][2];
}
funct(a,b);
printf("b[2][3] = %d \n", b[2][3]); // output: b[2][3] = 5
```

プログラム F.1: Ch で参照によって関数に配列を渡す処理

```
dimension A(10)
equivalence (A[3], f)
A[3] = 5;
```

これは次のように Ch に移植できます。

```
float A[10], *f
f = &A[2]
A[2] = 5            // *f = 5
```

FORTRAN ではデータ型の異なる変数を equivalence ステートメント内に指定できますが、FORTRAN 77 規格にはこのような種類の equivalence の一貫した処理方法がありません。

F.2 Ch および FORTRAN での参照呼び出し

Ch では、値または参照によって関数を呼び出すことができます。FORTRAN 77 の参照呼び出しに関する多くの制約が、Ch では緩和されています。たとえば、FORTRAN では、サブルーチンがサブルーチン自身を呼び出すこと、または関数の内部でその関数自身を使用することは許可されません。つまり、再帰的なサブルーチン呼び出しは FORTRAN では許可されません。したがって、サブルーチン呼び出しを入れ子または再帰的な入れ子にすることもできません。FORTRAN では、関数の実引数のデータ型は、関数の仮引数のデータ型と同じでなければなりません。Ch では、関数の実引数のデータ型が、関数内での参照の対応する仮引数のデータ型と異なっていてもかまいません。FORTRAN では、サブルーチンの内部でサブルーチンの引数を lvalue として使用する場合、呼び出し元サブルーチンの実引数は変数でなければなりません。Ch では、実引数が lvalue でない場合でも関数の内部で参照変数を lvalue として使用できますが、FORTRAN では使用できません。

引数を使用して配列を関数に渡すと、呼び出し元関数の配列用のメモリ領域が、呼び出された関数で使用されることに注意してください。つまり、プログラム F.1に示すように、Ch では配列が参照によって渡される一方、

関数 `funct()` の引数の変数 `c` と `d` はそれぞれ、int 型の 3 つの要素を持つ配列へのポインタであり、形状引継ぎ配列へのポインタです。

```
      SUBROUTINE FUNCT(A, M, N, R)
        INTEGER M, N
        COMPLEX A(M, N), R
        INTEGER I, J
        DO 10 I = 1, M
          DO 10 J = 1, N
            A(I,J) = R*R + 3
 10         A(I,J) = SIN(A(I,J)/R)
        R = 50
      END

      COMPLEX ZA(5,6), Z
      Z = CMPLX(1,2)
      CALL FUNCT(ZA, 5, 6, Z)
      STOP
      END
```

プログラム F.2: FORTRAN プログラム

Ch ではFORTRAN 77 のすべてのプログラミング機能がカプセル化されているので、FORTRAN のサブルーチンと関数を Ch の関数に移植するのはそれほど難しくはありません。たとえば、プログラム F.2に示す FORTRAN プログラムは、プログラムの機能を変えることなく、プログラム F.3に示す Ch プログラムに移植できます。

プログラム F.3では、関数 `FUNCT()` の内部にある複素配列 `A` に対して、複素配列 `ZA` の実引数から 5x6 という形状が引き継がれます。整数値 5 と 6 が、`M` および `N` という参照の引数に渡されます。複素変数 `Z` は関数に参照渡しされます。ブロック構造を持つ入れ子の do ループが関数の内部で使用されます。他の多くの数学関数と同様に、Ch の関数 `sin()` は多態性の関数であり、入力引数のデータ型に従って結果を計算します。この例では、関数 `sin()` によって複素数の値が生成されます。

このような Ch の新機能は、以前に FORTRAN を経験したユーザーがすぐに Ch のプログラムを書けるように、C と FORTRAN の差異を補うことを意図したものです。以前に FORTRAN を経験したユーザーは、複雑な応用工学の問題を解決する Ch プログラムを簡単に作成できることが実績によって示されています。配列構文や形状引継ぎ配列など、Fortran 90 の多くのプログラミング機能が Ch に組み入まれていることに注目してください。Fortran 90 は、FORTRAN 77 に比べて非常に複雑です。したがって、Fortran 90 の Ch への移植は Ch の設計目標に含まれていません。

```
void funct(complex a[:][:], int &m, &n, complex &r) {
  int i, j;
  for(i=0; i<m; i++)
    for(j=0; j<n; j++) {
      a[i][j] = r*r + 3;
      a[i][j] = sin(a[i][j]/r);
    }
  r = 50;
}

complex za[5][6], z;
z = complex(1,2);
funct(za, 5, 6, z);
```

プログラム F.3: FORTRAN プログラムから移植した Ch プログラム

付録G　Chで一般的に使用される移植性のあるシェルコマンドの概要

この付録では、Unix と Windows の両方の Ch で一般的に使用される、異なるプラットフォームに移植可能なコマンドの概要を示します。以下のコマンドはカテゴリ別に記載しています。これらのコマンドは、Ch で移植性のあるシェルプログラミングを行うために非常に有用です。

G.1　ファイルシステム

cd	作業ディレクトリを変更
chgrp	ファイルのグループまたはライブラリを変更
chmod	ファイルのアクセス許可を変更
chown	ファイルまたはライブラリの所有者を変更
cp	ファイルまたはディレクトリをコピー
dd	ファイルに対する入出力操作および変換の実行を禁止するときに使用
df	disk free (空きディスク) の意味。システムからアクセスできるドライブの使用可能な領域と使用中の領域を表示
du	階層構造のディレクトリ用に使用されているディスク領域の量を表示
find	名前、変更日時などの特定の条件に一致するファイルまたはディレクトリを検索
ln	ハードリンクまたはシンボリックリンクを設定
ls	ディレクトリの内容を一覧表示
mkdir	新しいディレクトリを作成
mv	ファイルの名前を変更またはファイルを他のディレクトリに移動
pwd	作業ディレクトリの名前を出力
rm	ファイルまたはディレクトリを削除
rmdir	ディレクトリを削除
touch	新しい空のファイルを作成、または既存のファイルの変更日時を更新
which	コマンド、関数、およびヘッダーファイルの絶対パスを表示

G.2　バイナリファイル

ar	アーカイブ管理ユーティリティ。他のプログラムにリンクするためのライブラリを作成
nm	実行可能ファイル、オブジェクトファイル、またはライブラリ内のシンボルを一覧表示
ranlib	ar 形式アーカイブのインデックスを生成 (高速なリンク処理が可能)
size	オブジェクトファイルおよび実行可能プログラム内のセクションのサイズを出力
strings	バイナリファイルに埋め込まれている ASCII テキスト文字列を出力
strip	ファイルを小さくするために、デバッグシンボルおよび行番号の情報をファイルから削除

G.3　テキストファイル

awk	1つ以上のファイルをスキャンし、特定の条件に一致するすべての行に対して処理を実行
cat	ファイルを連結して結果を標準出力に送信
cksum	ファイルの CRC チェックサムを計算
csplit	コンテキストに基づくファイル分割
cut	ファイルの各行の選択したフィールドを切り出し
egrep	正規表現を最大限に使用してファイル内のあるパターンを検索
expand	ファイル内のタブを指定した個数の空白に置換
fgrep	ファイル内の固定文字列を検索
fmt	簡単なテキストフォーマッタ
fold	指定した段数に収まるようにファイルを折り返し
grep	grep、egrep および fgrep は、ファイル内の行で一致するパターンを検索
head	ファイルの先頭から指定した数の行を出力
indent	C ソースコード用のフォーマッタ
join	ファイル内の一致するフィールドに基づいて 2 つのファイルを条件付きで結合
less	UNIX の more や pg と同様にページ単位で表示
make	特別な形式で格納された一連の依存規則に基づいてファイルを更新
md5sum	MD5 アルゴリズムを使用してファイルの”フィンガープリント”を計算
more	画面単位でコピー
nl	ファイルに行番号を追加
od	8 進形式、16 進形式、複数の ASCII 形式など、さまざまな形式でファイルの内容をダンプ出力
patch	ファイルに変更を適用
paste	各ファイルから複数の行を連結して出力ファイルで 1 行にすることによって、複数のファイルを 1 つのファイルに結合
pr	ファイルを出力
sed	ラインエディタである ed をベースにしたストリームエディタ。データストリームの編集に便利
sort	ファイル内のフィールドに基づいて、そのファイル内の行を並び替え
split	複数の固定サイズファイルにファイルを分割
sum	ファイルの簡単なチェックサムを計算
tail	ファイルの末尾から指定した数の行を出力
tr	文字変換
tsort	トポロジの並べ替え
troff	ドキュメントの製版または書式設定
unexpand	指定した個数の連続する空白をタブ文字に変換
uniq	並べ替えたファイルから重複行を除去
vi	スクリーンテキストエディタ
wc	ファイル内の文字数、単語数、行数の 1 つ以上を取得

G.4　ファイルの比較

cmp　2 つのファイルを比較して行数およびバイト数の差異を表示
comm　2 個のファイルに共通する行を報告
diff　ファイルを比較し、異なる行を複数の形式のいずれかで表示
diff3　3 つのファイルを比較
sdiff　2 個のファイルを並べて比較し、対話型処理によって 3 番目のファイルに結合

G.5　シェルのユーティリティ

basename　引数であるファイル名を、ディレクトリ名を除いて表示
date　システムの日時を表示
dirname　ファイル名のディレクトリ部分を出力
echo　引数を出力
env　プログラムを実行する環境を変更するために使用
expr　算術演算子、関係演算子、論理演算子、および文字列演算子を使用した式を評価
factor　正の整数のすべての素因数を取得
hostname　ローカルホストマシンの名前
id　ユーザーおよびグループの ID と名前を表示
logname　ユーザーのログイン名を表示
pathchk　パス名をチェック
printenv　コマンドが実行される環境を出力
sh　Bourne シェル
sleep　指定した期間、シェルでのすべての処理を停止
tee　ファイルへの標準入力をコマンドラインで指定したファイルにコピーし、標準出力にもコピー
test　ファイルの種類をチェックして値を比較
uname　現在のシステムの名前を出力
whoami　ログイン名を表示
xargs　引数リストを作成してユーティリティを起動

G.6　ファイルのアーカイブ

cpio　ファイルのアーカイブとバックアップのユーティリティ
gzip　ファイル圧縮プログラム。圧縮と圧縮解除の両方を実行
gunzip　gzip と同じ
tar　アーカイブの作成と操作

付録H　viテキストエディタの概要

この付録では、vi テキストエディタでよく使用される機能の概要を説明します。

入力モードへの移行

i　カーソルの前から文字を挿入
a　カーソルの後から文字を挿入
I　行の先頭から文字を挿入
A　行末から文字を挿入
o　カーソルの次の行から文字を挿入
O　カーソルの前の行から文字を挿入

カーソル移動

l　カーソルを 1 文字右へ
h　カーソルを 1 文字左へ
j　カーソルを 1 行下へ
k　カーソルを 1 行上へ
$　カーソルを行末へ
^　カーソルを行頭へ
0　カーソルを行頭へ
w　カーソルを次の単語の先頭後へ
e　カーソルを次の単語の最後へ
*n*G　*n* 行目に移動
G　ファイルの最後へ
H　カーソルを画面の先頭へ
M　カーソルを画面の中間へ
L　カーソルを画面の最後へ

削除

dw　1 語削除
dd　1 行削除
dG　以降の行をすべて削除
d1G　以前の行をすべて削除
*n*dd　*n* 行削除
D　行末まで削除
x　1 文字削除
*n*x　*n* 文字削除

変更

cw	単語を変更
cc	行を変更
C	行末まで変更
r	カーソルのある文字を置換
s	カーソルのある文字を置換
S	行を置換。cc と同様

画面制御

CTRL-d	後方にスクロール
CTRL-u	前方にスクロール
CTRL-f	次の画面
CTRL-b	前の画面

コピーと貼り付け

Y	行をコピー
yy	行をコピー
*n*yy	*n* 行をコピー
p	下に貼り付け
P	上に貼り付け
"*x*y	行をバッファ *x* にコピー
"*xn*yy	yank *n* 行をバッファ *x* にコピー
"*x*p	下にバッファ *x* から貼り付け
"*x*P	上にバッファ *x* から貼り付け
"*x*d	バッファ *x* へ切り取り

その他の機能

J	カーソルのある行と次の行を結合
u	元に戻す
/	後方検索
?	前方検索
n	次の候補
N	前の候補
.	最後の操作を繰り返す
ZZ	保存して終了
ESC	キャンセルコマンド

コマンドモード

:w	上書き保存
:q	終了
:wq	保存して終了
:n	次のファイル
:r **ファイル**	**ファイル**の読み込み
:e **ファイル**	**ファイル**の編集
:f	ファイル名
:! **コマンド**	シェル**コマンド**を実行
:*n*	カーソルを *n* 行目に移動
:set **オプション**	オプションの変更
:set number	行番号を表示
:set nonumber	行番号表示を解除
:[アドレス]s/**旧**/**新**/[g]	**旧** を **新** で置き換え
アドレス .	現在の行
$	最終行
%	ファイル全体
g	行内の各候補

入力モード時のキー

BACKSPACE	1 文字削除
CTRL-w	単語の削除
ESC	コマンドモードに移行

付 録I　最新バージョンへのコードの移植

I.1　Ch バージョン 6.1.0.13631 へのコードの移植

1. stdbool.h でブール型”bool’ が intr から unsigned char に変更されました。

2. 既定の `complex` 型が、`float complex` から `double complex` に変更されました。

I.2　Ch バージョン 6.0.0.13581 へのコードの移植

1. stdbool.h でブール型”bool’ が char から int に変更されました。

2. 以下は変更されました。

```
CPlot::func2D(double (*func)(double x, void *param),
              void *param, double x0, double xf, int n);
CPlot::func3D(double (*func)(double x, double y, void *param),
              void *param, double x0, double xf, double y0, double yf,
              int nx, int ny);
```

   変更後は次のとおりです。

```
CPlot::funcp2D(double x0, double xf, int n,
               double (*func)(double x, void *param), void *param);
CPlot::funcp3D(double x0, double xf, double y0, double  yf,
               int nx, int ny,
               double (*func)(double x, double y, void *param),
               void *param);
```

3. 以下は変更されました。

```
CPlot::origin(double x, double y);
```

   変更後は次のとおりです。

```
CPlot::boundingBoxOrigin(double x, double y);
```

4. 以下は変更されました。

```
CPlot::grid(int flag, .../* int type */);
```

変更後は次のとおりです。

```
CPlot::grid(int flag, .../* char *option */);
```

以下は削除されました。

```
PLOT_GRID_POLAR
PLOT_GRID_RECTANGULAR
```

以下を変更してください。

```
plot.grid(PLOT_ON, PLOT_GRID_POLAR);
plot.polarPlot(PLOT_ANGLE_DEG);
```

変更後は次のとおりです。

```
   plot.grid(PLOT_ON);
または
   plot.grid(PLOT_ON, "polar");
または
   plot.grid(PLOT_ON, "polar 30");
       // the interval of radials is 30 degrees
   plot.polarPlot(PLOT_ANGLE_DEG);
```

5. 以下は変更されました。

```
CPlot::arrow(double x_head, double y_head, double z_head,
             double x_tail, double y_tail, double z_tail, ...
             /* [int linetype, int linewidth] */ );
```

変更後は次のとおりです。

```
CPlot::arrow(double x_head, double y_head, double z_head,
             double x_tail, double y_tail, double z_tail, ...
             /* [string_t option] */ );
```

以下を変更してください。

```
plot.arrow(x1, y1, z1, x2, y2, z2, 1, 3);
```

変更後は次のとおりです。

```
char option[64];
sprintf(option, "linetype 1 linewidth 3");
plot.arrow(x1, y1, z1, x2, y2, z2, option);
```

6. `CPlot::axisRange(int axis, double minx, double max, double incr);`

   これは非推奨です。以下を使用してください。

```
CPlot::axisRange(int axis, double minx, double max);
CPlot::ticsRange(int axis, incr);
```

索引

\" ダブルクォートエスケープシーケンス, 153
\' シングルクォートエスケープシーケンス, 153
\?, 153
\\ バックスラッシュ文字エスケープシーケンス, 153
\a アラートエスケープシーケンス, 153
\b エスケープシーケンス, 153
\f 改ページエスケープシーケンス, 153
\n (改行エスケープシーケンス), 153
\r 復帰（キャリッジリターン）エスケープシーケンス, 153
\t 水平タブエスケープシーケンス, 153
\v エスケープシーケンス, 153
|| logical inclusive OR operator, 357
|| 論理 OR 演算子, 162, 168
| pipe, 106
| ビットごとの OR, 162, 168
|= ビットごとの OR 代入演算子, 162, 169
( ), 162
* 間接参照演算子, 162
間接演算子, 190
* 乗算演算子, 162, 164, 279, 280
乗算演算子, 351
* ワイルドカード文字, 99
*= 左辺値に乗算演算子, 162
*= 乗算代入演算子, 169, 354
+ 加算演算子, 162, 279, 280
+ 単項プラス, 163
単項プラス, 351
+ 符号, 162
++ インクリメント演算子, 162
++ 増加演算子, 355
++ 増分演算子, 175
+= 加算代入演算子, 169, 354
+= 左辺値に加算演算子, 162
, コンマ演算子, 162, 173
- sign, 279, 280
- 減算演算子, 162, 164, 279, 280
- 単項マイナス, 163
単項マイナス, 351
- 符号, 162
-= 減算代入演算子, 169, 354
-= 左辺値から減算演算子, 162
. 構造体メンバ演算子, 74, 410
. コマンド, 79, 82, 83
. メンバ演算子, 162
. メンバ演算子の構造, 176
.* 配列乗算演算子, 162, 165
配列の乗算, 351
親ディレクトリ, 99
現在の作業ディレクトリ, 99
./ 配列除算演算子, 162, 165
配列の除算, 351
.chlogin, 54, 660
.chlogout, 54, 660
.chrc, 54, 60, 63, 488, 495, 660
.chslogin, 54
.chsrc, 54
.cshrc, 660
.login, 660
.logout, 660
/ 除算演算, 279
/ 除算演算子, 162, 281
/ 除法演算子, 165
/= 左辺値に除算演算子, 162
/= 除算代入演算子, 354
:, 335, 339, 363
:: scope resolution operator, 162
:: スコープ解決演算子, 423, 653
= 代入演算子, 162, 169, 354
== 等号演算子, 167
== 等しい演算子, 162, 279, 281, 356
ワイルドカード文字, 99
?: 条件演算, 358
?: 条件演算子, 162, 169
[ ], 162
49
アドミニストレータープロンプト, 76
スーパーユーザープロンプト, 76
プリプロセッサ演算子, 122
プリプロセッサトークン, 126
#!/bin/ch, 114
#!/bin/csh, 114
#!/bin/ksh, 114
#!/bin/sh, 114
プリプロセッサ演算子, 122
プリプロセッサトークン, 127
#define プリプロセッサディレクティブ, 122, 123, 125
#defined プリプロセッサ演算子, 122
#elif プリプロセッサディレクティブ, 122, 123
#else プリプロセッサディレクティブ, 122
#endif プリプロセッサディレクティブ, 122
#error プリプロセッサディレクティブ, 122, 129

索引

#if プリプロセッサディレクティブ, 122, 123
#ifdef プリプロセッサディレクティブ, 122
#ifndef プリプロセッサディレクティブ, 122
#include プリプロセッサディレクティブ, 122, 123
#line プリプロセッサディレクティブ, 122, 128
#pragma, 64, 122, 129
#undef プリプロセッサディレクティブ, 122, 123
$ Bourne, Korn, BASH シェルプロンプト, 76
コマンド名置換, 98
$ 式置換, 97
$ 式の置換, 465
$ 変数の置換, 464
$ 変数置換, 96, 464
$argv, 659
% C シェルプロンプト, 76
% 除算の剰余演算子, 162
%= 左辺値を乗算して剰余を代入演算子, 162
&, 369
& アドレス演算子, 162, 174, 251, 358, 395
アドレス演算子, 190
& バックグラウンドでのコマンド, 108
& ビットごとの AND, 162, 168
&= ビットごとの AND 代入演算子, 162, 169
&& 論理 AND 演算子, 162, 168, 357
^ 簡易置換, 89
__BIG_ENDIAN__, 133
__class__, 45, 47, 443
__class_func__, 45, 47, 443
__DATE__, 132
__declspec, 215
__declspec(global), 73, 75
__declspec(local), 75, 150, 216
__FILE__, 132
__func__, 45, 47, 443, 641
__i386__, 133
__LINE__, 128, 132
__LITTLEENDIAN__, 133
__ppc__, 133
__STDC__, 132
__STDC_VERSION__, 132
__TIME__, 132
__VA_ARGS__, 126, 641
__x86_x64__, 133
AIX, 133
_argc, 45, 47, 66, 118, 659
_argv, 45, 47, 66, 118, 659
_CH_, 132
CHDLL, 132
_chlogin, 54
_chlogout, 54
_chrc, 54, 60, 63
_chslogin, 54
_chsrc, 54
_cwd, 45, 47, 49, 481
_cwdn, 45, 47, 49, 481
DARWIN, 133
_environ, 45, 47
_errno, 45, 47
_execv(), 482
_execvp(), 482
_fopen(), 482
_fork(), 482
_formatd, 45, 47, 55
_formatf, 45, 47, 55
_fpath, 44, 45, 47, 63, 129, 481, 487, 644
_fpathext, 45, 47, 61
FREEBSD, 133
_fstat(), 482
GLOBALDEF, 132
_histnum, 45, 47
_histsize, 45, 47, 660
_home, 45, 47, 49, 481
_host, 45, 47, 481
HPUX, 133
_iath, 44
_ignoreeof, 45, 47, 56, 639
_ignoretrigraph, 45, 47
_IOFBF, 446
_IOLBF, 446
_IONBF, 446
_ipath, 45, 47, 123, 129, 481, 487, 493
_lang, 45, 47, 49
_lc_all, 45, 47, 49
_lc_collate, 45, 47, 49
_lc_ctype, 45, 47, 49
_lc_monetary, 45, 47, 49
_lc_numeric, 45, 47, 49
_lc_time, 45, 47, 49
LINUX, 133
LINUXPPC, 133
_logname, 45, 47, 49
_lpath, 44, 46, 47, 129, 481, 487
_lstat(), 482
_M64_, 132
_new_handler, 46, 47, 419
_path, 44, 46, 48, 49, 60, 78, 129, 481, 487, 644
_pathext, 46, 48
_pipe(), 482
_popen(), 482
_ppath, 46, 48, 481
_prompt, 46, 48, 56, 76
QNX, 133
_remove(), 482
_rename(), 482
SCH, 132
_setlocale, 46, 48, 409
_shell, 46, 48, 49, 481
_socket(), 482

索引

_socketpair(), 482
SOLARIS, 133
_stat(), 482
_status, 46, 48, 66, 659
_stop(), 72
_system(), 482
_term, 46, 48, 49
_tz, 46, 48, 49
_user, 46, 48, 49, 481
_utime(), 482
_warning, 46, 48, 55
WIN32, 133
_X86_, 133
! イベント指示子, 85, 88
! 論理 NOT 演算子, 162, 168, 357
!= 非等号, 168
=+ 等しくない演算子, 356
!= 等しくない演算子, 162, 279, 281
−− 減少演算子, 355
−− 減分演算子, 175
−− デクリメント演算子, 162
−> structure pointer operator, 176
−> 構造体ポインタ演算子, 74, 410
−> ポインタ演算子, 162
< 標準出力ストリームのリダイレクト, 103
< 標準入力ストリームのリダイレクト, 103
< より大きい演算子, 162
< より少ない演算子, 356
< より小さい演算子, 166
<< 左シフト, 168
<< ビット左シフト, 162
<< 標準出力ストリーム演算子, 459
<< 標準入力ストリームのリダイレクト, 103
<<= 左シフト代入演算子, 162, 169
<= 以下演算子, 162, 166
<= 少ないか等しい演算子, 356
> Ch プロンプト, 76
> より大きい演算子, 168, 356
> より小さい演算子, 162
>= 以上比較演算子, 167
>= より大きいか等しい演算子, 356
>= より大きいもしくは同じ演算子, 162
>> ストリーム抽出演算子, 459
>> ビット右シフト, 162
>> 標準出力ストリームのリダイレクト, 103
>> 右シフト, 168
>>= 右シフト代入演算子, 162, 169
^ ビットごとの XOR 演算子, 162, 168
^= ビットごとの排他 OR 代入演算子, 169
^^ 論理 XOR 演算子, 162
論理 XOR 演算子, 168
^^ 論理排他 OR 演算子, 357, 644
~ ビットごとの補数, 162, 168
~ ホームディレクトリ, 99
2>&1 標準出力および標準エラーストリームのリダイレクト, 103
‘ コマンド置換演算子, 162, 176
コマンド置換関数, 101
0B, 155
0b, 155
0X, 155
0x, 155
16 進数, 155
16 進浮動小数点定数, 642
16 進浮動小数点の定数, 157
1 次元配列, 298
1 の補数, 168
2 進数, 155
8 進数, 155

abs(), 43, 262, 282, 286, 380
accept(), 482
access(), 43, 116, 482
acos(), 43, 262, 282, 286
acosh(), 43, 262, 282, 286
aio.h, 484
AIX, 133
alias, 57, 659
alias(), 43, 249
all(), 360
AND, 168
any(), 360
Aquaterm, 495
argc, 65, 243
argv, 65, 243, 659
array.h, 347, 484
arraycopy(), **233**
arrow(), 496, *see* CPlot, *see* CPlot
asin(), 43, 262, 282, 286, 381
asinh(), 43, 262, 282, 286, 381
assert.h, 484
atan(), 43, 262, 282, 286, 381
atan2(), 43, 262, 282, 381
atanh(), 43, 262, 282, 286, 381
atexit(), 43
auto, 42, 75
autoScale(), 496, *see* CPlot
awk, 682
axes(), 496
axis(), 496, *see* CPlot
axisRange(), 496, *see* CPlot, *see* CPlot

balance(), 552, **622**
barSizd(), 496
basename, 683
BASH, 76, 114
beginparalleltask, 44
bool, 642
border(), 496, *see* CPlot

borderOffsets(), 496, *see* CPlot
boundingBoxOrigin(), *see* CPlot
Bourne シェル, 76, 114
boxBorder(), 496
boxWidth(), 496
break, 42, 186

C+, 22
calloc(), 193, 294
carg(), 282
case, 42, 181
cat, 682
catch, 44
Cauchy, 607
ccompanionmatrix(), 552, **605**
cd, 83, 680
cdeterminant(), 552
cdiagonal(), 552, **600**
cdiagonalmatrix(), 552, **603**
ceil(), 43, 262, 282, 286, 381
cerr, 459, 653
cfevalarray(), 552, **572**
cfum(), **608**
cfunm(), 552
Ch, 3, 22
ch, 53, 112
Ch オプション, 58
CH_CARRAYPTRTYPE, 226
CH_CARRAYTYPE, 226
CH_CARRAYVLA, 226
CH_CHARRAYPTRTYPE, 226
CH_CHARRAYTYPE, 226
CH_CHARRAYVLATYPE, 226
CH_UNDEFINETYPE, 226
changeViewAngle(), 496, *see* CPlot
char, 42, 135, 399
CHAR_MAX, 135
CHAR_MIN, 135
charpolycoef(), 552, **589**
chdebug, 71, 85
chdir, 83
chdir(), 482
Chebyshev, 607
chgrp, 680
CHHOME, 49, 52
chlogin, 54
chmod, 60, 680
choldecomp(), 552, **612**
Cholesky 分解, 612
Chow, 607
chown, 680
chown(), 482
chparse, 70, 81, 85, 481
chplot.h, 484
chrc, 409, 495
chroot(), 482
chrun, 70, 81, 85, 481
chs, 53, 112, 482
chs.exe, 482
chshell.h, 484
chslogin, 54
Ch シェル, 53, 76, 114
ch デバッグ, 70
cin, 459, 653
cinverse(), 552, **620**
circle(), 496, *see* CPlot
Circul, 607
cksum, 682
class, 42, 649
Clement, 607
clinsolve(), 552
clock(), 43, 131
CLOCKS_PER_SEC, 131
closedir(), 473
cls, 640
cmean(), 552, **563**
cmp, 683
colorBox(), 496
combination(), 552, **568**
comm, 683
command files, 59
communication, 242
companionmatrix(), 552, **605**, **606**
complex, 42, 140, 272, 273, 641
　演算子, 279
　定数, 272
　複素関数, 282
　複素数, 272
　複素数に対する入出力, 278
complex(), 282, 287
complex.h, 273, 484, 642
ComplexInf, 42, 286, 642, 649
ComplexNaN, 42, 286, 642, 649
complexsolve(), 552, **557**
condnum(), 552
conj(), 43, 282, 286
const, 42
const メンバ関数, 653
continue, 42, 186
contourLabel(), 496, *see* CPlot
contourLevels(), 496, *see* CPlot
contourMode(), 496, *see* CPlot
conv(), 552, **626**
conv2(), 552, **628**
coordSystem(), 496, *see* CPlot
copy, 640
copyright, 2
copysign(), 165

corr2(), 552
corrcoef(), 552, **565**
correlation(), **566**
cos(), 43, 262, 282, 286, 381
cosh(), 43, 262, 282, 286, 381
count(), 360
cout, 459, 653
covariance(), 552, **565**
cp, 680
cpio, 683
cpio.h, 484
CPlot
  ˜CPlot, 496
  arrow(), 496, **509**
  autoScale(), 496
  axes(), 496
  axis(), 496, **509**
  axisRange(), 496, **508**
  barSize(), 496
  border(), 496, **509**
  borderOffsets(), 496
  boxBorder(), 496
  boxWidth(), 496
  changeViewAngle(), 496
  circle(), 496, **517**
  colorBox(), 496
  contourLabel(), 496
  contourLevels(), 496
  contourMode(), 496
  coordSystem(), 496, **540**
  CPlot(), 496
  data(), 496
  data2D(), 496, **499**
  data2DCurve(), 496, **499**, **503**
  data2DSurface(), **503**
  data3D(), 496, **499**
  data3DCurve(), 496
  data3DSurface(), 496
  dataFile(), 496, **503**
  dataSetNum(), 496
  deleteData(), 496
  deletePlots(), 496
  dimension(), 496, **504**
  displayTime(), 496
  enhanceText(), 496
  fillStyle(), 496
  func2D(), 496
  func3D(), 496
  funcp2D(), 497
  funcp3D(), 497
  getLabel(), 497
  getOutputType(), 497
  getSubplot(), 497, **519**
  getTitle(), 497
  grid(), 497
  isUsed(), 497
  label(), 497, **505**
  legend(), 497, **512**
  legendLocation(), 497, **512**
  legendOption(), 497
  line(), 497, **517**
  lineType(), 497, **527**
  margins(), 497
  origin(), 497
  outputType(), 497, **521**
  plotting(), **495**, 497
  plotType(), 497, **526**
  point(), 497
  pointType(), 497, **532**
  polarPlot(), 497, **535**
  polygon(), 497, **518**
  rectangle(), 497, **517**
  removeHiddenLine(), 497
  scaleType(), 497
  showMesh(), 497
  size(), 497
  size3D(), 497
  sizeRatio(), 497, **536**
  smooth(), 497
  subplot(), 497, **519**
  text(), 497, **510**
  tics(), 497
  ticsDay(), 497
  ticsDirection(), 497
  ticsFormat(), 498
  ticsLabel(), 498
  ticsLevel(), 498
  ticsLocation(), 498
  ticsMirror(), 498
  ticsMonth(), 498
  ticsPosition(), 498
  ticsRange(), 498
  title(), 498, **505**
cpolyeval(), 552, **582**
cproduct(), 550, 552, **562**
creat(), 482
createpkg.ch, 493
cross(), **551**, 552
crypt.h, 484
csplit, 682
csum(), 550, 552, **561**
ctrace(), 552, **600**
ctriangularmatrix(), 552, **604**
ctype.h, 484
cumprod(), 552, **562**
cumsum(), 552, **561**
curvefit(), 552, **577**
C シェル, 76, 111, 114, 658, 659

C 配列, 397

DARWIN, 133
data(), 496
data2D(), 496, *see* CPlot
data2DCurve(), 496
data3D(), 496, *see* CPlot
data3DCurve(), 496
data3DSurface(), 496
dataFile(), 496, *see* CPlot
dataSetNum(), 496
date, 683
DBL_MAX, 139
DBL_MIN, 139
DBL_MINIMUM, 139
dd, 680
deconv(), 552, **627**
default, 42
del, 640
delete, 42, 390, 419, 649
deleteData(), 496
deletePlots(), 496, *see* CPlot
DenavitHartenberg, 607
DenavitHartenberg2, 607
derivative(), 552, **591**
derivatives(), 552, **591**
determinant(), 552, **598**, **599**
df, 680
diagonal(), 552, **600**
diagonalmatrix(), 552, **603**
diff, 683
diff3, 683
difference(), 552, **591**
dimension(), 496, *see* CPlot
dir, 640, 680
dirent.h, 473, 484
dirname, 683
dirs, 112
DISPLAY, 29, 110
displayTime(), 496, *see* CPlot
dlfcn.h, 484
dlopen(), 43
dlrunfun(), 43, 249
dlsym(), 43
do, 42
Do-While, 183
dot command, 83
dot(), 553, **555**
double, 42, 139
double complex, 140, 641
Dramadah, 607
du, 680

echo, 683
egrep, 682
eigen(), 553, **622**
elementtype(), 43, 249, 374, 430
else, 42
else-if, 180
endl, 459, 653
endlocal, 640
endparalleltask, 44
enhanceText(), 496, *see* CPlot
enum, 42, 146
env, 110, 683
environ, 65, 246
EOF, 56
erase, 640
errno.h, 484
eval, 659
event_t, 44
exclusive, 644
exec, 83, 129, 489
execl(), 482
execle(), 482
execlp(), 482
execv(), 482
execve(), 482
execvp(), 482
exit, 85
exp(), 43, 262, 282, 286, 381
expand, 682
expm(), 553, **608**
expr, 683
extern, 42, 75

F_OK, 116
fabs(), 262, 282
factor, 683
factorial(), 553, **567**
false, 179, 642
fchdir(), 482
fchown(), 482
fchroot(), 482
fcntl.h, 484
fdopen(), 482
fenv.h, 484, 642
fevalarray(), 553, **571**
fflush(), 447, 469
fft(), 553, **624**
fgetc(), 470
fgetpos(), 445
fgets(), 43
fgrep, 682
Fiedler, 607
FILE, 468
fillStyle(), 496
filter(), 553, **629**
filter2(), 553, **633**

find, 680
findvalue(), 553, **568**
fliplr(), 553, **601**
flipud(), 553, **602**
float, 42
float.h, 484
floor(), 43, 262, 282, 286, 381
FLT_EPSILON, 267
FLT_MAX, 139, 268
FLT_MIN, 139, 267
FLT_MINIMUM, 139, 267
fminimum(), 553, **579**
fminimums(), 553, **580**
fmod(), 43, 262, 282, 290
fmt, 682
fold, 682
fopen(), 468, 482
for, 42, 184
foreach, 42, 185, 408, 644, 649
FORTRAN, 250, 257, 676
Fortran, 676
fplotxy(), **539**
fplotxyz(), **545**
fpos_t, 445
fprintf, 42, 457, 464, 649
fprintf(), 447, 457
fputc(), 471
fputs(), 471
Frank, 607
fread(), 43, 470, 471
free(), 43, 194, 294, 644
FreeBSD, 133
frexp(), 43, 262, 282, 286
fscanf, 458
fscanf(), 43, 452, 458
fseek, 469
fseek(), 472
fsetpos, 469
fsetpos(), 445
fsolve(), 553, **590**
fstat(), 482
ftell(), 472
func2D(), 496
func3D(), 496
funcp2D(), 497
funcp3D(), 497
funm(), 553, **608**
fwprintf(), 652
fwrite(), 471
fwscanf(), 652
fzero(), 553, **590**

gawk, 682
gcd(), 553, **556**
Gear, 607
getc(), 470
getenv(), 32, 43, 57, 109, 482, 659
gethostname(), 482
getLabel(), 497, *see* CPlot
getnum(), 553, **558**
getOutputType(), 497
gets(), 43
getSubplot(), 497, *see* CPlot
getTitle(), 497, *see* CPlot
glob.h, 484
global, 73, 75, 643
GNUTERM, 495
goto, 42, 73, 187, 311
grep, 682
grid(), 497, *see* CPlot, *see* CPlot
grp.h, 484
gunzip, 683

Hadamard, 607
Hankel, 607
head, 682
help, 31, 112
hessdecomp(), 553, **615**
Hessenberg 分解, 615
Hilbert, 607
histogram(), 553, **572**
history, 85, 87, 660
HOME, 49
hostname, 683
householdermatrix(), 553
HP_UX, 133
hypot(), 161, 650

id, 683
identitymatrix(), 553, **603**
IEEE 754, 641
IEEE 754 標準, 138
if, 42, 179
if-else, 180
ifft(), 553, **624**
imag(), 43, 276, 282, 286, 383
import, 67, 129
importf, 64, 67, 129
indent, 682
inet.h, 484
Inf, 42, 139, 461, 641, 649, 650
inline, 42, 150, 642
installpkg.ch, 493
int, 42, 135, 136
INT_MAX, 136
INT_MIN, 136
integral1(), 553, **595**
integral2(), 553, **596**
integral3(), 553, **597**

索 引

integration2(), 553, **596**
integration3(), 553, **597**
interp1(), 553, **573**
interp2(), 553, **574**
inttypes.h, 484, 642
inverse Hilbert matrix, 607
inverse(), 553, **619**
ioctl(), 43
iostream.h, 460, 484
isenv(), 32, 109
iskey(), 43
iso646.h, 484, 642
isUsed(), 497, *see* CPlot

join, 682

K&R C, 648
kill(), 482
Korn シェル, 76, 114

label(), 497, *see* CPlot
LANG, 49
LC_ALL, 49
LC_COLLATE, 49
LC_CTYPE, 49
LC_MONETARY, 49
LC_NUMERIC, 49
LC_TIME, 49
lchown(), 482
lcm(), 553, **557**
ldexp(), 43, 262, 282, 286
legend(), 497, *see* CPlot
legendLocation(), 497, *see* CPlot
legendOption(), 497
less, 682
libintl.h, 484
limits.h, 135, 484
lindata(), 553, **559**
line(), 497, *see* CPlot
lineType(), 497, *see* CPlot
link(), 482
linsolve(), 553, **617**
linspace(), 553, **559**
Linux, 133
Linux での起動, 28
llsqcovsolve(), 553, **619**
llsqnonnegsolve(), 553, **618**
llsqsolve(), 554, **617**
ln, 680
local, 75, 643
locale.h, 409, 484
log(), 43, 262, 282, 286, 381
log10(), 43, 262, 282, 286, 381
logdata(), 554, **559**
login, 29
logm(), 554, **608**
LOGNAME, 49
logname, 683
logspace(), 554
long, 42, 135, 137
long double, 648
long double complex, 648
long long, 135, 137, 641, 646
longjmp(), 186
loop, 182
    while loop, 182
ls, 680
lstat(), 482
LU 分解公式, 609
ludecomp(), 554, **609**
lvalue, 290

Mac OS X での起動, 28
Magic, 607
main(), 65, 243
make, 682
malloc(), 193, 294
malloc.h, 484
margins(), 497, *see* CPlot
math.h, 484
max(), 43, 249
maxloc(), 550, **560**, 568
maxloc(0, 554
maxv(), 554, **560**
MB_CUR_MAX, 152
mbstate_t, 445
mbstowcs(), 155
md5sum, 682
mean(), 550, 554, **563**
median(), 550, 554, **563**
memchr(), 402
memcmp(), 401
memcpy(), 43, 400
memmove(), 43, 400
memset(), 43, 403
min(), 43, 249
minloc(), 550, 554, **560**, 568
minv(), 554, **560**
mkdir, 77, 680
mkdir(), 482
modf(), 43, 262, 282, 286
more, 682
move, 640
mqueue.h, 484
multiple files, 67
mv, 680

NaN, 42, 139, 461, 641, 649, 650
netconfig.h, 484
netdb.h, 484

netdir.h, 484
netinet/in.h, 484
new, 42, 193, 390, 419, 649
new.h, 484
nl, 682
nm, 681
norm(), 554, **566**
NOT, 168
NULL, 42, 193, 376, 644, 649
nullspace(), 554, **621**
null 空間, 621
NULL ディレクティブ, 129
numeric.h, 484

od, 682
oderk(), 554, **592**
offsetof(), 482
open(), 43, 482
opendir(), 473
operator, 42
origin(), 497, *see* CPlot
orthonormalbase(), 554, **621**
outputType(), 497, *see* CPlot

pack(), 129
package, 129
parse, 81
Pascal, 607
paste, 682
patch, 682
PATH, 49
pathchk, 683
pclose(), 107
pinverse(), 554, **620**
pipe(), 482
pipeline, 106
PLOT_ANGLE_DEG, 535, 542
PLOT_ANGLE_RAD, 535, 542
PLOT_AXIS_X, 505
PLOT_AXIS_X2, 505
PLOT_AXIS_XY, 505
PLOT_AXIS_XYZ, 505
PLOT_AXIS_Y, 505
PLOT_AXIS_Y2, 505
PLOT_AXIS_Z, 505
PLOT_BORDER_ALL, 509
PLOT_BORDER_BOTTOM, 509
PLOT_BORDER_LEFT, 509
PLOT_BORDER_RIGHT, 509
PLOT_BORDER_TOP, 509
PLOT_COORD_CARTESIAN, 540
PLOT_COORD_CYLINDRICAL, 540
PLOT_COORD_SPHERICAL, 540
PLOT_OFF, 509
PLOT_ON, 509
PLOT_OUTPUTTYPE_DISPLAY, 521
PLOT_OUTPUTTYPE_FILE, 521
PLOT_OUTPUTTYPE_STREAM, 521
PLOT_PLOTTYPE_BOXERRORBARS, 526
PLOT_PLOTTYPE_BOXES, 526
PLOT_PLOTTYPE_BOXXYERRORBARS, 526
PLOT_PLOTTYPE_CANDLESTICKS, 526
PLOT_PLOTTYPE_DOTS, 526
PLOT_PLOTTYPE_FILLEDCURVES, 526
PLOT_PLOTTYPE_FINANCEBARS, 526
PLOT_PLOTTYPE_FSTEPS, 526
PLOT_PLOTTYPE_HISTEPS, 526
PLOT_PLOTTYPE_IMPULSES, 526, 539
PLOT_PLOTTYPE_LINES, 526, 539
PLOT_PLOTTYPE_LINESPOINTS, 526, 539
PLOT_PLOTTYPE_POINTS, 526, 539
PLOT_PLOTTYPE_STEPS, 526
PLOT_PLOTTYPE_SURFACES, 539
PLOT_PLOTTYPE_VECTORS, 526, 539
PLOT_PLOTTYPE_XERRORBARS, 526
PLOT_PLOTTYPE_XERRORLINES, 526
PLOT_PLOTTYPE_XYERRORBARS, 526
PLOT_PLOTTYPE_XYERRORLINES, 526
PLOT_PLOTTYPE_YERRORBARS, 526
PLOT_PLOTTYPE_YERRORLINES, 526
PLOT_TEXT_CENTER, 510
PLOT_TEXT_LEFT, 510
PLOT_TEXT_RIGHT, 510
plotting(), 497, *see* CPlot
plotType(), 497, *see* CPlot
plotxy(), **536**, 644
plotxyf(), **538**, 644
plotxyz(), **543**, 644
plotxyzf(), **544**, 644
point(), 497, *see* CPlot
pointType(), 497, *see* CPlot
polar(), 43, 249, 282, 286, 289
polarPlot(), 497, *see* CPlot
poll.h, 484
polycoef(), **586**
polyder(), 554, **583**
polyder2(), 554, **584**
polyeval(), 554, **582**
polyevalarray(), 554, **582**
polyevalm(), 554, **582**, **608**
polyfit(), 554
polygon(), 497, *see* CPlot
pop, 129
popd, 112
popen(), 107, 482
POSIX, 484
pow(), 43, 262, 282, 289, 382, 650
pr, 682
pragma, 67, 409, 490, 491

_fpath, 129
_ipath, 129
_lpath, 129
_path, 129
exec, 129, 489
import, 67, 129, 490
importf, 67, 129, 490
pack(), 129
package, 129, 491
remkey(), 129
remvar(), 129
printenv, 683
printf, 42, 649
printf(), 447, 457
private, 42, 417, 649
product(), 550, 554, **562**
protected, 44
pthread.h, 484
public, 42, 417, 649
push, 129
pushd, 112
putc(), 471
putenv(), 29, 32, 57, 109, 482, 659
puts(), 471
PWD, 49
pwd, 77, 83, 84, 87, 88, 97, 680
pwd.h, 484

QNX, 133
qr, 664
qrdecomp(), 554, **613**
qrdelete, 664
qrinsert, 664
QR 分解, 613
qsort(), 248

R_OK, 116
rank(), 554, **601**
ranlib, 681
rcondnum(), 554
re_comp.h, 484
read(), 43
readdir(), 473
readline.h, 484
real(), 43, 276, 282, 286, 383
realloc(), 193, 294
rectangle(), 497, *see* CPlot
recvevent, 44
regex.h, 484
register, 42, 75, 648
remenv(), 32, 109, 659
remkey, 85
remkey(), 129
remove(), 482
removeHiddenLine(), 497, *see* CPlot
remvar, 85
remvar(), 129
ren, 640
rename(), 482
residue(), 554, **587**
resize, 29
resize(), 29
restrict, 42, 642, 648
return, 42, 187
rewind, 469
rewinddir(), 473
rlimit, 56
rm, 680
rmdir, 680
rmdir(), 482
roots(), 554, **585**
Rosser, 607
rot90(), 554, **602**
rsf2csf, 664

safe Ch, 112
scaleType(), 497, *see* CPlot
scanf, 42, 649
scanf(), 452, 458
sched.h, 484
schurdecomp(), 554, **616**
Schur 分解, 616
sdiff, 683
sed, 682
SEEK_CUR, 472
SEEK_END, 472
semaphore.h, 484
sendevent, 44
set, 659
set_new_handler(), 419
setbuf(), 446
setenv, 111, 659
setjmp(), 186
setjmp.h, 484
setlocale, 409
setlocale(), 409
setrlimit(), 43, 56, 482
setvbuf(), 446
sh, 683
shape, 376
shape(), 43, 365
SHELL, 49
short, 42, 135
showMesh(), 497, *see* CPlot
showvar, 79, 85, 86
SHRT_MAX, 135
SHRT_MIN, 135
sign(), 554, **556**
signal.h, 484

signbit(x), 165
signed, 42
signed char, 135
signed int, 135
signed long, 135
signed long long, 135
signed short, 135
sin(), 43, 262, 282, 286, 381
sinh(), 43, 262, 282, 286, 381
size, 681
size(), 497, *see* CPlot
size_t, 399
size3D(), 497, *see* CPlot
sizeof, 42, 315, 321, 329
sizeof(), 282, 286
sizeRatio(), 497, *see* CPlot
sleep, 683
smooth(), 497
socket(), 482
socketpair(), 482
Solaris, 133
sort, 682
sort(), 550, 554, **569**
specialmatrix(), 554, **607**
split, 682
sqrt(), 43, 262, 282, 286, 381, **608**
sqrtm(), 554
sscanf(), 43, 452, 458
stackvar, 79
start, 640
startup, 480
stat(), 482
static, 42, 75
status, 659
std(), 550, 554, **564**
stdarg.h, 226, 426, 484
stdbool.h, 179, 484, 642
stddef.h, 152, 484
stdin, 445
stdin.h, 642
stdio.h, 484
stdlib.h, 484
stdout, 445
stop, 72
str2ascii(), 405, 643
str2mat(), 405, 643
stradd(), 43, 56, 60, 405, 488, 643
strcasecmp(), 404
strcat(), 43, 147, 401
strchr(), 43, 402
strcmp(), 43, 401
strcoll(), 43, 401
strconcat(), 404
strcpy(), 43, 400
strcspn(), 402
strerror(), 43, 403
streval(), 43, 108, 659
strgetc(), 405, 643
string.h, 399, 484
string_t, 42, 404, 649
stringcat(), 147
stringcat2(), 148
strings, 681
strip, 681
strjoin(), 404, 659
strlen(), 43, 403
strncasecmp(), 404
strncat(), 43, 401
strncmp(), 401
strncpy(), 43, 400
stropts.h, 484
strparse(), 109
strpbrk(), 402
strputc(), 405, 643
strrchr(), 402
strrep(), 405, 643
strspn(), 402
strstr(), 402
strtod(), 43
strtok(), 43, 402, 408
strtok_r(), 408
strtol(), 43
strtoul(), 43
struct, 42
strxfrm(), 43, 401
subplot(), 497, *see* CPlot
sum, 682
sum(), 550, 554, **561**
svd(), 554, **611**
swap(), 224, 253
switch, 42, 181
swprintf(), 652
swscanf(), 652
syslog.h, 484
system(), 60, 105, 482

tail, 682
tan(), 43, 262, 282, 286, 381
tanh, 381
tanh(), 43, 262, 282, 286
tar, 683
tar.h, 484
tee, 683
TERM, 49
termios.h, 484
test, 683
text(), 497, *see* CPlot
tgmath.h, 484, 642

this, 42, 649, 653
tics(), 497, *see* CPlot
ticsDay(), 497, *see* CPlot
ticsDirection(), 497, *see* CPlot
ticsFormat(), 498, *see* CPlot
ticsLabel(), 498, *see* CPlot
ticsLevel(), 498, *see* CPlot
ticsLocation(), 498, *see* CPlot
ticsMirror(), 498, *see* CPlot
ticsMonth(), 498, *see* CPlot
ticsPosition(), 498
ticsRange(), 498
time, 640
time.h, 484
title, 640
title(), 498, *see* CPlot
tiuser.h, 484
Toeplitz, 607
touch, 680
tr, 682
trace(), 554, **600**
transpose(), 43, 249, 353, 383
triangularmatrix(), 555, **604**
trigraph, 47
troff, 682
true, 179, 642
try, 44
tsort, 682
type, 640
typedef, 316, 326
TZ, 49

UCHAR_MAX, 135
UINT_MAX, 137
umask, 659
umask(), 43, 55, 482
unalias, 659
uname, 683
unexpand, 682
union, 42
uniq, 682
unistd.h, 484
unlink(), 482
unset, 659
unsetenv, 659
unsigned, 42
unsigned char, 135
unsigned int, 135
unsigned long, 135
unsigned long long, 135
unsigned short, 135
unwrap(), 555
urand(), 555, **555**
USER, 49
USHRT_MAX, 136
using, 460, 653
utime.h, 484

va_arg(), 226
va_arraydim(), 226, 426, 430
va_arrayextent(), 226, 426, 430
va_arraynum(), 226, 426, 430
va_arraytype(), 226, 426, 430
va_copy(), 226
va_count(), 226, 426
va_datatype(), 226, 426, 430
va_end(), 226, 426
va_list, 226
VA_NOARG, 226, 229, 426
va_start(), 226
va_tagname(), 226
Vandermonde, 607
ver, 640
verify, 640
vfprintf(), 43, 457
vfwprintf(), 652
vi, 682, 684
vim, 682
virtual, 44, 150
VLA, 304
VLAs, 641
void, 42, 146
vol, 640
volatile, 42, 150, 648
vprintf(), 43, 457
vsprintf(), 43, 457
vswprintf(), 652
vwprintf(), 652

W_OK, 116
wait.h, 484
wc, 682
wchar.h, 484, 642
wchar_t, 399, 409
wcstombs(), 155
wctype.h, 484, 642
Web プロット, 545
which, 112, 119, 244, 680
while, 42, 182
whoami, 683
Wilkinson, 607
Win32, 133
Windows, 639
Windows 2000, 639
Windows 7, 639
Windows Vista, 639
Windows XP, 639
Windows での起動, 30
WinMain(), 65

wprintf(), 652
wscanf(), 652

X_OK, 116
xargs, 683
xcorr(), 555, **635**
xhost, 29, 110

値呼び出し, 201
アリティ可変関数, 654
アリティ可変の関数, 644
アンラップ, 570

一様乱数, 555
イベント指示子, 87
入れ子された関数, 242
　入れこされた再起関数, 220
　スコープ, 212
　レキシカルレベル, 212
入れ子にされた関数, 211
　入れ子にされた関数のプロトタイプ, 215
入れ子の関数, 654
インデックスを表す添字の範囲, 142

エクステント, 141
エスケープ文字, 152
演算子, 150, 162
　アドレスと間接演算子, 174
　関係演算子, 165
　関数型キャスト演算子, 172
　キャスト演算子, 172
　コマンド置換, 176
　コンマ演算子, 162, 173
　三項条件演算子, 162
　算術演算子, 165
　条件演算, 358
　条件演算子, 169
　増分演算子と減分演算子, 175
　代入演算子, 169
　単項演算子, 162, 174
　ビットごとの演算子, 168
演算でのスカラから配列への上位変換, 360

オーバーロード, 150, 343, 655
オブジェクトの記憶期間, 74, 305

カーブフィッティング, 577
外積, 551
回転行列, 602
概要, 262
科学データの解析としての用法, 550
科学データのプロットのための用法, 495
拡張された複素平面, 273
過少決定, 617
過剰決定, 617
型修飾子, 150
可変長配列, 304, 641
空ステートメント, 178
簡易置換, 89
環境変数, 109
　COLUMNS, 29
　DISPLAY, 29
　export, 111
　getenv(), 32, 109
　isenv(), 32, 109
　LINES, 29
　putenv(), 29, 32, 109
　remenv(), 32, 109
　setenv, 111
　その他のシェル, 111
関数, 149, 201, 378
　入れ子にされた関数, 211
　可変長の計算配列を返す関数, 379
　関数間の通信, 242
　関数ファイル, 247
　関数プロトタイプ, 205
　計算配列を返す関数, 378
　固定長の計算配列を返す関数, 378
　再帰関数, 210
　汎用関数, 249
関数の最小値, 560, 579
関数の最大値, 560, 579
関数ファイル, 35, 61
関数プロトタイプのスコープ, 325
不完全配列, 142

キーワード, 42
疑似逆行列, 619
既定の出力書式, 55
既定の入出力形式, 457
起動, 28, 52
逆行列, 619
逆高速 Fourier 変換, 624
共用体, 145, 328, 411
行列, 343
行列解析, 607
行列のトレース, 600
行列のノルム, 566
行列のランク, 601

区切り子, 50
組み込みコマンド, 83
クラス, 81, 144, 328, 415, 642
　private メンバ, 417
　public メンバ, 417
　this, 424
　入れ子にされたクラス, 434
　インスタンス, 418
　コンストラクタ, 418
　スコープ解決演算子, 423

静的メンバ, 421
ポリモーフィズム, 424
メンバ関数, 415
メンバ関数内のクラス, 437
繰り返しステートメント, 182

計算配列, 37, 141, 151, 343, 348, 397, 643
計算配列へのポインタ, 388
形状, 141
形状が完全に指定された配列, 431
形状が完全指定された配列, 362
形状引継ぎ配列, 141, 142, 317, 322, 363, 431, 643
形状引継ぎ配列へのポインタ, 394, 643
形状無指定配列, 141, 308, 366, 643
goto ステートメント, 311
switch ステートメント, 311
構造体と共用体のメンバ, 313
現在のシェル, 60, 69, 75, **78**, 81, 83
検索順序, 66

構造体, 143, 328, 410
構造体または共用体のメンバ, 143
高速 Fourier 変換, 624
コールバック関数, 242
固定長の多次元配列, 300
固有値, 622
コマンドステートメント, 114
コマンド置換, 41, 101
コマンドファイル, 33
コマンド補完, 482
コマンドモード, 30
コマンドラインオプション, 58
コマンドライン引数, 243
コメント, 50
固有ベクトル, 622
コンマ演算子, 162

最小公倍数, 557
最大公約数, 556
三角行列, 604
三項条件演算子, 162
参照, 142, 146, 251
ステートメント内の参照, 251
引数の参照渡し, 253
参照型, 250, 653
参照の配列, 643
参照配列, 369
配列参照, 345
参照呼び出し, 201, 677
参照渡し, 224
三連文字, 41

式ステートメント, 178
式置換, 97
式の置換, 465
式の評価, 108
識別子, 44, 72
事前定義済みの識別子, 44
スコープの規則, 72
名前空間, 74
リンケージ, 73
識別子の名前空間, 74
識別子のリンケージ, 73
字句要素, 41
事前定義済みの識別子, 44
自動記憶期間, 158, 178
集合体の浮動小数点型, 140
出力, 111, 445
条件数, 599
条件付き組み込み, 122
常微分方程式, 592
初期化, 158, 327, 345

随伴行列, 605
スクリプトファイル, 34, 61
スコープ解決演算子, 653
スコープの規則, 72
ステートメント, 178
break, 186
case, 181
continue, 186
else-if, 180
goto, 187
if, 179
if-else, 180
loop, 182
return, 187
switch, 181
空, 178
繰り返しステートメント, 182
式, 178
選択ステートメント, 179
単純ステートメント, 178
複合ステートメント, 178
分岐ステートメント, 186
ラベル付きステートメント, 189
ストリーム, 445

正規直行基, 621
制限, 150, 483
整数定数, 155
静的記憶期間, 75, 158, 178, 305
静的メンバ, 653
セーフ Ch, 480
セーフ Ch シェル, 76
セーフシェル, 53
積分, 595
線形空間, 621
選択ステートメント, 179

相関関係, 565
相互相関, 635

対角行列, 600, 603
代数方程式
　根, 585
多項式, 581
　因数分解, 587
　カーブフィッティング, 578
　行列の特性多項式, 589
　係数の解, 586
　導関数, 583
　評価, 582
畳み込み, 626
単位行列, 603
単項演算子, 162
単純ステートメント, 178

置換, 96, 98
　コマンド置換, 101
　コマンド名置換, 98
　式置換, 97
　式の置換, 465
　ファイル名置換, 99
　変数置換, 96
　変数の置換, 464
逐次出力ブロック, 464
中央値, 563

ツールキット, 488

定数, 150, 151, 157
　複素数, 158
　ポインタ, 158
ディレクトリ操作, 473
ディレクトリの読み込み, 476
ディレクトリを閉じる, 474
ディレクトリを開く, 474
データ解析, 558
データの並べ替え, 568
データ変換規則, 275
デバッグ, 70

統計, 558
トークンから文字列への変換, 126
　# プリプロセッサトークン, 126
特異値分解, 611
特殊行列, 607
　Cauchy, 607
　Chebyshev, 607
　Chow, 607
　Circul, 607
　Clement, 607
　DenavitHartenberg, 607
　DenavitHartenberg2, 607
　Dramadah, 607
　Fiedler, 607
　Frank, 607
　Gear, 607
　Hadamard, 607
　Hankel, 607
　Hilbert, 607
　InverseHilbert, 607
　Magic, 607
　Pascal, 607
　Rosser, 607
　Toeplitz, 607
　Vandermonde, 607
　Wilkinson, 607
ドットコマンド, 75

内積, 555

入出力形式, 447, 463
　集合体データ型, 463
入出力のアンバッファリング, 445
入出力のバッファリング, 445
入力, 445
入力引数, 573, 574

排他, 168
パイプライン, 106
配列, 141, 142, 150, 306, 330, 343, 643
　インデックスを表す添字の範囲, 142
　下限値, 330
　計算配列, 141, 151
　形状引継ぎ配列, 141, 142, 317
　形状無指定配列, 141, 308
　構造体または共用体のメンバ, 143
　固定添字範囲, 330
　参照, 142
　上限値, 330
　配列の宣言, 306
　不完全配列, 142
配列演算, 351
　アドレス演算子, 358, 395
　インクリメント演算, 354
　関係演算, 355
　算術演算, 351
　代入演算子, 354
　デクリメント演算, 354
　論理演算子, 357
配列演算子
　キャスト演算子, 359
配列全体, 345
配列のデータ型変換, 350
配列の要素, 347
配列要素の合計, 561
配列要素の積, 562
配列要素の評価, 571

配列を渡す, 298
  1次元配列を渡す, 298
  多次元配列を渡す, 298
バックグラウンドコマンド, 108
バックグラウンドでのコマンド, 108
パッケージ, 67, 490
パブリック, 653
バランス行列, 622
反転行列, 601
汎用関数, 43, 249, 262, 641, 648
汎用配列関数, 380

引数置換, 94
非線形方程式, 590
左シフト, 168
ビットフィールド, 145, 412
表記規則, 4
標準偏差, 564

ファイル操作, 468
ファイル名置換, 99
ファイル名補完, 482
フィルタリング, 626
複合ステートメント, 178
複素演算, 279
  通常の複素数, 279
  複素メタ数値, 279
複素関数, 291
複素数に関する lvalue, 290
複素数, 35, 158, 272
複素方程式, 557
複素メタ数値, 275, 286
符号関数, 556
符号なし char, 135
符号なし int, 137
符号なし short, 136
浮動小数点, 157
浮動小数点型, 137
プライベート, 653
プリプロセッサディレクティブ, 122
フルバッファリング, 445
プログラムモード, 33
プログラムのオプション, 119, 244
プログラムの起動, 65
プログラムの実行, 64
  複数ファイル, 67
プログラムの終了, 66
プロット, 495
プロットのエクスポート, 521
プロットのズーム, 521
プロンプト, 76
  # アドミニストレータープロンプト, 76
  # スーパーユーザープロンプト, 76
  $ Bourne, Korn, BASH シェルプロンプト, 76
  % C シェルプロンプト, 76
  > Ch プロンプト, 76

平均値, 563
別名, 92, 93
別名の解除, 95
変数置換, 96
変数の置換, 464

ポインタ, 141, 158, 190, 242
  関数へのポインタ, 234
  ポインタ配列, 195
  ポインタへのポインタ, 197, 244
ポインタ演算, 190
包含, 168
ポリモーフィズム, 424
ポリモーフィック
  参照の配列, 426
  汎用数学関数, 425
ポリモーフィックな関数, 426

マクロ拡張で結合するトークン, 127
  ## プリプロセッサトークン, 127
マクロ置換, 124
マシンイプシロン, 267
マルチバイト文字, 152

右シフト, 168

メタ数値, 270, 461
メモリの動的な割り当て, 193
メモリの動的割り当て, 326
メンバ関数, 653
メンバ関数の受け渡し, 439

文字, 399
文字セット, 41
文字定数, 151
文字列リテラル, 154
文字列, 147, 399
文字列長, 644

有限の拡張された複素平面, 273

予約済みのシンボル, 44

ライブラリ, 484
ラインバッファリング, 445
ランク, 141
乱数, 555

リーマン球面, 273
リダイレクト, 103
履歴置換, 86

ループ
  Do-While ループ, 183

索 引

Foreach ループ, 185
For ループ , 184

列挙体, 413
連立一次方程式, 616

ローカル, 150, 215
論理演算子, 168

ワイド文字, 152, 409
ワイド文字列, 155, 409