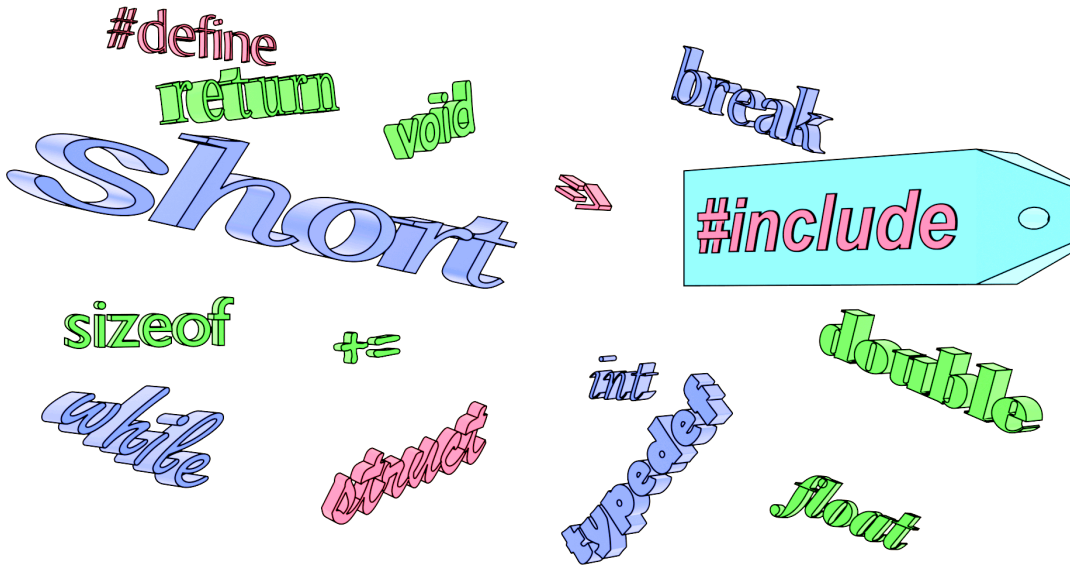


キックオフ C 言語



まえがき

本書は、10年以上にわたるプログラミング実習の授業支援の経験から生まれました。通常の入門書には書かれないようなことまで書いています。

通常のC言語の入門書であれば、文法の説明から始まって、正しいプログラムの書き方を説明してあります。しかし、コンパイルエラーの読み方や、「最初のエラーが重要」との教えまで言及したものはなかなか見つかりません。

授業でつまづいている人に出会うと、コンパイルエラーの意味がわからなかったり、大量のエラーメッセージのどこから対処してよいかわからなかったり、あるいは止めたつもりプロセスが動き続けてファイルを書き込みロックしてコンパイルができなくなっている、などということがよくあります。本書ではそのような人にも役立つよう、実習授業のレジメに書いてありそうな、現実問題への対処方法も記載しました。

またサンプルコードには、役に立たない動作ではなく、単体として見ても意味のある動作をするように心がけました。1つの動作を実現するのにも実装方法は何通りもありますから、段階を踏んで改良していくスタイルを積極的に採用しました。

執筆陣は、著名なオープンソースソフトウェアの開発に携わったり、世界規模のプログラミングコンテストで好成績を修めた人ばかりです。このような経験も生かして、命名の習慣や実際のコーディング現場での現実的な対処方法などにも踏み込んで記述しました。現実のコーディング現場では絶対に使われない、関数ブロック内のプロトタイプ宣言や構造体宣言などは、サンプルコードからも排除しました。

昨今のスマートフォンも含めたコンピュータ環境を鑑みると、いまC言語を学んだからといって、将来もC言語でソフトウェア開発をするとは、必ずしも言えません。したがってC言語に深く精通することを目標とはせず、**コンピュータ言語一般に通用する知識**を身につけることを目標とし、同じような事象にも、言語が違うとどのように取り扱われ方が変わるのか、注釈を加えました。

プログラミング学習の指導法として、「文法を学んだら、後は自分で考えなさい」というものもありますが、本書を執筆しているうちに、それでは上達しない、少なくともこの方針で育てられた人材が社会で活動するには悪影響があると、一層強く感じるようになりました。一人で仙人のように、他人のサンプルも読まずにプログラムを作るのならともか

く、チームでの共同作業を想定すると、語彙はまわりの人と合わせる必要があります。つまりソースコードは、プログラマにとって会話の媒体です。上達の近道は「いろいろな人の（できれば上質の）プログラムをたくさん読む」ことです。多くの人が同じような処理を何度も書くのには、それ相応の理由があるので、理由を完全に理解せずとも、まずはその**慣用句**を覚えるのが、会話を成立させる近道です。独創的な、他人に理解されないすばらしい処理を思いつくよりも重要なことです。これは英語のような、外国語の習得にも似ているように思います。いくら文法上は正しくても、ネイティブならその言い回しはしない、となればそれまでです。考えてわかることではありません。

この考えに至ったのは、執筆中の経験のおかげです。自分ならこう書くけれども、初学者なら違う書き方をするかもしれない、しかし世の中では見たことがない、という処理がいくつもありました。調べてみると、初学者の書きそうなものには問題があって、自分はそこまで理解せずに、**慣用句**を使っていたために問題を避けていたのだとわかりました。つまり、文法だけから正しい書き方にたどりつくには、とても初学者には説明しきれない、膨大な知識が必要だったのです。この時になって、ようやく慣用句のありがたみが理解できました。

本書が、初学者の人にとって、よい慣用句に出会う機会となり、またプログラマとしてのよい心構えを身につけてもらうきっかけとなれば、望外の喜びです。そして本書以外の書物にも触れて、よりよいプログラミング技術を身につけていただければと思います。

本書を執筆するにあたっては、多彩な関係者のご協力をいただきました。学生実験のスタッフ、特に黒澤 隆之さんには、まったく頭が上がりません。イラスト作成には、理工学部 情報科学科 巳波研究室の高木美紀さん、スライド作成には、同研究室の矢野皓己さん、大島由裕さん、犬童梨子さんにご協力いただきました。そして巳波研究室の卒業生の高野歩路さん、大学院生の三柳海渡さん、大崎研究室の南口宙太さん、授業科目（プログラミング実習 I）のチーティングアシスタントの酒井一徳さん、杉本祐貴さん、梅林立さん、数理科学科助教の陰山真矢さんには、数々の助言をいただきました。三田市の飲食店「煮こみやりん。」と「酒楽スタンドにこいち」では、構想を練らせていただいただけでなく、東野弘志さん、東野朝海さん、高田学さんをはじめとする常連客の方にも話題にいただき、また雑談の中からよい例題を思いついたこともありました。この場を借りて厚く御礼申し上げます。

2022 年春
情報・機械系 学生実験準備室

本書の位置づけ

本書の読者は、理系の大学在籍者で、初めて C 言語を学ぶ人を想定しています。高校までの数学の知識を使って説明することがあります。

C 言語の機能を網羅的に紹介するわけではありません。プログラミングというものの考え方を教えるために題材を絞ります。省くものを挙げては仕方ないのですが、例えば以下のものを省いています。

- 代替できる場面の多い switch 文、正しく使うことの難しい goto 文。
- ビット演算、ビットフィールド、共用体、列挙型。
- ポインタの演算、特に配列の添字として以外に加減算。
- 大部分の標準ライブラリ。メモリの動的確保を含む。
- 再帰呼び出し、分割コンパイル。

一方で、教科書として追加している部分も多々あります。それぞれの演習での入出力例を付けていること、また良くあるミスについて述べていることです。

問題分割の習慣を身につけてもらうため、関数を早い段階で説明しました。入門書では触れられることの少ない、(int 型を流用した) 論理型の関数や変数にも言及しました。これは、教えられなければ使い方を誤る代表格の機能ですから、例題や練習問題にも取り入れました。

習得目標は、駐車場の自動清算機を模倣することです。そのために、順次いろいろな操作について述べていきますが、もちろんそれ以外の例も用いて C 言語の機能について触れていきます。

スペース文字の表記

通常のスペースは、文字の間隔が開くだけで、見えない文字ですが、スペースを入力することが重要な場面では、" "のように、下線がついたように表記します。

採用する C 言語規格

C 言語の言語規格 (programming language standard) は、(初期を除くと) 米国国家規格協会 (ANSI) や国際標準化機構 (ISO) によって、約 10 年毎に改訂され、以下の略称で呼ばれています。日本産業規格 (JIS X 3010) にも同等のものが取り込まれています。

K&R (1972 年頃) 旧スタイルの関数定義、関数の引数の型チェックなし

C89/C90 (ANSI C) (ANSI 1989 年, ISO 1990 年) 新スタイルの関数定義、プロトタイプ宣言による引数の型チェック、void 型や列挙型 (enum) の導入

C99 (1999 年 [5]^{*1}) 変数定義がブロックの途中でもよい、1 行コメント (//)、可変長配列、論理型 (bool) や複素数型、インライン関数指示子の導入

C11 (2011 年 [6]) gets() の廃止などによる脆弱性対処、型による分岐 (Generic) の導入、可変長配列と複素数をオプションに格下げ

C23 (2023 年発行予定 [7]) 旧スタイルの関数定義の廃止、2 の補数表現が必須、2 進数リテラル、10 進浮動小数点、POSIX 関数 (str(n)dup, memccpy) の導入

本書では、他の言語との親和性も考慮して、C99 を採用します。昨今のコンパイラは初めからこの文法を受け付けますが、まれにオプション指定が必要な場合があります^{*2}。

想定する開発環境

本書では、テキストエディタとコマンドラインのコンパイラを前提に解説します^{*3}。想定するコンパイラは次のようなものですが、これ以外のものでも大丈夫です。

- GNU Compiler Collection (GCC) :
Windows の Cygwin, MinGW, WSL2, Linux を含む Unix 系
- Clang : Mac の Xcode, Unix 系
- Microsoft Visual C++ : Windows

ターミナルとシェルは、Windows のコマンドプロンプトと cmd.exe では、不自由ですが、何とか使えないこともありません。Cygwin Terminal (mintty) と bash なら、ウィンドウの大きさも自在に変えられますし、コピー&ペーストも便利にできてお薦めです。MinGW では mintty を、WSL2 では wsltty を別途導入したほうがよいでしょう。Mac と Unix 系なら標準のもので十分です。

^{*1} ISO 規格書の正式版は有償ですが、(ほとんど差のない) ドラフト版は無償で公開されています。

^{*2} GCC (Ver. 5 より前) では "gcc_-std=gnu99_source.c" のように -std=gnu99 が必要です。(C A.10 節)

^{*3} 統合開発環境の利便性は高いですが、初学者には、その裏で動いているプリミティブな機構を理解してもらうことも重要だと考えています。

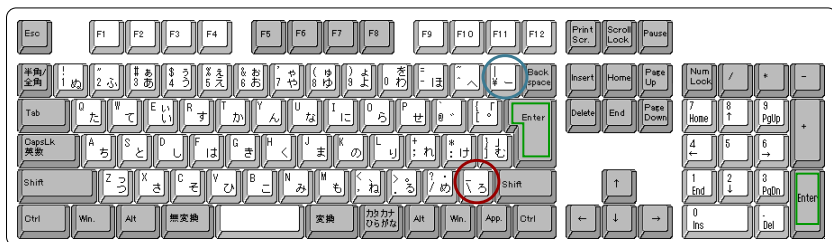
テキストエディタも、どれでも構いませんが、さすがに Windows のメモ帳では機能不足です。C 言語用の入力支援のあるものを選んでください。Microsoft 社の Visual Studio Code は、便利な上に OS を選ばないので、よい選択肢です。

Cygwin のインストールのコツ

詳細な方法には立ち入りませんが、ポイントを列挙します。

- 個人ファイルのフォルダ名にスペースや漢字が含まれていると、Cygwin に限らず、動作のおかしくなるソフトがあります。後から変更すると、様々な機能に影響があって、これも動作不良を引き起こします。アルファベットのみのユーザを新規に作るのが安全です。
- 一般に、インストール作業には管理者権限が必要です。一般ユーザであれば、管理者ユーザでログオンしなおして作業します。さらに、右クリックメニューの「管理者として実行」まで必要な場合もあります。
- (特にサードパーティ製の) ウィルス対策ソフトの誤判定で、インストールに失敗する事例を数多く目にしています。ソフトによっては、報告もせずに Cygwin に必要な DLL を削除します。信頼できるインストーラを使って、「リアルタイム検出」機能は一時的に OFF にします。インストール時間の短縮 (数分の 1) にもなります。
- gcc はデフォルトではインストールされないので、パッケージを追加する必要があります。Devel カテゴリの gcc-core パッケージです。
- Cygwin Terminal の初期フォルダは、個人ファイル置き場 (ドキュメントフォルダ) とは異なり、C:\cygwin64\home\userid のような場所になっています。環境変数の HOME でカスタマイズ可能です。ただし、クラウドサービスのファイルの同期対象、あるいはウィルス検査の対象にすると、コンパイル速度が極端に遅くなることがあるので、場所はよく考えて決める必要があります。
- Cygwin と直接の関係はありませんが、Windows の BitLocker という暗号化の機能が、意図せず起動ドライブで有効になっているのを散見します。この機能は、パソコン本体を紛失した際には情報漏洩の危険を低減しますが、ちょっとしたことで Windows 自体が起動しなくなり、また故障時のデータ救出がほぼ不可能になります。取り扱うデータの機密性が低いなら、無効化を検討すべきでしょう。

図1 日本語キーボード (ODGA109A) のキー配列



バックslash・円記号

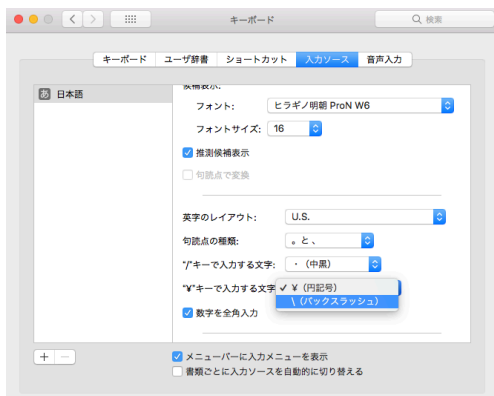
本書では、頻繁に \ (バックslash) を記載しています。この文字は、日本ではこれまで ¥ (円記号) に置き換えて使われてきました。そのため図1のように、日本語キーボードには \ と ¥ の両方のキーがあるのに、日本語 Windows 環境では区別されず、どちらを入力しても円記号が表示されてきました*4。

ところが Unicode が普及して、この2文字が区別できるようになりました。したがって、どちらで入力するかは、環境 (OS だけでなく、アプリや設定) によって異なります。

区別できる場合 バックslashを入力してください。

区別できない場合 どちらで入力しても構いません。円記号が表示されても、バックslashに読み替えてください。

Mac の日本語環境では、2文字を区別するので、バックslashを入力する必要があります。1文字だけなら **option** と同時に **¥** を押せばよいのですが、**¥** で常にバックslashにするには設定変更が必要です。「システム環境設定」→「キーボード」→「入力ソース」→「日本語」と進んで、「¥"キーで入力する文字」をバックslashにしてください。



*4 英語キーボード (101 キーボード/US 配列) には円記号がありませんが、エンターキーの上にあるバックslashを入力しても円記号が表示されたことでしょう。

■ エンターキー・リターンキー

キーボードの刻印に関して、もうひとつ話題があります。エンターキーとリターンキーの違いです。こちらは単純で、キーボードによって「Enter」と「↵」の2通りの刻印があるというだけで、区別の必要はありません。

本書では「Enter」と表記します。よく使われるキーなので、フルキー部分とテンキー部分の2ヶ所に用意されています。形状や位置も特徴的ですから、多くの人が刻印を気にしていないでしょう。

■ 図書の推薦

学習前や学習中に並列して読むと役立ちそうなものを挙げておきます。

- 「C プログラミング入門以前」 [9] は、当たり前としてなかなか説明されないことを、言語化してあります。ざっと目を通すだけでも価値があります。
- 「やさしいC」 [13] は簡潔な説明で、網羅的に内容を取り上げています。
- 「Cの絵本」 [8] は、イラストが多く、特に初学者には読みやすいと思います。
- 「新・明解C言語 入門編 第2版」 [12] は、入門書の位置づけながら、(本書では省略した) 難しい内容も盛り込まれています。C99に準拠した貴重な解説書です。

また、本書を一通り読み終えたら、次のような書物も参考になるでしょう。

- 「C言語による標準アルゴリズム事典」 [11] は、言語の文法の次に学ぶべき、アルゴリズムの数々が簡潔にまとめられています。
- 「プログラミング作法」 [3] は、どのコンピュータ言語にも共通する「良い習慣」を説明した名著です。
- 「組込みソフトウェア開発向けコーディング作法ガイド：C言語版」 [10] は、PDFでも無償で公開されていて、組み込みソフトウェア分野に限らず、参考になります。

言語規格を知るには、次の書物が(難解ですが) 有用です。

- 最初のC言語の本である「プログラミング言語C」 [1] (通称K&R) は、C言語の作者であるカーニハンとリッチーによるC言語の解説書で、通称は二人のイニシャルから来ています。文法からライブラリとして提供される機能の実装例まで、簡潔に説明されているバイブルです。日本語訳の [2] とともにベストセラーですが、残念ながら内容は少々古く、C89で止まっています。
- 「Cリファレンスマニュアル」 [4] は、C99に対応した、言語規格の解説書です。

目次

まえがき	iii
本書の位置づけ	v
スペース文字の表記	v
採用する C 言語規格	vi
想定する開発環境	vi
Cygwin のインストールのコツ	vii
バックスラッシュ・円記号	viii
図書の推薦	ix
第 1 章 プログラムを作る	1
1.1 シェルの操作	2
1.2 C 言語プログラムの実行まで	3
1.3 初めてのプログラム	4
1.3.1 ソースコードの入力	4
1.3.2 コンパイル・実行	5
1.3.3 記号の読み	6
1.3.4 プログラムの説明	6
1.3.5 スペース・改行・タブ	6
1.4 コメント	7
1.5 複数行表示	8
1.5.1 コンソール	8
1.5.2 文字列リテラルと改行文字	8
1.6 簡単な計算	10
1.7 入力と出力	12
1.8 簡単なゲーム	13
第 2 章 型・値・式・変数	15
2.1 定数と変数と型	16

2.2	変数	16
2.2.1	変数定義	16
2.2.2	変数への代入	17
2.2.3	変数の初期化	18
2.2.4	識別子	19
2.3	整数	20
2.3.1	符号付き整数型	20
2.3.2	符号なし整数型	20
2.3.3	整数型の表示方法	21
2.3.4	整数型の値の範囲	21
2.4	浮動小数点数	22
2.4.1	浮動小数点型	22
2.4.2	浮動小数点型の表示方法	22
2.4.3	浮動小数点型の値の範囲と精度	23
2.5	文字	24
2.5.1	文字型	24
2.5.2	文字型の表示方法	24
2.5.3	文字型の値の範囲	25
2.5.4	文字列と文字列定数	26
2.6	式と単純な演算子	27
2.6.1	単項演算子	28
2.6.2	優先順位	28
2.7	変数と複雑な演算子	29
2.7.1	値の入れ替え (スワップ)	29
2.7.2	インクリメント・デクリメント・複合代入	29
2.7.3	型変換 (キャスト)	30
2.8	マクロ定数	31
2.9	型ごとの限界値を探る	32
2.10	練習問題	33
第3章	関数 (1)	35
3.1	関数の入力=引数・関数の出力=戻り値	36
3.2	関数呼び出しと実行順序	38
3.3	関数を作る意義	39
3.4	複数の引数	40
3.5	ブロックと変数の有効範囲 (スコープ)	41

3.6	関数の再利用とプログラムの信頼性	43
3.7	void 型の関数	44
3.8	出力 = return	45
3.9	入力 = 引数	46
3.9.1	仮引数・実引数	46
3.9.2	値渡し・参照渡し	47
3.9.3	引数なし	49
3.10	練習問題	49
第 4 章	条件分岐	51
4.1	条件分岐の if else と論理型	52
4.1.1	多重分岐	53
4.2	if の入れ子と論理演算	54
4.3	論理演算の組合せ	56
4.3.1	ド・モルガンの法則	57
4.4	if を羅列する弊害	58
4.4.1	条件の網羅	58
4.4.2	条件の網羅と関数	58
4.4.3	条件に影響のある操作	59
4.4.4	if の羅列と多重分岐	60
4.5	論理型の変数と関数	62
4.5.1	論理型の定数	63
4.5.2	if に現れる論理型	64
4.6	論理積と論理和の短絡評価（発展的内容）	66
4.7	練習問題	69
第 5 章	繰り返し処理 (1)	71
5.1	while 文による繰り返し	72
5.2	決まった回数の繰り返しとループ変数	74
5.3	for 文による繰り返し	74
5.4	ループ処理の実例	76
5.4.1	偶数を表示	76
5.4.2	個数を数える	78
5.4.3	合計を計算する	80
5.4.4	平方根で 3 乗根を求める（漸化式）	82
5.5	良いループと悪いループ	85
5.6	練習問題	86

第 6 章	繰り返し処理 (2)	87
6.1	2重ループ	88
6.1.1	for のループ変数のスコープ	89
6.2	ループを抜ける・次のループへ切り上げる	90
6.2.1	無限ループ	90
6.2.2	continue の活用例	91
6.3	繰り返し処理にまつわる話題	92
6.3.1	ループ処理が空	92
6.3.2	少なくとも 1 回は実行するループ	92
6.3.3	逆順ループ	93
6.3.4	break したかどうかを後から判定する	93
6.4	より高度なループ処理の実例	95
6.4.1	九九の表	95
6.4.2	コマンドで区切って列挙	96
6.4.3	キーボードから正しい値を受け取るまで繰り返す	97
6.4.4	2重ループの中断	99
6.4.5	素数判定	100
6.5	練習問題	102
第 7 章	関数 (2)	103
7.1	プロトタイプ宣言	104
7.1.1	暗黙的な宣言	106
7.1.2	標準ライブラリ関数のプロトタイプ宣言	107
7.1.3	プロトタイプ宣言における省略と重複	107
7.2	変数のスコープ・変数の寿命	109
7.2.1	ローカル変数・グローバル変数	109
7.2.2	時間的な有効期間=寿命と static	110
7.2.3	空間的な有効範囲=スコープと static	111
7.2.4	static の使用例	112
7.2.5	変数名や関数名の重複	114
7.3	変数や関数の命名規則や習慣	116
7.4	練習問題	117
第 8 章	配列	119
8.1	配列の定義と初期化	120
8.1.1	配列要素の最大値	121
8.2	要素数とマクロ定数	122

8.2.1	初期化子の個数	124
8.3	関数に配列を渡す	125
8.3.1	配列は参照渡し	126
8.4	配列を順番に操作する	127
8.4.1	配列の最大値を返す関数	127
8.4.2	0 始まり・1 始まり	128
8.4.3	配列の正順・逆順コピー	129
8.5	配列をランダムに操作する	130
8.5.1	エラトステネスのふるい	130
8.5.2	度数分布 (ヒストグラム)	132
8.5.3	覆面算 (発展的内容)	134
8.6	2次元配列	135
8.6.1	多次元配列	136
8.6.2	2次元配列の縦横合計	137
8.7	練習問題	138
第9章	文字列とポインタ	139
9.1	文字列=char の配列	140
9.2	文字列操作の標準ライブラリ関数	141
9.2.1	文字列の長さ	142
9.2.2	文字列のコピー	144
9.2.3	文字列の連結	146
9.2.4	書式変換	148
9.2.5	文字列の比較	150
9.3	ポインタ	151
9.3.1	ポインタを扱う演算子	151
9.3.2	配列とポインタの関係	152
9.3.3	NULL ポインタ	152
9.4	文字列の配列	153
9.5	練習問題	154
第10章	構造体	155
10.1	構造体	156
10.1.1	構造体の使用例	158
10.1.2	構造体の代入・引数・戻り値	160
10.1.3	構造体のスコープ	160
10.1.4	構造体のアライメント・パディング (発展的内容)	161

10.2	基本データ型・ユーザ定義型	162
10.2.1	typedef と struct の組合せ	163
10.3	構造体の配列	164
10.4	型にまつわる命名の習慣	165
10.5	例題・分数計算	166
10.6	練習問題	168
第 11 章	ファイルの読み書き	169
11.1	ファイルとは	170
11.2	プログラムからの入出力	171
11.2.1	オープン（前処理）	172
11.2.2	読み書き（主処理）	173
11.2.3	クローズ（後処理）	174
11.2.4	プログラムの中断（エラー処理）	175
11.3	ファイルの使用例	176
11.3.1	書き込み	176
11.3.2	文字単位の読み取り	178
11.3.3	行単位の読み取り	180
11.3.4	書式つきの読み取り	183
11.3.5	終端検査、エラー検査	184
11.3.6	2つのファイルを同時に読み書き	185
11.4	オープンするファイルを実行時に決める	186
11.5	現実的なファイルオープン	187
11.6	練習問題	188
第 12 章	最終目標	189
12.1	規定課題	190
12.1.1	部品の作成	191
12.1.2	部品の結合	193
12.2	自由課題	196
第 13 章	さらなる成長に向けて	197
13.1	プログラミング環境の上手な操作	198
13.2	効率のよいデバッグ術	199
13.3	コーディング上の良い習慣	199
13.4	プログラムの上達へむけて	200
13.5	値の変化の頻度に応じた扱い	200

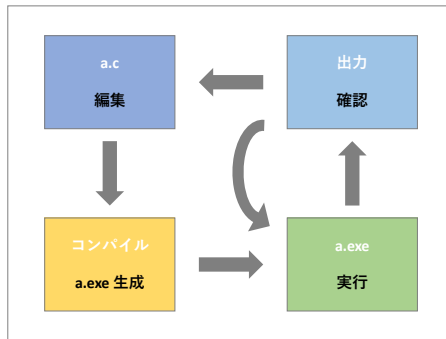
13.6	開発のための技術・ツール	201
13.7	浮動小数点型の性質	202
13.7.1	丸め誤差	202
13.7.2	情報落ち・桁落ち	203
13.7.3	無限大・非数	203
13.7.4	定数	204
13.8	入出力インタフェース	205
13.8.1	コマンドライン引数	205
13.8.2	キーボード	206
13.8.3	標準入出力・リダイレクト・フィルタ	207
13.9	疑似乱数	209
13.10	関数の引数の可変長	210
付録 A	一覧表	211
A.1	記号の読み	211
A.2	ASCII コード	212
A.3	エスケープシーケンス	213
A.4	EBCDIC コード	214
A.5	演算子の種類	215
A.6	ユーティリティ関数 <stdlib.h>	216
A.7	数学関数 <math.h>	217
A.8	printf()/scanf() の書式文字列	218
A.9	予約語	219
A.10	コンパイルオプション	219
参考文献		221
ソースコード一覧		222
索引		224

第 1 章

プログラムを作る

本章では、簡単なシェルの操作と、初めての **C** 言語によるプログラムを作ります。また、最後に簡単なゲームを頑張って入力してみましょう。

プログラミングをする上で、重要なものとして**開発環境**があります。作業の流れは、大まかに以下の図のようになります。



これらの作業を効率よく行える「統合開発環境」もあって、利便性は高いのですが、まずはこれらの作業を別々に実行することで、それぞれの役割を理解しましょう。

キーワード

- ソースコード, コンパイル
- スタンダード・アイ・オー
- main() 関数, ブロック, コメント
- エスケープシーケンス, 改行文字

1.1 シェルの操作

Cygwin Terminal を開いて、次のようなコマンドを入力してみましょう。黄色地の文字はキーボードから入力することを示します。Windows や Mac では、ファイルをグループ分けした入れ物をフォルダ (folder) と呼びますが、元々は Unix 系でディレクトリ (directory) と呼んでいるものです。コマンドにも `directory` のキーワードが入っています。

`ls` : (list) ファイル名の一覧を表示します。

- `ls` `[Enter]` は現在のフォルダのファイル名を表示します。
- `ls -l` `[Enter]` はファイルの作成日などの情報も表示します。

`pwd` : (print working directory) `pwd` `[Enter]` は現在のフォルダを表示します。

`cd` : (change directory) フォルダを移動します。

- `cd hoge` `[Enter]` のように、フォルダ名を指定すると、そのフォルダに移動します。1 階層下に移動することになります。
- `cd ..` `[Enter]` は 1 階層上のフォルダに移動します。
- `cd` `[Enter]` はホームディレクトリ^{*1}へ戻ります。
- `cd` を入力してから、ファイルマネージャのフォルダアイコンを Cygwin Terminal のウィンドウにドラッグするとフルパス名が入力されるので、続けて `[Enter]` を押すだけで、一度に目的のフォルダに移動できます。

`mkdir` : (make directory) フォルダを作ります。

- `mkdir hoge` `[Enter]` は hoge というフォルダを作ります。

`history` : (history) `history` `[Enter]` は最近実行したコマンドを表示します。

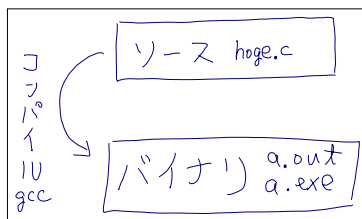
`cat` : (concatenate) ファイルの内容を表示します。

- `cat hoge.txt` `[Enter]` は hoge.txt というファイルの内容を表示します。

このように、人間が対話的に操作して、プログラムを起動するためのプログラムをシェル (shell) あるいはコマンドインタプリタ (command interpreter) といいます。シェルにはいくつもの種類があって、Cygwin Terminal の場合は `bash` というプログラムが動いています。

ファイルマネージャ (Windows の Explorer や Mac の Finder) は、シェルの GUI 版です。コマンド版 (`bash`) との相互作用も理解しておいてください。片方で作ったファイルが、もう片方でも出現することを確かめておきましょう。

^{*1} シェルを最初に起動したときにいるフォルダのことです。環境変数の `HOME` でカスタマイズできます。



1.2 C 言語プログラムの実行まで

C 言語では、人間の入力したプログラムを直接実行するのではなく、実行形式に翻訳してから実行するという、2 段階の構成になっています*2。

この翻訳方式を採用する言語では、元になるプログラムをソースコード (source code)、翻訳作業をコンパイル (compile)、コンパイルするプログラムをコンパイラ (compiler) と呼びます。(本書で想定するコンパイラは [vi](#) ページ)

ソースコードを保存したファイルはソースファイル (source file) といいます。C 言語のソースファイルは、拡張子を ".c" にします。生成される実行ファイル (executable file) のファイル名は、Unix 系では "a.out"、Cygwin では "a.exe"*3 が既定値です。Windows の Visual Studio の cl などでは、ソースファイルの拡張子を ".exe" に変えたものです。

呼び名を短くして、ソースコードをソース (source)、実行ファイルをバイナリ (binary)*4ということもあります。

1. ソースコード (*.c) を編集
2. コンパイル (a.out や *.exe を生成)
3. 実行
4. 実行結果の確認

思い通りの動作をするプログラムになるまで、この 4 つの作業を何度も繰り返します。

_____ Windows のファイルマネージャの初期設定では拡張子が表示されないの
で、この作業に不都合です。設定を変更して、拡張子を表示させましょう。_____

*2 プログラムを翻訳せずに実行するインタプリタ (逐次解釈) 方式に比べて、時間をかけて実行ファイルを生成できるので、実行効率がよいと謳われています。

*3 Windows では、拡張子の ".exe", ".com", ".bat" が実行ファイルの目印になっています。

*4 バイナリの直訳は「2 進数の」ですが、ここでは「CPU が直接理解できる機械語の」という意味です。

1.3 初めてのプログラム

では、ソースコード 1.1 をテキストエディタで入力してみましょう。ファイル名は "hello.c" としてください。

ソースコード 1.1 初めてのプログラム (hello.c)

```
1 #include <stdio.h>
2
3 int main(void) {
4     printf("Hello, World!\n");
5     return 0;
6 }
```

インデント

1.3.1 ソースコードの入力

ソースコードに使える文字は限られていて、アルファベットと数字や記号、正確には **ASCII 文字** (☞ A.2 節) です。アルファベットの大文字と小文字は、区別します。バックslash (\) とエンターキーの注意は [viii](#) ページを参照してください。

空白 (スペース (space)) の空き具合は、最初はサンプルを真似してください。

- スペースも ASCII 文字にします。いわゆる「全角スペース」はエラーになります。エディタによっては、スペース文字を視覚的に表示する機能があります。
- 行末には、スペース文字をたくさん並べて次の行まで送るのではなく、`Enter` キー (↵) で改行文字 (newline character) を入力します。改行文字もエディタによっては特殊文字として表示できます。
- **行頭にある隙間** を字下げとか **インデント (indent)** といいます。インデントするには `Tab` キーを押します。1 文字で広い幅の開く **タブ文字 (tab character)** が入力されるか、複数のスペース文字に展開されるか、どちらになっても構いませんが、`Tab` キー 1 度だけで適当な (4 文字ほどの) 空き具合にならなければ、テキストエディタの設定を見直してください。拡張子によって動作が切り替わるので、最初に「名前をつけて保存」する必要があります。

逆に、文字の大きさや色、書体は自由に選んでもらって構いません。たいていのテキストエディタには、予約語 (特別な役目のキーワード) に色をつける機能があります。等幅フォントにしておく、文字数が数えやすくてよさそうに思います*5。

*5 英語圏の中には、ソースコード上でも「等幅よりもプロポーショナルフォントのほうが読みやすい」という人もいます。縦に揃えるという概念がないのかもしれませんが。

1.3.2 コンパイル・実行

ソースコードが入力できたら、コンパイルしましょう。GNU Compiler Collection (GCC) を例にすると、C 言語コンパイラのコマンドは "gcc" です*6。シェル上で `gcc hello.c` `Enter` と入力します。

エラーがなければ、`ls` `Enter` (Windows のコマンドプロンプトなら `dir` `Enter`) で実行ファイルが生成されたことを確かめます。

次は実行です。生成された実行ファイルを指定します。

Unix 系・Mac `./a.out` `Enter`

Cygwin `./a.exe` `Enter` あるいは `./a` `Enter`

Windows `hello.exe` `Enter` あるいは `hello` `Enter`

いずれでも、「Hello, World!」の文字が表示されたら成功です。

```
[Cygwinのgccの場合]
$ gcc hello.c

$ ls
a.exe      hello.c

$ ./a.exe      (Unix系なら ./a.out)
Hello, World!
```

```
[WindowsのVisual Studioの場合]
C:\Users\taro>cl hello.c

C:\Users\taro>dir /w
hello.c hello.exe

C:\Users\taro>hello.exe
Hello, World!
```

- 先頭に "./" が必要なのは、シェルの制限 (セキュリティ上の安全策) のためです。
- Windows や Cygwin では、拡張子部分の ".exe" は省略可能です。

コラム：エラーメッセージの読み方 (1)

コンパイル時にエラーが出た場合は、**最初のエラー**が重要です。メッセージがスクロールして画面の外に流れていっても、戻して先頭部分を読みましょう。

ソースファイル名と行番号が書いてありますので、まず場所を特定しましょう。

「エラー: expected ';;' before 'int' 'int' の前にセミコロン (;) が抜けていることを指摘していますが、実際には直前の行末を指していることがあります。

「エラー: プログラム内に逸脱した '\357' があります」文字コードの入ったメッセージが3回連続したら、マルチバイト文字の混入を疑いましょう。

📖 12 ページに続く

*6 Mac の Xcode にも gcc コマンドがありますが、实体は Clang になっています。明示的に "clang" コマンドを使っても構いません。Windows の Visual Studio なら "cl" コマンドです。

1.3.3 記号の読み

C言語ですぐに必要な記号の読みをまとめました。必ずすべて読めるようになってください。もっと多くの記号はA.1節にまとめました。

.	ピリオド, ドット	*	アスタリスク
,	コンマ, カンマ	/	スラッシュ
:	コロソ	\	バックスラッシュ, 逆スラッシュ
;	セミコロン	'	シングルクォーテーション, 一重引用符
&	アンパサンド, アンド記号	"	ダブルクォーテーション, 二重引用符
	バーティカルライン, 縦棒, パイプ	_	アンダーバー, アンダースコア, 下線

1.3.4 プログラムの説明

- `#include` は外部ファイルを読み込む命令です。意味は7.1.2項で説明します。
- `<stdio.h>` はファイル名で、標準入出力 (standard input/output) の意味です。発音は「スタンダード・アイ・オー」です。「スタジオ」とは意味も綴りも違います。
- 最初に `main()` 関数が実行されると決まっています。
- `printf()` は画面に文字を表示する関数です。
- 1つの文 (statement) の終わりはセミコロン (;) です。
- { から } まではブロック (block) と呼ばれる、プログラム上の重要な要素です。開始と終了の対応も重要で、どちらが抜けてもコンパイルエラーが大量発生します。
- “~” で囲まれた部分は、文字列定数あるいは文字列リテラルといいます。そのままメッセージとして扱われて、記号や漢字を含めてもエラーになりません^{*7}。

1.3.5 スペース・改行・タブ

C言語のソース上のスペース文字は、何文字並べても、あるいは改行文字やタブ文字に置き換えても、文法上は同じ意味になるところが多くあります。人間にとっては、適切なスペーシング (spacing) (空白をどれだけ空けるか) でプログラムが読みやすくなるので、工夫しましょう。行頭のインデントは、これから特に重要になります。

コンマ (,) とセミコロン (;) の後ろには、スペースを1個入れるのが習慣です。(行末では不要です。) これは英文タイプライターの規則とも合致します。

^{*7} マルチバイト文字が入ってもエラーにはなりませんし、開発環境と実行環境が首尾一貫していれば、何とか化けずに表示されるでしょう。ただし Shift_JIS はいくつかの文字がおかしくなるので、コンパイラに対策が必要です。なお Java では、ソースコードの文字エンコードを指定できて、実行環境との整合性を言語が保証します。

```
{ ブロック }  
/* コメント */  
" 文字列リテラル "
```

1.4 コメント

プログラムを作る上で、メモを残したいこともあるでしょう。作成日とか、参考にした URL とか、プログラムの動作自体を書き留めると有益です。

ソースコード上の**コメント (comment)** とは、プログラムとしては何の動作もしない部分です。コメントには、このようなメモを残せます。C 言語では 2 つの形式があります。

複数行コメント /* から */ まで (例: /* ここはコメント */)

1行コメント // から行末まで (例: // ここはコメント)

複数行コメントは入れ子にはできず、コメント開始の /* が何回あっても、1 回の */ でコメントが終了します。

1 行コメントは、C99 で正式に言語規格に取り入れられました。C++ で先に採用されていたこともあり、コンパイラの独自拡張でサポートされていた期間も長くありました。

本書のサンプルコード中でも、説明のためにコメントを使っています。コンパイルエラーになる部分は、コメントにして実行できないことを表します。

コラム：コメント

コメントはプログラム上の非常に重要な機能です。メモを残すことはもちろんですが、一時的に動作をやめてみるのにも、コメントが活躍します。

ちなみに、プログラムの一部をコメントにすることを、日本語でも「コメントアウト」(comment out) と、英語表現をそのまま使う人もいます。(comment out は動作です。出来上がったコメントを「コメントアウト」と呼ぶのは珍妙です。) 逆にコメントをやめることは、英語では uncomment といいますが、なぜか日本語で「アンコメント」という人は少ないです。

1.5 複数行表示

では次に、表示するメッセージを2行に増やしてみましょう。それには、コンソール上で改行の起こる仕組みを理解しておく必要があります。

1.5.1 コンソール

プログラムを実行すると、文字が表示されますが、正確には仮想ターミナル（ターミナルエミュレータ）というアプリケーションのウィンドウに表示されます。コンピュータの黎明期には、本当に文字だけのやりとりをする機器（キーボードとモニタ）があって、**コンソール** (console) と呼ばれていました。今ではそれをソフトウェアで実現しています。つまり、C言語のプログラムから見ると、相変わらずコンソールを通じて文字のやりとりをしているのですが、我々から見れば仮想ターミナルのアプリケーション上で操作しています。このため、例えばマウスの操作はC言語プログラムへの指示になりません。

コンソールの文字出力には癖があって、タイプライターのような動作をします。つまり1文字表示すると、次に表示する場所は右にずれます。右端まで到達すると、行が進んで、表示位置が左端に戻ります。この動作を改行といいます。行の途中で改行動作をさせるための特殊文字が**改行文字** (line feed character) です。

1.5.2 文字列リテラルと改行文字

ダブルコーテーションで囲われた"~"の文字列リテラルには、たいていの文字を書くことができると説明しましたが、ソースコード上の1行で完結する必要があります。そのため、改行文字を含めるには、特殊文字の記法である**エスケープシーケンス** (escape sequence) を用いて `\n` と表記します。(☞A.3節)





```
/* エラーになる例 */  
printf("Hello, World!  
"); // 途中で改行してはいけない
```

```
/* 正しい例 */  
printf("Hello, World!\n");  
// ここで改行が起こる↑
```

この改行文字 (`\n`) の役目をよく理解しましょう。ソースコード上で2行に分かれていることと、実行してコンソールに2行表示されることは無関係で、`\n` を何回表示しているかが重要です。次の2つの例は、どちらも同じ1行を表示します。

```
printf("Hello, World!\n");
```

```
printf("Hello, ");  
printf("World!\n");
```


タイプライターは 
 左上から右へと 
 そして下へと 
 タイプする 

複数行表示する方法はいくつもあります^{*8}。以下の例は、すべて同じ動作をします。よく使われるのは、最初の (A) と最後の (E) です。

- (A) `\n` を 1 回含む `printf()` を繰り返します。
- (B) 1 行で `printf()` を繰り返します。ソースコード上の改行は、スペース文字と同じ役割です。ソースコードが横長になって読みにくくなります。
- (C) `printf()` は 1 回で、文字列リテラル中に改行文字を何度も登場させます。
- (D) 文字列リテラルを分割します。少し驚きですが、**連続する文字列リテラルは連結される**ので、このようなことができます。
- (E) 分割した文字列リテラルを複数行に振り分けます。ソースコード上のスペースと改行は同じ意味なので、このようなことができます。これが一番読みやすいでしょう。

```

/* (A) printf() を 2 回 */
printf("Hello, World!\n");
printf("Hello, World!\n");

/* (B) 1 行で printf() を 2 回 */
printf("Hello, World!\n"); printf("Hello, World!\n");

/* (C) 1 行で \n を 2 回 */
printf("Hello, World!\nHello, World!\n");

/* (D) 文字列リテラルを 2 分割 */
printf("Hello, World!\n" "Hello, World!\n");

/* (E) 分割した文字列リテラルを 2 行に */
printf("Hello, World!\n"
      "Hello, World!\n");

```

^{*8} 他の言語に目を向けると、たいていのスクリプト言語では、ヒアドキュメント (here document) という機能で、文字列を複数行にわたって羅列できます。Java では文字列を簡単に連結できます。

1.6 簡単な計算

コンピュータは計算機とも呼ばれるくらいですから、計算は得意です。簡単な計算をさせてみましょう。詳しい文法は次章以降で説明しますので、まずはソースコード 1.2 の通りに入力してください。実行すると「6」の数値が表示されるはずですが、`printf()` は、`%d` をそのまま表示するのではなく、コンマの後の計算式の値に置き換えて表示します。

ソースコード 1.2 簡単な計算

```
1 #include <stdio.h>
2
3 int main(void) {
4     printf("%d\n", 1 + 2 + 3); // 6
5     return 0;
6 }
```

次はソースコード 1.3 で変数を使ってみましょう。`int` というのは、その後ろの変数が整数を格納することを示します。3つの変数 `a`, `b`, `c` を作って、値を表示してみました。

ソースコード 1.3 簡単な変数

```
1 #include <stdio.h>
2
3 int main(void) {
4     int a = 1;
5     int b = a + 2; // 1 + 2 = 3
6     int c = b + 3; // 3 + 3 = 6
7     printf("a=%d, b=%d, c=%d\n", a, b, c); // a=1, b=3, c=6
8     return 0;
9 }
```

`printf()` に複数の `%d` があれば、コンマで区切られた値を順番に使います。ここでは3つの値があるので、表示は `a=1` のように区別がつくよう工夫してみました。

ソースコード 1.3 の実行結果

```
a=1, b=3, c=6
```

このプログラムを改造して、数値を変えたり、変数を増やしたりしてみましょう。足し算を掛け算に変えて、階乗を求めるのもよいでしょう。(引き算はもちろん - です。掛け算の \times はキーボードにないので、`*` で代用します。割り算の \div も `/` で代用しますが、みなさんの想像する動作とは異なるかもしれません。)

ここまでは整数の計算をしてきましたが、次は小数の計算です。ソースコード 1.4 では、行列 $\begin{pmatrix} a & b \\ c & d \end{pmatrix}$ と、その逆行列 $\frac{1}{ad-bc} \begin{pmatrix} d & -b \\ -c & a \end{pmatrix}$ を表示しています。

ソースコード 1.4 簡単な行列計算

```

1 #include <stdio.h>
2
3 int main(void) {
4     double a = 1.0;
5     double b = 2.0;
6     double c = 3.0;
7     double d = 4.0;
8     double t = a * d - b * c; // 行列式
9     printf("%f %f -> %f %f\n", a, b, d/t, -b/t);
10    printf("%f %f    %f %f\n", c, d, -c/t, a/t);
11    return 0;
12 }
```

`double` というキーワードは、名前からは想像しにくいのですが、変数が小数を格納するという指示です。数値にも小数点をつけて `1.0` のようにします。`printf()` での表示には `%f` を使います。行列式の $ad-bc$ は何度も使うので、変数 `t` を作って代入しています。

ソースコード 1.4 の実行結果

```

1.000000 2.000000 -> -2.000000 1.000000
3.000000 4.000000    1.500000 -0.500000
```

これもいろいろ改造してみてください。特に $\begin{pmatrix} 1 & 2 \\ 2 & 4 \end{pmatrix}$ のように、逆行列の存在しないときに何が起こるのか、試しておきましょう。

コラム：コンパイラはどうやって作られるか

C 言語のコンパイラも、実は C 言語のプログラムとして作られているものがあります。特に GCC は、動作チェックも兼ねて、`gcc` のソースコードを `gcc` コマンドでコンパイルします。

1.7 入力と出力

プログラムに限った話ではないのですが、物事の動作というのは、1) 何かを受け取り、2) 処理をして、3) 結果を出す、というものに分かれています。例えば、料理であれば、1) 材料を準備し、2) 切ったり焼いたりして、3) 皿に盛り付けると完成、という流れでしょう。もちろん実際にはこれらは様々な順序で行われています。自動販売機を例にすれば、1) お金を受け取り、2) お金の種類から受け取った金額を変更し、3) ボタンを光らせます。そして、1) ボタンを押してもらい、2) おつりを計算して、3) 品物とおつりを出す、という様々な作業が入ります。

このように、動作を記述するには、処理だけでなく、受け取ることと、結果を出すことが必要です。プログラムでは、これらをそれぞれ「入力」と「出力」といいます。

コラム：コンパイル時のエラーと警告

コンパイル時にエラーが出ると、実行ファイルは生成されませんので、無理に実行することもできません。どうしてもソースコードを修正する必要があります。

警告の場合は、実行ファイルは生成されているので、とりあえず実行できます。潜在的な大きな問題を指摘されている場合もありますが、正常に実行できるくらいなら、簡単に修正できますので、警告もなくすように心がけましょう。

コラム：エラーメッセージの読み方 (2)

☞ 5 ページより続く

エラーメッセージには専門用語の英単語や、中途半端な翻訳が混じっていてわかりにくいですが、英単語は辞書を引くなどして徐々に覚えましょう。おかしな訳語でも、メッセージをそのまま検索エンジンで調べると、ズバリの原因を解説したページに行き着くこともあります。

「警告: 関数 **'printf'** の暗黙的な宣言です」関数名を間違えると、文法エラーではないので、このようにメッセージがわかりにくくなります。(☞ 7.1.1 項)

「**'WinMain'** に対する定義されていない参照です」main 関数の綴りを間違えると、(特に Cygwin では、このように) さらに不可解なエラーになります。

「**Device or resource busy**」文法上の間違いではなく、実行ファイルの生成に失敗しています。(☞ 73 ページの頻出ミス)

1.8 簡単なゲーム

本章の最後として、簡単なゲームを作ってみます。といっても、今回はまだC言語について何も説明していないので、単に頑張って打ち、エラーがないようにするだけです。これまでの二つの例から分かるとおり、こういったソースコードを作る際には、打ち間違い、空白の有無などに気をつけないといけません。

ソースコード 1.5 丁半プログラム（丁か半かを0か1で指定する）

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <time.h>
4
5 int main(void) {
6     srand(time(NULL));
7     int i = rand() % 2;
8     int d = -1;
9     printf("0か1か => ");
10    scanf("%d", &d);
11    if (d != 0 && d != 1) {
12        printf("0か1を入れてください\n");
13    } else if (i == d) {
14        printf("当たり!\n");
15    } else {
16        printf("外れ!\n");
17    }
18    return 0;
19 }
```

では実行してみしましょう。先ほどと同じようにコンパイルして、実行します。全体の動作例は次のようになるでしょう。赤の斜体の文字は、キーボードから入力した文字です。

丁半プログラムの実行例（最初の二回の実行はこのようになるとは限りません）

```
$ gcc -Wall 01-chohan.c
$ ./a
0か1か => 1
当たり！

$ ./a
0か1か => 1
外れ！

$ ./a
0か1か => a
0か1を入れてください
```

10回くらい適当に0か1を入れれば1回くらいは当たりが出ると思います*9。また、0か1以外を入れた場合、0か1を入れるようにメッセージを出します。

本当に0か1以外だとメッセージを出すのでしょうか？いくつか試してみましたか？どんなものを試してみましたか？

実は、このプログラムには一つ欠点があります。試しに、0aと入れてみてください。0でも1でもないはずですが、これは0を入れたかのように動きます。また、01や00のように、あまり一般的でない書き方をするものも問題なく動きます。05はだめといわれます。

さて、皆さんはこれを大丈夫と思いますか？それともまずいと思いますか？

実際のところ、これがよいのか悪いのかは、簡単には決定できません。ある側面では、0や1に相当するものが入力されているので00や01は問題ないとも考えられませんが、0aは最初にあるのが0だから大丈夫とも考えられます。もちろん逆も考えられて、0や1を入力してほしいのだから00や01は違うし、0aにいたってはまったく違う文字が混じっているのだからまずい、ともいえます。このように、プログラムの入力のよしあしはプログラムをどのような環境で使うのか、つまりどのような入力を想定しているのかによっても変わります。そのため、この問題についてはこの本では取り扱うことができません。代わりに、なぜこうなるのか、を皆さんが理解できるようにすることをこの本では重視します。

問題を回避するために、この本ではほとんどの課題で、入出力の形式を可能な限り明示的に限定します。たとえば、ソースコード 1.5 では、入力を0だけ、1だけ、0または1以外で始まる入力の三種類を想定すること、という指示を書きます。

コラム：疑似乱数

ソースコード 1.5 の7行目の「`rand() % 2`」が0か1かどちらかの疑似乱数を生成しています。詳細は 13.9 節で説明します。

本物の乱数なら、次の値の予測がまったくできなくて、二度と再現できません。ところがプログラムで生成する乱数は、(わかりにくい)規則で作りに出しているもので、再現可能です。この点で、本物ではない「疑似」の乱数というわけです。プログラムの開発(間違い探し)には都合のよい性質でもあります。

*9 おおよそ正解が1/2だと思うと10回連続で外す可能性は $1/2^{10} \approx 0.1\%$ ですから、まあ大丈夫でしょう。

第2章

型・値・式・変数

駐車場の自動清算機は、どのように車を管理しているのでしょうか。駐車開始時刻から経過時間を知って、駐車料金を算出しているはずですが。計算機のプログラムとして模倣するためには、これらをプログラミングによって実現せねばなりません。

計算機には、数値、もっと言えば0か1しかわかりません。しかし、0と1の列をどのように解釈するかによって、多種のデータを表現することができます。ここでは、その始まりとして、数値や文字がどのように表現されているか見ることにします。また、可変な値としての変数や、データが何を表すのかを意味する**型**についても見ていきます。

キーワード

- 型, 定数, 変数
- 整数, 浮動小数点数, 文字
- int, double, char
- スワップ
- キャスト
- 処理系依存

表 2.1 よく使われる型

型名	表すもの	定数の例
int	整数	256 0x100
double	浮動小数点数	365.24 3.6524e+2
char	文字 (1 文字)	'A' '#' '0'
char*	文字列	"Aa0#" "256" "A"
void	値なし	—

2.1 定数と変数と型

プログラムで扱う値は、次の 2 通りに大別されます。

定数^{*1} ソースコード上に、直接記述した値です。その名の通り、変化しない値です。

変数 値を保存する入れ物です。代入したときに、保存されている値が変化します。

このような値には、**型 (type)** と呼ばれる種類があります。型ごとに表せるものが決まっています、整数、小数、あるいは文字などがあります。表 2.1 のものがよく使われるので、最初に覚えましょう。既に 1.6 節では、int と double を使いました。

ソースコード上の定数は、自動的に型が決まります。例えば、「10」は int 型ですし、「1.23」は double 型です。変数の型は、int などの**型名**で指定します。

2.2 変数

変数は、値を保存する**箱**のようなものです。数学の変数と似ていますが、値の変化するタイミングや、名前のつけ方も違います。詳しく見ていきましょう。

2.2.1 変数定義

変数は、**定義 (definition)**^{*2}してから使います。定義には、型と変数名を明記します。型が同じなら、複数の変数をまとめて定義できて、変数名をコンマで区切って書き並べます。

```
/* 文法 */
型名 変数名;
型名 変数名1, 変数名2, ...;
```

```
/* 実例 */
int a;
double x, y; // 同じ型をまとめて
```

*1 読みは「ていすう」と「じょうすう」の両方があります。

*2 定義ではなく、メモリ配置を伴わない**宣言 (declaration)**の用語を使う書籍もあります。分割コンパイラでは区別が必要ですが、(本書の守備範囲のように)単一ソースファイルでは、実質的に差はありません。

2.2.2 変数への代入

変数に値を代入するには、**代入演算子 (assignment operator) =** を使います。変数は何度でも代入できて、最後に代入した値一つだけを保持します。古い値は、新しい値を代入した瞬間に忘れてしまいます。

```
/* 文法 */
変数名 = 式;
```

```
/* 実例 */
int a, b;
a = 1;
b = a + 2; // b = 1 + 2
// a + 2 = b; // コンパイルエラー
```

$a = b$ と $b = a$ は、数学では同じ意味のこともあります。プログラムでは違います。= の右辺と左辺で役割が異なっていて、右辺の**式の値**を計算して、左辺の**変数に格納**します。格納先の左辺は、普通の計算式ではなく**単一の変数**である必要があって、**左辺値 (lvalue)** という造語で呼ぶほど重要な概念です。右上の例でわかるように、「a」や「b」は左辺値ですが、「a+2」は左辺値ではないので、代入できません。

= の**両辺の型は一致**させるのを基本としてください。C 言語は、一致していないと（おせっかいなことに）自動変換してしまうので、かえって思わぬ動作不良を引き起こします。もちろん、変換できなければコンパイルエラーになります。

右辺の「a+2」のように、計算式に変数が現れると、保存しておいた値を取り出します。この操作を「**変数を参照する (refer)**」とも「**変数にアクセスする (access)**」ともいいます。

よくある誤解に、= で関係式（恒等式）を定義していると思う人がいますが、そうではありません。C 言語では、その瞬間の式の値を格納しているに過ぎないので、代入した変数は、次に代入するまで値が変化しないことに注意してください。

コラム：変数定義の場所

古い C 言語規格 (C89 まで) では、変数定義の行える場所に強い制約があり、ブロック (☞ 3.5 節) の開始直後に限られていました。そのブロックで使う変数の一覧になる利点があるものの、変数を使用する場所が離れる欠点もあり、初期化忘

れの一因にもなっていました。

C99 からこの制限は撤廃され、ブロックの途中でも変数定義が行えます。これで C++ や Java など、多くの言語と同じになりました。同じ変数を使いまわす必要がなくなり、初期化忘れの検出にも役立ちます。

残念なことに、C 言語の入門書は古いスタイルのままのものが多く、本書は数少ない C99 準拠の入門書となっています。

2.2.3 変数の初期化

変数の値は、多くの場面で、代入するまでどんな値になっているのか保証がありません^{*3}。代入されていない変数の状態を**未初期化 (uninitialized)** とか**代入忘れ**、入っている値は**不定 (indeterminate)** だとも、俗に**ゴミ**ともいいます。代入したつもりで、忘れたままゴミの値を使っていると、動作がおかしくなります。しかも、ゴミのはずの値が、特定の条件で 0 になることもあって、代入忘れに気づきにくいので要注意です。

代入忘れを防ぐために、様々な工夫がされています^{*4}。ここで紹介する、**変数の定義と同時に代入する初期化 (initialize)** の構文も有用です。

```
int a; // 変数定義
a = 5; // 通常の代入
```

```
/* 初期化つき */
int a = 5;
```

型が同じ変数は、やはり次のように、まとめることができます。

```
/* 変数 1 つずつ */
int a = 5;
int b = 2;
int c = a + b;
```

```
/* 同じ型の変数をまとめて */
int a = 5, b = 2, c = a + b;
```

コラム：初期化

初期化という言葉には、2通りの意味があります。

狭義 変数定義と同時に行う代入

広義 変数を定義して、初めて行う代入（同時でなくてもよい）

狭義の初期化だけの機能もあります。警告の「変数が未初期化」は広義です。

^{*3} C 言語は様々な操作を許すために、代入忘れをエラーにしません。他の言語に目を向けると、エラーにしたり、変数を定義したと同時に 0 で初期化されると決めている言語も多くあります。

^{*4} スコープを狭くしたり（☞42 ページのコラム）、コンパイラの警告（☞A.10 節）を活用します。

2.2.4 識別子

識別子 (identifier) は、変数名や関数名に使う文字の並びです。シンボル (symbol) ともいいます。数学の変数名は 1 文字ですが、プログラムでは長い名前がつけられます。

- アルファベットの太文字と小文字、数字、アンダースコア (.) が使えます。
- アルファベットの太文字と小文字は、区別されます。
- 長さは、多くの場面で少なくとも先頭から 31 文字は区別される^{*5}ので、ほぼ気にしなくて大丈夫です。
- プログラム上の特別な役割のある予約語 (reserved word) (☞ A.9 節) は使えません。
- 先頭文字はアルファベットにしておきます。
 - 数字で始めると、数値と解釈されます。
 - アンダースコア (.) で始まる名前は、システムが使うかもしれないので、プログラムで作成してはいけません。

命名の習慣は ☞ 7.3 節で紹介します。ひとまず**変数には小文字**を使っておきましょう。

```
/* 正しい識別子 */
ab
auto_run
power2
seven_zip
unix__
```

```
/* 誤った識別子 */
a b (×スペース)
auto (予約語に一致)
power-2 (×ハイフン)
7zip (×先頭の数字)
__unix__ (×先頭の_)
```

コラム：静的型付け vs. 動的型付け

C 言語では、コンパイル時に型を厳格に決定してしまいます（静的型付け）。型の一致しない代入や演算は、自動変換されたり、コンパイルエラーになったりします。型を首尾一貫することで、正しいプログラムを作る助けにする狙いがあります。言語によっては、実行時まで型を決定しないものもあります（動的型付け）。

^{*5} 長すぎてもエラーにはなりませんが、先頭部分が一致すると、違う変数名でも同じと扱われることとなります。実際に何文字まで区別されるかは、文脈や環境によっても異なります。

2.3 整数

整数を扱うための**整数型** (integer type) を説明します。これまで使ってきたように、1 や 20 など、数字だけの並びは整数型の定数です。先頭が 0 なら 8 進数で解釈されます。先頭が 0x なら 16 進数になり、数字に加えて a~f の英字を使います。大文字と小文字の区別はありません。

```
int a = 123;    // 123   (10進数)
int b = 0123;  // 83    (8進数)
int c = 0x123; // 291   (16進数)
int d = 0xFF;  // 255   (16進数)
```

整数型には、符号の有無などでいくつも種類がありますが、ひとまず int だけ覚えておけば大丈夫です。順番に見ていきましょう。

2.3.1 符号付き整数型

一番よく使う**符号付き整数型** (signed integer type) が int です。環境に応じて、CPU にとって一番扱いやすいものが選ばれています。そして割り当てられるメモリサイズの違いで 4 種類あります*6。(本書では使い分けの必要な場面はほとんどありません。)

「short int」 ≤ 「int」 ≤ 「long int」 ≤ 「long long int」

不等号はメモリサイズの大小を表していて、4 種類の中で同じものもあり得ます。また、int に long のような修飾する単語が付くと int は省略できるので、以降は省略します。

2.3.2 符号なし整数型

負の整数が不要な場面があります。ビット演算や、正の大きな値を扱うときなどです。このために**符号なし整数型** (unsigned integer type) が用意されています。

本書では積極的には用いませんが、標準ライブラリで利用されている (☞9.2 節) ので、存在だけは知っておきましょう。型の名前としては、符号付き整数の型の前に unsigned のキーワードを付けて「unsigned int*7」「unsigned long」のように*8します。

_____ 複数のバイト列で整数を表す場合の内部表現 (バイトの並び順・エンディアン)*9や、マイナスの値の実現方法*10は何通りかあって、CPU によって異なります。C 言語規格では選択の幅を持たせています。_____

*6 long long は C99 で正式に導入されました。

*7 ここでも int は省略できて「unsigned」だけでも同じ意味ですが、本書では省略しないことにします。

*8 unsigned との対称で「signed」のキーワードもあるので、「signed long」という型名も表記可能ですが、long は元から符号付きなので、効果はありません。2.5.1 項の signed char のみ意味があります。

*9 上位ビットから並べるビッグエンディアン、下位ビットから並べるリトルエンディアンなどがあります。

*10 C99 では、次の 3 通りが想定されています。(i) 2 の補数 (ii) 1 の補数 (iii) 符号ビット

表 2.2 <limits.h> で定義された整数型の主な定数 (1)

シンボル	意味	保証値	保証値の絶対値
SHRT_MIN	short の最小値	-32767	$2^{15} - 1$
SHRT_MAX	short の最大値	+32767	$2^{15} - 1$
USHRT_MAX	unsigned short の最大値	65535	$2^{16} - 1$
INT_MIN	int の最小値	-32767	$2^{15} - 1$
INT_MAX	int の最大値	+32767	$2^{15} - 1$
UINT_MAX	unsigned int の最大値	65535	$2^{16} - 1$
LONG_MIN	long の最小値	-2147483647	$2^{31} - 1$
LONG_MAX	long の最大値	+2147483647	$2^{31} - 1$
ULONG_MAX	unsigned long の最大値	4294967295	$2^{32} - 1$
LLONG_MIN	long long の最小値	-9223372036854775807	$2^{63} - 1$
LLONG_MAX	long long の最大値	+9223372036854775807	$2^{63} - 1$
ULLONG_MAX	unsigned long long の最大値	18446744073709551615	$2^{64} - 1$

2.3.3 整数型の表示方法

int を printf() で表示する際の書式文字列は %d です。8 進表記にするなら %o、16 進表記なら %x です。%X なら大文字になります。(☞ A.8 節)

% と d の間に 5 を指定すると、必ず 5 文字以上で表示して、足りなければ先頭にスペース文字が追加されます。05 なら、スペース文字の代わりに 0 で埋められます。

```
printf("%d", 123); // "123" (10進)
printf("%o", 123); // "173" (8進)
printf("%x", 123); // "7b" (16進)
printf("%X", 0xff); // "FF" (16進)
```

```
printf("%5d", 123); // " 123"
printf("%05d", 123); // "00123"
```

2.3.4 整数型の値の範囲

整数型ごとの記憶できる値の範囲は、表 2.2 のように、<limits.h> に定義されています。C 言語の規格では、最低限の範囲が規定されています。つまり、もっと広い範囲の値を格納する処理系 (コンパイラ) があっても構いません。(このように処理系の自由によってよいものを **処理系依存 (implementation dependent)** といいます*11。) 最低限の範囲を「保証値」として載せておきました。C 言語は、パソコンだけでなく、小さな組み込みプロセッサなどでの利用も想定して、このような選択に幅を持たせています。

*11 処理系依存の性質をあてにしたプログラムは、違う環境では動作がおかしくなる可能性があります。このような、動作保証のない状態を「**可搬性 (portability)** を失っている」といいます。

2.4 浮動小数点数

整数ばかりを扱っていても、平均を求めたくなれば、小数付きの計算が必要です。 6.02×10^{23} のような (誤差を含めた) 大きな数を扱いたいこともあるでしょう。

```
double a = 1.23; // 1.23
double b = 1.23e+2; // 123.0
double c = 123E-2; // 1.23
double d = .23; // 0.23
```

このための型が**浮動小数点型** (floating point type)^{*12}です。内部的には、^{かすう}仮数部と指数部に分けて格納され、この値は**浮動小数点数** (floating point number) と呼ばれます。

- 1.23 のように、小数点が入ると浮動小数点型の定数になります。
- $6.02e+23$ のように、e が入ると**指数表記** (exponential notation) の定数です。e の前が仮数、後が指数と解釈され、 6.02×10^{23} の意味です。仮数は 1~10 に正規化されている必要もなく、小数点なくても構いません。e は E でも同じです。
- 多用する機能ではありませんが、整数部分か小数部分が 0 であれば (片方だけ) 省略可能で、0.5 は「.5」、1.00 は「1.」と表記できます。

2.4.1 浮動小数点型

具体的な型としては、領域サイズの違いで、次の 3 通りあります。すべて符号付きです。

float 浮動小数点 (floating point) から容易に想像できる名称ですが、整数型の **short** のような役割で、あまり積極的には用いられません。

double 小数付きの数値といえば、通常はこの **double** を用います。float に 32 ビット、double に 64 ビットと「倍」(double) の領域を割り当てることが多いので、このような名称になっているのですが、言語規格上は倍にする必要はありません。

long double さらに高精度の計算ができそうな名称ですが、実際には環境によって精度や計算速度が大きく異なる^{*13}ので、いつでも役立つわけではありません。

2.4.2 浮動小数点型の表示方法

double を `printf()` で表示する際の書式文字列は `%f` です。float も同じです。(☞ A.8 節)

- `%f` は、小数での表記になります。
- `%e` は、 $6.02e+23$ のような指数表記になります。
- `%g` は、指数に応じて `%f` か `%e` を自動選択します。小数の末尾の 0 が省かれます。

^{*12} 実数 (real number) と呼ぶ人もいますが、数学的には正確ではありません。実際には「有限小数」です。FORTRAN 言語での型名は REAL なので、影響されているのかもしれませんが。

^{*13} long double の実体の例です。(1) double と同じ。(2) 128 ビットの領域を割り当て、演算に使うのは 80 ビット分のみ。(3) 128 ビットの領域の全体で演算をして、計算時間が double の何倍もかかる。

表 2.3 <float.h> で定義された浮動小数点型の主な定数

シンボル	意味	保証値
FLT_RADIX	内部表現に使用される基数 (2, 16, ...)	
FLT_MAX	float の最大値	1e+37 (10 ³⁷)
DBL_MAX	double の最大値	1e+37 (10 ³⁷)
LDBL_MAX	long double の最大値	1e+37 (10 ³⁷)
FLT_DIG	float の有効精度 (10 進数表記の桁数)	6
DBL_DIG	double の有効精度 (〃)	10
LDBL_DIG	long double の有効精度 (〃)	10

※最小値は、最大値の符号違いです。DBL_MIN の意味は [13.7.4](#) 項。

- %8f は、8 文字以上で表示します。8 文字より桁の少ない数なら、先頭にスペース文字を追加します。8 文字より多い数は、必要な文字数まで増やします。
- %.3f は、小数部分が 3 桁になります。%8.3f のように、全体の文字数と同時に指示できます。省略時は 6 と同じです。%.0f なら整数で表示し、小数点もなくなります。

```
printf("%f", 1.2); //"1.200000"
printf("%e", 1.2); //"1.200000e+00"
printf("%g", 1.2); //"1.2"
```

```
printf("%.2f", 1.2e3); //"1200.00"
printf("%.2e", 1.2e3); //"1.20e+03"
printf("%.2g", 1.2e3); //"1.2e+03"
```

2.4.3 浮動小数点型の値の範囲と精度

型ごとの性質をあらわす定数は、表 2.3 のように <float.h> に定義されています。浮動小数点型の性質は複雑で、多くの定数があるのですが、さしあたり最大値・最小値と精度をおさえておきましょう。

表の中に、(マイナスの) 最小値を表す定数はありません。整数型とは異なり、浮動小数点型は必ず符号付きですから、表現できる値の範囲は正負対称です。この性質から、double の最小値は、最大値を利用して -DBL_MAX と表せます。

_____ 現実には、IEEE 754 という規格書で規定されたフォーマットを採用することが多く、単精度 (32 ビット) を float に、倍精度 (64 ビット) を double に割り当てることがほとんどです。精度は float で 7 桁、double で 15~16 桁、と覚えておくと役立ちます。 _____

2.5 文字

プログラムでは、数値だけでなく、文字を扱いたい場面もあるでしょう。まずは1文字を表す**文字定数**です。文字定数は、シングルコーテーション(')で1文字だけを囲みます。0文字でも2文字以上でも、マルチバイト文字*¹⁴でもコンパイルエラーです。

```
char c = 'A'; // 文字定数
// char d = '12'; // NG: 複数文字
// char e = 'あ'; // NG: マルチバイト
```

コンピュータでは一般に、文字は**文字コード** (character code) に変換してから扱います。C言語でも、文字を扱っているかのようにみせかける文法がありますが、最終的には「整数値」に変換しています。つまりこの文字定数は、内部的には数値(文字コード)に置き換わります。そしてプログラムにとって、通常の数値か文字かの違いは、表示の際の形式が10進数なのか、それとも対応する文字コードの文字なのか、だけなのです。

2.5.1 文字型

文字は `char` 型で保存します*¹⁵。`char` は `int` などと同じ、整数型の一員ですが、符号付きかどうかは処理系依存です。1バイトの領域を割り当てると決まっています。

ほかに、符号付きの `signed char` と、符号なしの `unsigned char` という型もあります。どちらも8ビット以上の領域が割り当てられると決まっています。どちらかという文字のためではなく、`signed char` は `short` よりもメモリを節約するためであったり、`unsigned char` はバイト単位のファイルやI/O入出力の際の最小単位として使われることが多く、本書では出番がありません。ちなみに、どちらか片方が `char` と一致するとも決まっています。一致するほうは1バイトになるわけです。

よく使われる文字コード体系は、**ASCIIコード** (☞A.2節) ですが、C言語は文字コード体系に依存しないように設計されています。

2.5.2 文字型の表示方法

`char` を `printf()` で表示する際の書式文字列は `%c` です。`%d` などで**文字コード**が表示されることも理解しておきましょう。右はASCIIコードでの例です。

```
printf("%c", 'A'); // "A" (文字)
printf("%d", 'A'); // "65" (10進数)
printf("%o", 'A'); // "101" (8進数)
printf("%x", 'A'); // "41" (16進数)
```

*¹⁴ マルチバイト文字にはワイド文字 (`wchar_t`) という型が用意されていますが、本書では立ち入りません。

*¹⁵ 1文字単位で扱う場合は、大は小を兼ねる、のことわざのように、`int` で代用する場面も多々あります。

表 2.4 <limits.h> で定義された整数型の主な定数 (2)

シンボル	意味	保証値	保証値の絶対値
SCHAR_MIN	signed char の最小値	-127	$2^7 - 1$
SCHAR_MAX	signed char の最大値	+127	$2^7 - 1$
UCHAR_MAX	unsigned char の最大値	255	$2^8 - 1$
CHAR_BIT	char のビット数	8	
CHAR_MIN	char の最小値	SCHAR_MIN または 0	
CHAR_MAX	char の最大値	SCHAR_MAX または UCHAR_MAX	

「文字定数」と「文字コード」の関係は理解しにくいので、右に例を続けます。'A' は単なる値ですから、文字コードを調べて、その値を書いても動作は同じです。

```
printf("%c", 'A'); // "A" 文字定数
printf("%c", 65); // "A" 10進数
printf("%c", 0101); // "A" 8進数
printf("%c", 0x41); // "A" 16進数
```

ただし、「65の値がAの文字だ」とわかる人は少ないので、これでは人に解読しにくいプログラムになってしまいます。さらに、文字コード体系は環境ごとに違う可能性があるため、環境に依存したプログラムにもなります。ですから、このことは動作原理として理解しておいて、**プログラムには'A'の文字定数**を使いましょう。

文字に関する操作、例えば大文字変換も、文字コード体系によって変わります。このような環境に依存する動作は、システム（コンパイラ）が用意しています。

```
#include <ctype.h>
... 省略 ...
printf("%c", toupper('a')); // "A" 大文字変換
printf("%c", tolower('X')); // "x" 小文字変換
```

2.5.3 文字型の値の範囲

文字型の記憶できる値の範囲は、表 2.4 のように、<limits.h> に定義されています。

コラム：文字定数と文字型

定数にも型があります。1 や 2 などの数値は皆さんの予想通り int 型です。では 'a' のように書かれた文字定数の型は何だと思いませんか？

実は、文字定数の型は int であって、char ではありません。つまり普通の数値と同じです。そうは言っても、この文字定数は char が表現できる範囲に収まっていることが保証されているので、char に代入しても問題は起こりません。

2.5.4 文字列と文字列定数

`printf()` で出力するメッセージは、ダブルコーテーション (") で囲んでいます。1.5.2 項で説明した通り、これは文字列の定数 (文字列リテラル) です。

文字列 (string) は、文字の連なったもので、長さは何文字でもよくて、1文字、0文字の文字列もあります。既に活用していますが、マルチバイト文字も含めて大丈夫です。

```
// char c = 'This'; // NG:複数文字
char c = 'A';
// char c = ''; // NG:空文字
```

```
char *s = "This is 文字列.";
char *t = "A"; // 1文字の文字列
char *u = ""; // 空文字列
```

文字列の型は、`char` とポインタを組み合わせた `char*` ですが、この型の変数を使いこなすには、9章の配列とメモリ確保の仕組みを理解せねばなりません。ここでは、文字列リテラルの復習にとどめておきましょう。

- \ など、特殊な文字を含めるにはエスケープシーケンス (☞ A.3 節) を使います。
- 改行文字は `\n` と書きます。ソースコード上で本当に改行してはいけません。
- ソースコード上で連続する2つの文字列リテラルは、連結されます。

文字列を `printf()` で表示するときの書式文字列は `%s` です。今は定数を表示しても役立つ気がしませんが、将来はプログラムで生成した変数を表示するのに使います。

```
printf("%s is a string.", "文字列"); // "文字列 is a string."
```

なお、`printf()` の書式文字列は文字列リテラルにしましたが、今後とも定数にしてください。変数にすると、セキュリティ上の危険を考慮せねばなりません。

コラム：定数の型と接尾辞

定数の末尾にアルファベットの接尾辞を書き加えると、定数の型を制御できます。小文字でも同じ意味ですが、^{エル}1は^{いち}1と見間違えるので、大文字がよいでしょう。

10進数は、基本的には `int` で、大きな数は値に応じて `long` あるいは `long long` に自動的に切り替わります。明示的に

```
long long a = 65536LL;
long double x = 360.0L;
```

指定するときに、`L` あるいは `LL` を使います。`unsigned` にするなら `U` です。同時に指定すると `123ULL` のようになります。なお、`short` や `char` の定数は存在しないので、どうしても必要ならキャスト (☞ 2.7.3 項) します。

浮動小数点の場合は、基本的に `double` で、自動的に切り替わりません。接尾辞で精度を選ぶことになって、`F` で `float`、`L` で `long double` になります。

表 2.5 四則演算の演算子

演算子	演算の種類
+	加算 (和)
-	減算 (差)
*	乗算 (積)
/	除算 (商)
%	剰余

表 2.6 単項演算子 (抜粋)

演算子	演算の種類
+	単項プラス
-	単項マイナス
++	インクリメント
--	デクリメント

2.6 式と単純な演算子

「1+2+3」のような式は、演算が行なわれて、最終的には1つの値に置き換わります。式中の「+」のような、演算の種類を表すものを**演算子 (operator)**、「1」や「2」のような演算対象となる値を**被演算子**、あるいは**オペランド (operand)** といいます。

四則演算 (four arithmetic operations) の演算子は、表 2.5 のように5種類あります。

- これらは「2+3」のように、演算子の前後に2つのオペランドをとる**2項演算子**です。
- 数学では $2x$ のように掛け算記号を省略できますが、プログラムでは省略できないので「 $2*x$ 」と書きます。
- 整数同士の演算は、結果も整数です。
- 2つのオペランドの型が揃っていないと、(int→doubleのように、情報量の多い型のほうに)自動的に格上げされます。

最後の%は余りを求める**剰余演算子 (modulus operator)**^{*16}です。割り算の、商と剰余が別々の演算子になっているので、5種類の演算子があります。整数で演算を行なうと、商も剰余も整数で得られます。商は整数に切り捨てられているようにも見えますが、実際のところは(被除数)=(除数)×(商)+(剰余)の関係を満たすようにしているのでしょう。マイナスの値が入ると状況が複雑になります (☞ 28 ページのコラム)。

整数演算の0の割り算は、規格上の動作は未定義ですが、たいていプログラムが強制終了させられます。Cygwinでは以下のようなメッセージが表示されます。

```
Floating point exception (コアダンプ)
```

符号付きの整数演算で、表現できる範囲を越えた桁あふれ(オーバーフローやアンダーフロー)が起こった場合も、規格上の動作は未定義です。現実には演算結果がおかしくなるだけがほとんどですが、プログラムとしては、どれも起こらないように対策が必要です。

^{*16} C言語では、%は整数専用です。浮動小数点の場合は、数学関数のfmod(x,y)を使います。Javaなど、型の区別なく%で扱える言語も多数あります。

2.6.1 単項演算子

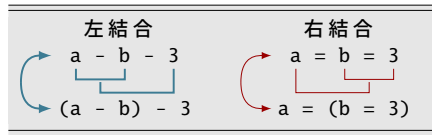
式の先頭に「-3」が出てきた場合、このマイナスは2項演算子ではなく、オペランドが1つの単項演算子 (unary operator) として働き、符号を反転します。表 2.6 のように、マイナスとの対称で、「+3」のような、プラスの単項演算子もあります。

2.6.2 優先順位

数学の式では、カッコ () の中を先に計算しますが、プログラムでも同じです。カッコの種類は、プログラムでは一通りだけで、大括弧のような大きさ違いはありません。

和や差よりも積や商を優先するのも、数学と同じです。つまり「1+2*3」は「1+(2*3)」と同じです。さらに A.5 節を参照してもらえると、単項演算子の優先順位が高いので、「(-2)*(-3)」は「-2 * -3」とカッコがなくても同じだとわかりますが、細かい規則を覚える必要はありません。

同じ優先順位の演算が並んだら、多くの演算子は左のものが優先です (左結合)。代入演算子の = は、(数少ない) 右結合なので、右から評価して、しかも式の値は代入した値そのものになり、同じ値を連続して代入できます。



コラム： 剰余はプラスに揃えてから

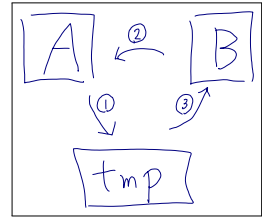
剰余は、日常生活ではそれほど意識して使うことはありませんが、コンピュータ上では、周期的な動作をさせるなどの場面で多用されます。

ところで、マイナスの値が含まれると、商とともに、剰余はややこしくなります。(被除数)=(除数)×(商)+(剰余) の関係は常に満たされるのですが、a % b の符号は (A) a と同符号 (B) b と同符号 (C) 常にプラスの3通りの動作が考えられます。

どれにするかは C89 まで処理系依存でしたが、C99 で (A) に定められました。(A) は Java や、近年の C++ と同じで、多数派ともいえます。しかし、Perl や Python では (B) を採用していますし、数学の有力な定義では (C) になっていたりと、まちまちです。

a	b	(A)		(B)		(C)	
		/	%	/	%	/	%
+10	+7	+1	+3	+1	+3	+1	+3
+10	-7	-1	+3	-2	-4	-1	+3
-10	+7	-1	-3	-2	+4	-2	+4
-10	-7	+1	-3	+1	-3	+2	+4

細かな言語規格をあてにせず、a も b もプラスの値に限定するのが安心です。



2.7 変数と複雑な演算子

プログラミング特有の変数の使い方や、数学にはない演算子を見ていきます。

2.7.1 値の入れ替え（スワップ）

2つの変数の値を入れ替える操作は**スワップ** (swap) といい、慣用句になっています。変数は、代入された瞬間に、それまで記憶していた値を忘れます。このため2つの変数同士で代入していたのでは、代入された方の変数の値は失われるので、工夫が必要です。

記憶してきた値を失わないためには、3つめの**待避用の変数**が必要になります。つまり a と b をスワップするには、a=b の前に、(1) 3つめの変数 tmp に a の値を待避します。(2) a=b が終わってから、(3) 待避しておいた tmp (元は a の値) を b に代入します。待避用の変数を思いつくところがポイントです。

tmp (あるいは temp) という名前は temporary (一時の、臨時的の) の省略形です。使い捨てにする変数名によく使われます。

```
int a = 3, b = 5;
int tmp = a; a = b; b = tmp; // a = 5, b = 3
```

2.7.2 インクリメント・デクリメント・複合代入

プログラムでは「 $x = x + 1;$ 」という表現をよく見かけます。数学の方程式だと思えば、 x を移項して「 $0=1$ 」の“解なし”になる、おかしい式ですが、プログラムでの動作は、右辺を計算して、左辺に代入するので、結果として x の値が **1 増えます**。同様に「 $x = x - 1;$ 」なら x が **1 減ります**。

```
int x = 100;

x = x + 1; // x に 1 加える
printf("%d\n", x); // "101"

x = x - 1; // x から 1 減らす
printf("%d\n", x); // "100"

x++; // x に 1 加える
printf("%d\n", x); // "101"
```

1 加える、という操作はよく使うので、**インクリメント (increment)** と呼ばれていて、「`x++;`」という専用の演算子があります。1 減らすのは**デクリメント (decrement)** で、「`x--;`」です*17。

「`x = x + 5;`」の 5 加える操作には、専用の呼び名も演算子もありませんが、右辺と左辺に同じ変数が現れるので、**複合代入演算子 (compound assignment operator)** を用いると「`x += 5;`」と少し短くなります。

<code>x = x + 5;</code>	⇔	<code>x += 5;</code>
<code>x = x - 5;</code>	⇔	<code>x -= 5;</code>
<code>x = x * 5;</code>	⇔	<code>x *= 5;</code>
<code>x = x / 5;</code>	⇔	<code>x /= 5;</code>
<code>x = x % 5;</code>	⇔	<code>x %= 5;</code>

- 四則演算すべてに、この複合代入があります。変数部分が高いときに有用です。
- `+` と `=` の間にはスペース文字を挟んではいけません。
- `+` と `=` を間違えて「`x += 5;`」と入れ替えると単なる代入になるので、要注意です。

2.7.3 型変換 (キャスト)

整数と浮動小数点数とでは、プログラムでの扱いが違いました。整数の 5 と 2 の割り算では、商は $5 / 2 = 2$ 、余りは $5 \% 2 = 1$ と求められますが、小数付きの 2.5 を得るにはどうしたらよいでしょうか。

小数付きの結果を得るには、演算前から小数付きの値にしておきます。つまり `5.0/2.0` なら 2.5 が得られます。では、整数の 5 や 2 から 2.5 を得るにはどうしたらよいでしょう。

整数を小数付きにするには、**キャスト (cast)** 演算子で、強制的な型変換を行います。

```
/* 文法 */
(型名) 値
```

```
/* 実例 */
(double)5 // → 5.0
```

この機能を用いて、`(double)5/(double)2` とすると `double` の 2.5 が得られます。ここで注意してほしいのが、**キャストの順序**です。`(double)(5/2)` だと、整数同士の $5/2 (=2)$ の計算をすませてから `double` に変換するので、2.0 になってしまいます。

```
/* 成功 */
int a = 5, b = 2;
double c = (double)a / (double)b;
           2.5    5.0      2.0
```

```
/* 失敗 */
int a = 5, b = 2;
double c = (double)(a / b);
           2.0      2
```

三角形の底辺 w と高さ h を `int` 型で保持しているとします*18。面積の計算は、次のように `double` 型に揃えるのがよいでしょう。数学の公式では $w \times h \div 2$ と、理想的な 2 で割りますが、プログラムでは、誤差を含む可能性のある 2.0 で割る*19 ことになります。

*17 前置の「`++x;`」や「`--x;`」もあります。後置ともほぼ同じ動きです。(☞75 ページのコラム)

*18 もっとも、面積を求めるつもりがあるのなら、初めから w, h を `double` 型にしておくのが現実的です。

*19 実行環境に依存しますが、浮動小数点の内部表記が 2 進数なら、2 の割り算は誤差なく行なえます。

```
int w = 10, h = 5;
printf("三角形の面積は %f\n", (double)w * (double)h / 2.0);
```

キャストの優先順位は四則演算よりも高いので (☞ A.5 節)、double の計算結果を整数値に丸めるときなど、キャストする式全体をカッコで囲う場面がよく出てきます。

コラム：キャストの使用は最小限に

キャストによる情報の欠落には注意しましょう。キャストは double を整数型に丸め込むことも可能です。元の値の

```
(int)3.14 // →3 (注意)
(char)257 // →1 (注意)
```

小数部分はなくなります。また、大きな整数値を char にキャストすることもできます。上位ビットがなくなって、下位 1 バイトだけが取り出されます。

意図的ならもちろん構わないのですが、キャストは危険でも強制的に操作を行うよう指示するものですので、おかしな指示でも警告は出ません。(もちろん不可能ならコンパイルエラーです。) ですから、「試行錯誤してキャストでエラーがなくなった」というのは、まったくあてになりません。つじつま合わせのために、むやみにキャストするのはやめておきましょう。

2.8 マクロ定数

永続的に値の変化しない定数があります。プログラムでは、変数に代入してから利用してもよいのですが、マクロ (マクロ定数) のほうが安全です。間違っ値を代入するとエラーになるからです。表 2.2 などのシンボルもマクロ定数で実現されています。

マクロは、`#define` 命令^{*20}で「文字の置き換え」を指示します^{*21}。マクロ名にも識別子を使いますが、変数とすぐに区別がつくよう、すべて大文字にするのが習慣です。

```
/* 文法 */
#define マクロ名 置き換え文字
```

```
/* 実例 */
#define MAX_SIZE 10
#define POW3_3 (3*3*3)
```

^{*20} # から始まるキーワードはプリプロセッサ命令に分類され、1 行の終わりが命令の一区切りです。セミicolon は関係ありません。命令の途中で改行したいなら、行末に \ をつけて、行継続することを示します。

^{*21} このため、数値以外にも使い道がありますが、逆に式の優先順位は考慮されないで、定義する数値全体をカッコで囲う必要もあります。

2.9 型ごとの限界値を探る

型の上下限の値をソースコード 2.1 で確かめてみましょう。INT_MAX などのシンボルを使う前には、`#include <limits.h>` や `#include <float.h>` が必要です。「sizeof(型名)」で型の占める領域のバイト数がわかります。printf() の "%zu" の意味は 9.2.1 項で説明します。

GCC Intel 64 ビットでの実行例は右のようになりました。int が 4 バイト (32 ビット)、long long が 8 バイト (64 ビット) だとわかります。double の精度は保証値 (10 桁) よりもかなり良いです。環境によっては、また違った結果になります。

ソースコード 2.1 の実行例 (環境依存)

```
INT_MAX=+2147483647
INT_MIN=-2147483648
sizeof(int)=4

LLONG_MAX=+9223372036854775807
LLONG_MIN=-9223372036854775808
sizeof(long long)=8

FLT_MAX=+3.40282e+38
FLT_DIG=6
sizeof(float)=4

DBL_MAX=+1.79769e+308
DBL_DIG=15
sizeof(double)=8
```

ソースコード 2.1 型ごとの上下限の値を表示

```
1 #include <stdio.h>
2 #include <limits.h> // INT_MAX, LLONG_MIN など
3 #include <float.h> // DBL_MAX, FLT_DIG など
4
5 int main(void) {
6     printf("INT_MAX=%d\n", INT_MAX);
7     printf("INT_MIN=%d\n", INT_MIN);
8     printf("sizeof(int)=%zu\n", sizeof(int));
9     printf("\n");
10
11     printf("LLONG_MAX=%lld\n", LLONG_MAX);
12     printf("LLONG_MIN=%lld\n", LLONG_MIN);
13     printf("sizeof(long long)=%zu\n", sizeof(long long));
14     printf("\n");
15
16     printf("FLT_MAX=%g\n", FLT_MAX);
17     printf("FLT_DIG=%d\n", FLT_DIG);
18     printf("sizeof(float)=%zu\n", sizeof(float));
19     printf("\n");
20
21     printf("DBL_MAX=%g\n", DBL_MAX);
22     printf("DBL_DIG=%d\n", DBL_DIG);
23     printf("sizeof(double)=%zu\n", sizeof(double));
24     return 0;
25 }
```


コラム：多倍長演算

もっと大きな値を扱いたくなったら、あるいは、もっと高精度の小数を扱いたくなったら、多倍長演算が必要になります。CPU（ハードウェア）の直接の命令では計算できなくなくなって、ソフトウェアで模倣（エミュレート）するため、計算速度が数倍も遅くなるのは、原理的に避けられません。

残念ながら C 言語の規格には、多倍長演算のための標準ライブラリはありません。GNU Multi-Precision Library (GMP) という有名なライブラリがありますが、C 言語からは使いにくいので、C++ からの利用を考えたほうがよいでしょう。Java なら BigInteger / BigDecimal、Perl にも bigint / bignum などの標準のライブラリがあります。Python3 なら標準機能に組み込まれています。

コラム：四捨五入

double 型の x を四捨五入して、整数を得たいとしましょう。int にキャストすると小数部分がなくなるので、0.5 を足してからキャストする $(\text{int})(x+0.5)$ が四捨五入になる、というのが慣用句でした。しかし、状況次第で問題があります。

- x がマイナスなら、0.5 を足すのではなく、引かねばなりません。（キャストは 0 方向への丸めですから、正負で丸め方向が違います。）
- x が絶対値の大きな値なら、int ではあふれるかもしれません。

double 型のまま四捨五入するなら、数学関数（☞ A.7 節）を用いて $\text{floor}(x+0.5)$ あるいは C99 で新設された $\text{round}(x)$ が簡便です。さらに long long 型で得るなら $\text{llround}(x)$ もあります。なお、 $\text{printf}(\text{"\%.0f"}, x)$ の丸め方向は、C99 では FLT_ROUNDS（☞ 13.7.4 項）に従うことが期待されていて、現実には四捨五入とは微妙に異なる、偶数丸めであることが多いです。

2.10 練習問題**1. [代入・左辺値・複合演算子]**

プログラム中の int 型変数 x について、以下を説明せよ。

- 「 $x = 10;$ 」と「 $10 = x;$ 」の違いは何か。（コンパイルエラーに着目せよ。）
- 「 $x = x + 5;$ 」は、どのような作用があるか。また += 演算子を用いて同じ動作を実現せよ。

2. [変数の保持するもの]

右のプログラムの無駄を省いて、単純にせよ。

3. [スワップ]

変数 a, b, c がある。 $\overrightarrow{a \leftarrow b \leftarrow c}$ とプログラムで3つの値を入れ替えてみよ。

4. [左辺値]

1元1次方程式 $ax + b = c$ の解 x をプログラムで求め、表示せよ。 x, a, b, c はすべて `double` 型の変数と定義せよ。

```
#include <stdio.h>

int main(void) {
    int a = 10;

    a = 20;
    a = 30;
    a = 40;
    a = 50;
    printf("%d\n", a);
    return 0;
}
```

5. [剰余計算]

度数法の2つの角度 a, b ($0 \leq a, b < 360$) の和と差を、それぞれ0以上360未満で求めよ。 a, b とも `int` 型とし、剰余演算子 `%` の両辺とも負にならないよう注意せよ。

6. [剰余計算]

正の整数 x を123とするとき、以下の値を表示せよ。

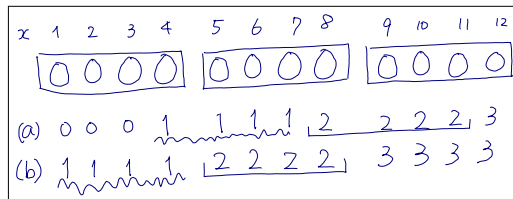
- x を10で割った余りはいくらか。
- x を10で割った商を y とする。 y を10で割った余りはいくらか。
- x を100で割った商を z とする。 z を10で割った余りはいくらか。

この3つの値が、 x の十進数表記とどのような関係になっているのか考察せよ。 x を別の値に変えて試し、考察を補強せよ。

7. [切り捨て・切り上げ]

x 人が4人がけの長椅子に、隙間を空けずに座る。次の(a)(b)を、それぞれ x の式で表せ。(ヒント:(b)は、 x の3ずれた(a)と同じ。)

- (a) 満席の長椅子の個数 (満席ではない椅子は1台あってもよい)
 (b) x 人が座るのに必要な長椅子の個数



8. [切り上げ]

1以上の整数 x について、西暦 x 年が何世紀であるかを表示せよ。西暦0年は存在せず、西暦1年から100年までは1世紀、101年から200年までは2世紀と、100年ごとに新しい世紀になる。

第3章

関数 (1)

本章では、プログラムの部品を作る方法を説明します。プログラミングに限らず、世の中の複雑な大きな問題を解決する場合に、小さな問題に分割して、段階的に対処していくことがよくあります。駐車場の自動清算機のように複雑な作業を実現する時にも、やはりプログラムを小さな機能（部品）に分割して作っていきます。この小さな部品が「関数」です。

プログラミング言語の「関数」は、数学で習った「関数」と似たところもあります。関数をうまく使えば、再利用によってプログラム全体の長さを短くしたり、プログラム信頼性を高めたりすることも可能です。処理に名前をつけるので、内部の細かな動きを忘れてよくなり、プログラムの見通しがよくなります。

C言語に限らず、ほとんどのコンピュータ言語に備わった機能ですので、ぜひとも身につけましょう。

キーワード

- 関数, 引数, 戻り値
- 関数の入出力
- スコープ
- 値渡し, 参照渡し

3.1 関数の入力=引数・関数の出力=戻り値

数学で習った関数を思い出すと、例えば2乗を返す関数は、 $y = x^2$ とか、 $f(x) = x^2$ のように表記していました。 $f(x) = x^2$ のほうがC言語の関数に近いので、この表記で説明すると、 f が **関数名**、左辺の (x) が **入力となる変数**、右辺の x^2 が **関数の返す値** です。このように数学の関数は3つの要素で構成されています。

C言語での関数も、やはり3つの要素があります。

(数学の文法)	(数学の実例)
関数名 (入力となる変数) = 関数の返す値	$f(x) = x^2$
<pre>/* C言語の文法 */ 型名 関数名 (引数) { return 戻り値; } </pre> <p>関数ブロック</p>	<pre>/* C言語の実例 */ double f(double x) { return x * x; } </pre>

関数名 数学ではアルファベット1文字にしますが、プログラムでは変数名と同じく識別子 (☞ 2.2.4 項) ですので、何文字の英数字でも構いません。

入力となる変数 関数名の後ろに、型名と変数名のペアを書き並べ、全体を () で囲みます。この変数を **引数**^{ひきすう}*1 と呼びます。

関数の返す値 関数の出力に相当するもので、**戻り値**^{もど} とか **返り値**^{かえ} と呼びます。型名は関数名の前に書きます。値を書く場所は、引数の括弧に続く { } で囲ったブロックの中の **return** 文のすぐ後です。このブロックには、**return** 文だけでなく、戻り値を計算する手続きも書き並べます。

頻出ミス

戻り値は **return** 文に書きますが、戻り値の型は **関数名の前** と、離れたところに書くので注意してください。

「関数の戻り値の型」を、簡単に「関数の型」ともいいます。「関数 f は **double** 型」という言い回しもあります。

*1 因数 (いんすう) と区別するため、あえて湯桶読み (訓音の混じった変則的な読み) で「ひきすう」と呼ぶようです。

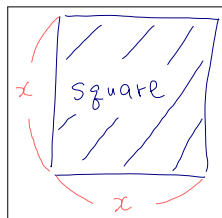


表 3.1 関数の入力と出力

	入力	出力
名称	ひきすう 引数	もど かえ 戻り値, 返り値
	(parameter), (argument)	(return value)
変数名や値を書く場所	関数名の後の () の中	return 文
型を書く場所	各変数名の前	関数名の前

$f(x) = x^2$ を C 言語で実現したものがソースコード 3.1 の 4~6 行目です。関数の名前は square としてみました。引数は double 型の変数 x で受け取り、戻り値を double 型で返します。計算手順はあまりにも短いので、return 文の後ろに直接 $x*x$ と書きました*2。

ところで、square 関数は $f(x) = x^2$ の実現例として作りましたが、ここからは、正方形の面積を求める関数だとして話を進めます。x が正方形の 1 辺の長さだと思うわけです。英語の square には「2 乗」以外に「正方形」の意味もあるので、関数名も計算手順このままで大丈夫です。

ソースコード 3.1 正方形の面積を求める関数

```

1 #include <stdio.h>
2
3 /* 正方形の面積 (1 辺の長さを x とする) */
4 double square(double x) {
5     return x * x;
6 }
7
8 int main(void) {
9     printf("1 辺の長さが 1.0 の正方形の面積は %f です。 \n", square(1.0));
10    printf("1 辺の長さが 5.0 の正方形の面積は %f です。 \n", square(5.0));
11    return 0;
12 }

```

*2 C 言語にべき乗演算子はありません。(x^y はビット演算の排他的論理和です。) そのため、2 乗や 3 乗なら掛け算を繰り返すのが慣用句になっています。なお、数学関数にはべき乗関数 pow(x,y) があります。

3.2 関数呼び出しと実行順序

さて、この自作した `square` 関数を呼び出すにはどうしたらよいでしょう。関数が登場してよいのは、計算式の中です。そして数学と同じで、関数名に続けて `()` の中に引数として渡す値を書きます。もし引数が複数あれば、いつも `printf()` でやってるように、コマで区切って並べます。

C 言語のプログラムには、**最初に `main` 関数が実行される**という約束があります。そこで `main` 関数から呼び出すことにします。`printf()` の引数に計算式を書けるので、そこに `square()` を書いてみます。この例がソースコード 3.1 の 9 行目と 10 行目です。

- 9 行目の呼び出しでは、引数 `x` に `1.0` を与えたので、この時の 5 行目の `x*x` の値は `x` に `1.0` が代入された状態で計算することになって、 $1.0*1.0=1.0$ になります。
- 10 行目の呼び出しでは、引数 `x` に `5.0` を与えたので、この時の 5 行目の `x*x` の値は $5.0*5.0=25.0$ になります。

このそれぞれの値が呼び出し元に返され、`printf()` で表示されます。

ソースコード 3.1 の実行結果

```
1辺の長さが1.0の正方形の面積は1.000000です。  
1辺の長さが5.0の正方形の面積は25.000000です。
```

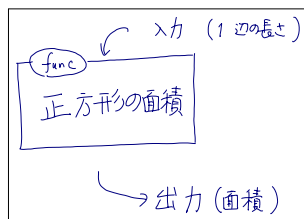
このように、`main()` から、何度でも `square()` を呼び出すことができます。ソースコード上は `main()` よりも `square()` のほうが先に記述してありますが、**実行順序とは無関係**ということ。呼び出される関数の引数の値は、呼び出し元 (`main` 関数) が与えますので、その時ごとに違った値にすることができます。

ソースコード上に記述する関数の順序を説明しておきます。ここでは `main()` を後に書いています。これはコンパイル (実行プログラムへの翻訳) 作業の都合で、関数呼び出しよりも前に、関数の宣言が必要なためです。ひとまずは、呼び出される関数を先に書くことにして、順序制約から逃れる方法は 7.1 節で説明します。

コラム：自作関数 vs. システム関数

こうして見ると、我々がいつも書いている「`int main(void) ...`」も、実は関数の 1 つであったことに気づくでしょう。`main` 関数は、引数なし、戻り値が `int` 型の `0` という、一風変わったものでした。

実は `printf()` も関数です。C 言語のシステムが用意してくれているものですが、我々が自作した関数と区別は (本当は) ありません。



3.3 関数を作る意義

無事に `square()` の呼び出しができましたが、このような単純な計算を、わざわざ部品として独立させる意義は感じられるでしょうか。関数にせざとも `5.0*5.0` と書いておいたほうが手っ取り早いと感じる人もいるかもしれません。でも、この関数にわざわざ「正方形の面積を求める」という役目を与えたことを思い出して欲しいのです。

関数に独立させる意義は、**手続きに名前をつける**ことにあります。関数内の手続きが単純か複雑かは後の話です。そして今の例では、引数が正方形の 1 辺の長さ x だけですから、「正方形の面積は 1 辺の長さだけで計算できて、正方形の場所には影響されない」ことも読み取れます。さらに面積が小数付きの `double` 型で得られることもわかります。このように、手続きに必要な「**入力**」と「**出力**」が**明確に列挙される**という重要な側面もあるのです。

このような意義を感じ取れば、単純な `square()` を作ることも無駄ではないと思えるでしょう。もし図形の面積を求める関数がいくつもあって、そのうちの 1 つが `square()` であったとすれば、ますます存在感が増すでしょう。そうなるように、次は三角形の面積を求める関数を作ってみます。

コラム：バグ・デバッグ

プログラミング用語の**バグ (bug)** の語源は「虫」で、プログラムの動作不良や欠陥を指しています。その虫 (bug) を取り除いて (de-)、正しいプログラムにすることを**デバッグ (debug)** という造語で呼んでいます。

3.4 複数の引数

ここで三角形の面積を求める関数を作りたいのですが、三角形の面積の公式はいくつもあります。その中で、式の少し複雑な、ヘロンの公式を使ってみます。この公式によると、3辺の長さを a, b, c としたときに、 $s = \frac{a+b+c}{2}$ という変数を用意して、面積 T は

$$T = \sqrt{s(s-a)(s-b)(s-c)}$$

で求められます。これをプログラムにしたものがソースコード 3.2 の `triangle` 関数 (5行~9行) です。

5行 関数の名前は `triangle` で、`double` 型の値を返すことにします。引数として3辺の長さが必要なので、関数名の後ろに3つの変数を書き並べています。複数の引数は、このようにカンマで区切って書き並べます。今回は型がどれも `double` と共通ですが、まとめて「`double a,b,c`」と書くことはできません。

6行 s という変数を用意して、ヘロンの公式に従った値の代入を同時に行なっています。

7行 面積を求めて `area` という変数に代入しています。`return` 文に計算式を直接書くこともできますが、少々複雑なので、ここでは一旦変数に代入しています。

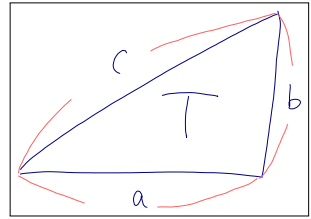
8行 `return` 文で、求めた面積を返します。5行目のとおり、この関数の戻り値は `double` 型ですので、`return` にも `double` 型の値を与えます。

ソースコード 3.2 三角形の面積を求める関数

```

1 #include <stdio.h>
2 #include <math.h>
3
4 /* 三角形の面積 (3辺の長さをa,b,cとする・ヘロンの公式) */
5 double triangle(double a, double b, double c) {
6     double s = (a + b + c) / 2;
7     double area = sqrt(s*(s-a)*(s-b)*(s-c));
8     return area;
9 }
10
11 int main(void) {
12     printf("3辺の長さが%g,%g,%gの三角形の面積は%gです。 \n",
13           1.0, 1.0, 1.0, triangle(1.0, 1.0, 1.0));
14     printf("3辺の長さが%g,%g,%gの三角形の面積は%gです。 \n",
15           3.0, 4.0, 5.0, triangle(3.0, 4.0, 5.0));
16     return 0;
17 }

```

2行目の「`#include <math.h>`」という記述は、数学関数（ここでは平方根を求める `sqrt()`）を使うときに必要なものです^{*3}。詳しくは 7.1.2 項で説明します。数学関数の一覧は A.7 節にあります。

実行結果は、次のようになります。

ソースコード 3.2 の実行結果

3辺の長さが1,1,1の三角形の面積は0.433013です。
3辺の長さが3,4,5の三角形の面積は6です。

3.5 ブロックと変数の有効範囲 (スコープ)

変数には、どの範囲で使えるのかという概念があって、変数の使える範囲をその変数の**有効範囲**^{*4}、あるいは**スコープ (scope)** といいます。C 言語では、変数のスコープはその変数の定義された関数ブロックに制限されています。つまり、ソースコード 3.2 の6行目の変数 `s` は、関数の終わりである9行目までで使えます（実際に7行目でも使っています）が、それ以外の `main` 関数のような場所では使えないということです。たとえ `main` 関数で同じ名前の `s` という変数を用意したとしても、`triangle()` の `s` とは異なる、別の値を記憶する変数として扱われます。このおかげで、関数ごとの変数名に重複があるかどうか気にせず、その関数のことだけを考えてプログラムを作ればよいことになります。

関数の処理部分の `{ }` で囲んだ部分は関数ブロックといたしました。関数ブロックの内側の一部を、この `{ }` で囲んでもよくて、囲んだ部分を単に**ブロック (block)** といいます。

^{*3} Unix 系の `gcc` などのコンパイラでは、さらに `-lm` のコンパイルオプションが必要になる場合もあります。（[A.10 節](#)）

^{*4} 日本語訳はあまり一定せず、「可視範囲」「通用範囲」などとも呼ばれます。専門用語としてもっとも定着しているのは、カタカナの「スコープ」でしょう。

```
int main(void) {  
    {  
        /* ブロック1 */  
    }  
    {  
        /* ブロック2 */  
    }  
}
```

```
int main(void) {  
    {  
        /* 外ブロック */  
        {  
            /* 内ブロック */  
        }  
    }  
}
```

上記では2つのブロックを作ってみました。左のように直列に並べたり、右のように入れ子にしたりすることができます。ブロックだけをこのように記述することはそれほどありませんが、条件分岐やループ処理などの機能と組み合わせて、複数のブロックを頻繁に使います。

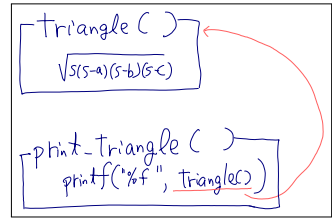
そしてスコープで重要なことがあります。変数の定義された場所が関数ブロックよりもっと内側のブロックであれば、スコープはそのブロックに、つまり関数ブロックよりも狭くなります。(逆に関数ブロックの外で定義したときの話は、7.2.3項で説明します。)

引数の場合は、変数定義が関数ブロックの外にあるようにも見えますが、スコープは関数ブロックの内側の先頭で定義されたのと同じと考えます。つまり、引数のスコープは関数ブロック全体です。

コラム：スコープは狭く

小さなプログラムを作っている間は、1つの変数をいろいろな関数で共有できたほうが便利に思えるかもしれません。しかし、プログラムの間違いで共有の変数の値をおかしくしてしまうことを想像してみてください。プログラムが大きいほど、共有している関数が多いほど、間違いを探すのに手間がかかります。

スコープを狭くしておけば、変数名の重複を気にしなくてよくなったり、コンパイル時に初期化忘れを検出してくれたり、良いことがたくさんあります。短絡的な便利さに目を奪われないようにしたいものです。



3.6 関数の再利用とプログラムの信頼性

さて、ソースコード 3.2 で三角形の面積をもっといろいろ計算したいと思えば、どうしたらよいでしょう。main() の printf() を増やせばよいでしょうか。もちろん、注意深く増やすことは可能です。テキストエディタのコピー機能を使って、printf() 部分をコピーして、3 辺の数値を編集すればよいのですが、同じ数値が表示用と計算用の 2 度出現することに注意してください。もし異なる数値を書いてしまったとしたら、実行結果はおかしなことになるてしまいます。

```
printf("3辺の長さが%g,%g,%gの三角形の面積は%gです。 \n",
      3.0, 4.0, 5.0, triangle(3.0, 4.0, 5.0));
//          表示          計算
```

このような不整合を、プログラムの構造によって防ぐ方法があります。それにはやはり関数を用います。ソースコード 3.3 の新しく作った print_triangle() (11 行~14 行) を見てみましょう。引数 a,b,c を printf() の表示と triangle() の面積計算の両方に使います。この関数を使う限り、原理的に不整合が起こらないというわけです。このような手法の積み重ねが、プログラムの信頼性向上に貢献します。

```
printf("3辺の長さが%g,%g,%gの三角形の面積は%gです。 \n",
      a, b, c, triangle(a, b, c));
//          表示          計算
```

さて、print_triangle() の、関数としての目新しい機能を 2 つ述べておきます。

- 自作関数 triangle() を、別の自作関数 print_triangle() から呼び出せます。
- print_triangle() は、printf() で表示を行なうだけの一風変わった関数です。詳しくは次の節で説明します。

ソースコード 3.2 とソースコード 3.3 を比べると、まったく同じ実行結果が得られるのに、関数を余分に書いたほうがプログラムが長くなったように思えるかもしれません。しかし、表示したい三角形を増やしていくと、長さが逆転します。

3.7 void 型の関数

ソースコード 3.3 の 11 行目の `print_triangle()` を見てみると、関数の型が `void` と宣言されています。`void` とは、日常生活ではあまり使わない言葉ですが、「無効の」「空っぽの」というような意味があります。空っぽを返すということは、何も返さないということです。

何も返さない関数を、数学の関数から連想すると、存在意義を疑いたくなりますが*5、この関数では `printf()` による表示を行っていますので、C 言語で考えると意味のある作業をしています。

この関数には返す値がないので、`return` 文也没有ありません (14 行目)。呼び出し側では、この関数の戻り値を受け取ることはできないので、変数に代入したりせずに、単に関数呼び出しを書き並べます (17-18 行目)。

ソースコード 3.3 三角形の面積を表示する関数

```

1 #include <stdio.h>
2 #include <math.h>
3
4 /* 三角形の面積 (3 辺の長さを a, b, c とする・ヘロンの公式) */
5 double triangle(double a, double b, double c) {
6     double s = (a + b + c) / 2;
7     double area = sqrt(s*(s-a)*(s-b)*(s-c));
8     return area;
9 }
10
11 void print_triangle(double a, double b, double c) {
12     printf("3 辺の長さが %g, %g, %g の三角形の面積は %g です。 \n",
13         a, b, c, triangle(a, b, c));
14 }
15
16 int main(void) {
17     print_triangle(1.0, 1.0, 1.0);
18     print_triangle(3.0, 4.0, 5.0);
19     return 0;
20 }

```

図解: 青い矢印は、関数 `triangle` の引数 `a, b, c` のスコープを示しています。赤い矢印は、関数 `print_triangle` の引数 `a, b, c` のスコープを示しています。

*5 Pascal という言語では、値を返さないものは関数ではなく、手続き (プロシージャ) として、区別して扱います。

3.8 出力 = return

return 文は、関数の処理の最後を書いて、戻り値を示すのに使いました。return に書いた値は、関数の型（関数名の前に書いた型）で返されるので、この2つの型は一致させておくのが真っ当です。（しかし現実には甘くはありません。下のコラム参照。）

void 型の関数には、return 文を書く必要はありませんが、書きたいければ書いても構いません。その場合の return 文には、返す値がないので、式を書かずにすぐに ; を書きます。

void 型に限らず、関数には複数の return を書いても構いません。もし関数の途中で書くと、それ以降の処理を行わずに打ち切るという動作をします。もちろん単独で用いることはありませんが、4章の条件分岐と組み合わせると、if 文の else 節を書かなくてよくなるので、インデントが深くならずすみすみます。

コラム：return の書き忘れ、型違い

void 型以外の関数では return 文が必要なのは当然ですが、書き忘れてもエラーにならずに警告止まりです^a。条件分岐（4章）の結果、return 文にたどり着かないという場合もあるので、見抜けるかどうか、プログラマは注意力を試されてるようなものです。ちなみに、return のないままで呼び出し元に返される値は、初期化されてない変数と同様で、どんな値になるのかの保証がありません^b。

さらに、関数の型と return に書いた式の型が一致しなければ、黙ってキャストされて警告すら出ないかもしれません。例えば int を返す関数で return 3.14; とすれば、int の 3 が返される^cということです。

このように、コンパイルに成功してもまったく油断できません。警告レベルを上げる（[図 A.10 節](#)）などして対処しましょう。

^a Java や C# のように設計の新しい言語では、当然のようにエラーになります。

^b 最後に評価した式の値が（偶然にも）採用されることがあり、return 忘れを気づきにくくしています。

^c Java では、このような情報の欠落するキャストは行われず、エラーになります。

表 3.2 仮引数と実引数の違い

	仮引数 (formal) parameter	実引数 (actual) argument
ソース上の場所	<code>int f(<u>int x</u>) {...}</code>	<code>y = f(<u>2</u>);</code>
ソースに現れる回数	1回	何回でも
文法上の役割	変数	式(値)
具体的な値の決め方	関数が呼び出されて受け取る	呼び出すときに関数に与える

3.9 入力=引数

関数にとっての入力である引数について、もう少し掘り下げてみます。

3.9.1 仮引数・実引数

これまで、ぼんやりと「引数」と書いてきましたが、呼び出し側と、呼び出された側とは役割が違うので、表 3.2 のように、区別する呼び方があります。

仮引数 (formal parameter) 呼び出された関数から見た引数です。必ず変数であって、具体的な値は、関数が呼び出されるまでわかりません。

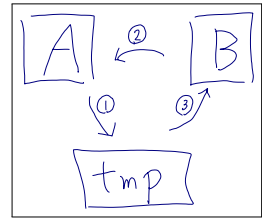
実引数 (actual argument) 関数を呼び出す側が与える値です。定数を与えている場合など、具体的な値が事前にわかることがあります。プログラムに何度現れてもよくて、その度毎に値が異なっても構いません。

この2種類は、文脈からも判断できるので、区別はそれほど重要ではありません*6が、意味が明確になる場面もあります。本書でも、ここからは必要に応じて使い分けてみます。

コラム：引数の型違い

`return` 文にも型にまつわる話題がありましたが、引数にもあります。関数呼び出しの際に、実引数と仮引数の型が異なる場合は、自動的に仮引数(受け取り側)の型にキャストされます。`double` のところに `int` の数値を渡したときはよいのですが、逆だと困るのにエラーにならないのも、`return` と同じです。

*6 英単語でも `parameter` と `argument` の区別は曖昧です。区別したとしても `formal parameter` と `actual parameter` のように、両方も `parameter` にしたり、逆に両方も `argument` にする例もみられます。



3.9.2 値渡し・参照渡し

2.7.1 節の変数の値を入れ替える動作（スワップ）を関数にしましょう。しかしソースコード 3.4 では、main 関数の変数の入れ替えが起きません。確かに swap 関数内では入れ替わっているのに、なぜでしょうか。

ソースコード 3.4 変数の値を入れ替える (?)

```

1 #include <stdio.h>
2
3 /* aとbの値を入れ替える(?) */
4 void swap(int a, int b) {
5     printf("swap:(before) a=%d, b=%d\n", a, b); // a=1, b=2
6     int tmp = a; a = b; b = tmp;
7     printf("swap:(after) a=%d, b=%d\n", a, b); // a=2, b=1
8 }
9
10 int main(void) {
11     int a = 1, b = 2;
12     swap(a, b); // aとbの値を入れ替えたい
13     printf("main: a=%d, b=%d\n", a, b); // a=1, b=2 のまま
14     swap(1, 2); // 1と2の値が入れ替わる?
15     return 0;
16 }

```

ここで思い出してほしいことは、まずは変数のスコープです。4行目のaと、11行目のaは同じ変数でしょうか、違う変数でしょうか。変数のスコープは、変数の定義されたブロック内（引数は関数のブロック内）でしたから、4行目のaは関数の終わる8行目まで、11行目のaは16行目までで、この2つは異なる変数です。

次に、12行目で swap を呼び出すときの実引数には、(a と b という変数ではなく) 1 と 2 という値がコピーされて引き渡された、ということです。受け取った swap 関数は、仮引数の a を 1 で、b を 2 で初期化して、関数ブロックを実行します。main の a と b の変数には、影響を及ぼせないのです。

表 3.3 引数の渡し方の戦略

	値渡し (call by value)	参照渡し (call by reference)
渡されるもの	値	変数を特定するアドレスやポインタ
呼び出し側の変数の変更	できない	できる
C 言語での実現方法	通常の変数	配列やポインタを利用
再帰呼び出し	できる	できない

多くの人の想像するように、呼び出し側（ここでは main）の変数まで影響が及ぶコンピュータ言語もあります。このような引数の渡し方を**参照渡し** (call by reference) といいます。対して、C 言語の採用するのは**値渡し** (call by value) といいます。

この2通りの戦略は、言語によっても違いますし、使い分けのできる場合もあるような微妙な違いですから、正確に見分けるのは難しいかもしれません。見分けるヒントとして、14行目のように `swap(1,2)` というような呼び出しをしてみましょう。これがコンパイルエラーにならなければ、そもそも1と2の値が入れ替わるはずはありませんから、12行目の a と b の値も変化しないだろうと類推できるでしょう。

それでは C 言語では、`swap` 関数のようなものは作れないのでしょうか。それには技があって、8章の配列とか、9章ポインタの知識で参照渡し相当を実現すれば、呼び出し元の変数を変更できます（[9.3.1](#) 項）。今のところは、**変数の値が変化するのは、その関数で代入したときのみ**と思っておいてください。

コラム：値渡しと再帰呼び出し

C 言語は、多くの人の想像する「参照渡し」ではなく、わざわざコピーを作って渡す「値渡し」を採用していますが、なぜでしょうか。関数の独立性が高まるというメリットはもちろんです、それだけでしょか。

C 言語が登場した頃、新しい言語の特徴として求められた機能の中には、(1) コンパイラ型 (2) 構造化プログラミング (goto 文を使わない) (3) 局所変数 (4) 再帰呼び出し、といったものがありました。この中の、再帰呼び出しを実現するために必要な機能だという側面もあるでしょう。参照渡しは「値渡し+ポインタ」で実現できるので、C 言語では値渡し一本に割り切った、という推測が成り立ちます。

再帰呼び出しは本書の守備範囲を越えますが、ある関数が自分自身を（引数を変えて）呼び出す手法で、場面によっては非常に威力を発揮します。

3.9.3 引数なし

引数のない関数は、我々が普段から書いている main 関数が代表格でしょう。「ない」ことを表すのに、関数の仮引数部分に (void) と書きます。() のように、何も書かないのとは意味が違うので注意してください (☞7.1.1 項の脚注)。

もちろん、関数を呼び出す側では、実引数を空にすればよくて、void と書いてはいけません。

コラム：引数のない関数

引数なしを示すのに、Java 言語のように、単に () と空にすればよい言語もあります。C 言語では、過去の言語規格との互換性のため、void のキーワードが必要になりました。どちらかという苦肉の策です。

3.10 練習問題

1. [関数の入出力]

ソースコード 3.2 の triangle() 関数は、三角形の面積を double で返す。引数は3辺の長さをあらわす3つの double である。このことから三角形の面積についてわかることを述べよ。(例えば、何に依存しているか、どのような値をとるのか。)

2. [関数の入出力]

右の関数 cubic(x) は x^3 を計算するが、不自然なところがある。修正方法を2通り提案せよ。

```
int cubic(double x) {
    return x * x * x;
}
```

3. [関数の組合せ]

標準ライブラリの数学関数 <math.h> に、2つの double の値 x, y の最大値を返す関数 double fmax(double x, double y) がある。これを用いて、3つの値 a, b, c の最大値を返す関数 double fmax3(double a, double b, double c) を作れ。

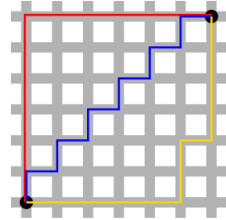
4. [関数の再利用]

次の関数を作れ。引数はすべて double 型とする。

- double norm(x, y) は、原点と点 (x, y) の間の距離を返す。
- double dist(x1, y1, x2, y2) は点 (x1, y1) と点 (x2, y2) の間の距離を返す。
- double tri(x1, y1, x2, y2, x3, y3) は、3点 (x1, y1), (x2, y2), (x3, y3) を頂点とする三角形の、3辺の長さの合計を返す。

5. [関数の再利用]

マンハッタン距離とは、マンハッタン島にあるような、格子状の道路のみを通ったときの最短距離のことである。したがって2点 $(x_1, y_1) - (x_2, y_2)$ の間のマンハッタン距離は $|x_2 - x_1| + |y_2 - y_1|$ である。(道順は右図のように何通りもある。)



以下では、このマンハッタン距離についての関数を作れ。

- `int manhattan(int x1, int y1, int x2, int y2)` は距離を返す。
- `int sum_manhattan(int x, int y)` は $(0,0) - (x,y)$ の距離と $(x,y) - (10,10)$ の距離の合計を返す。

さまざまな x, y について `sum_manhattan()` の値を調査し、どのような性質があるか調べよ。 `<stdlib.h>` にある絶対値関数 `int abs(int j)` を使ってよい。

6. [関数の再利用]

次の関数を作れ。(三角関数は A.7 節を参照せよ。)

- `double deg2rad(double deg)` は `deg` (度) をラジアンに変換する。
- `double rad2deg(double rad)` は `rad` (ラジアン) を度に変換する。
- `double sin_deg(double deg)` は `deg` (度) の正弦を返す。
- `double cos_deg(double deg), tan_deg(double deg)` はそれぞれ `deg` (度) の余弦、正接を返す。

円周率 π を表す定数は、C 言語規格には存在しないが、コンパイラの独自拡張でサポートされ、`<math.h>` に `M_PI` というマクロ定数が定義されていることが多い*7。

7. [関数の再利用と信頼性]

摂氏 (degree Celsius) と華氏 (degree Fahrenheit) の温度を変換しあう、次の1組の関数を作れ。なお、摂氏 c と華氏 f の間には $c = \frac{5}{9}(f - 32)$ の関係が成り立つ。

- `double fahr2celsius(double f)` は華氏 f を摂氏に変換する。
- `double celsius2fahr(double c)` は摂氏 c を華氏に変換する。

そして、関数を片方ずつ作る場合に比べ、同時に2個作った場合のデバッグの効率性や、作られた関数の信頼性について考察せよ。

8. [関数の組合せ]

a, b に適当な値を選んで $f(x) = ax^2 + bx$ を定義せよ。定数 ϵ をマクロで定め、 $g(x) = \frac{f(x+\epsilon) - f(x)}{\epsilon}$ を計算せよ。 ϵ を小さな値にした場合に、 $g(x)$ が $f(x)$ の微係数になっていることを確かめよ。同様に $h(x) = \frac{g(x+\epsilon) - g(x)}{\epsilon}$ を計算せよ。

9. [機能を独立させる] ㊦ 12.1.1 項

*7 Visual Studio では `#include <math.h>` の前に「`#define _USE_MATH_DEFINES`」の必要な場合がある。逆に `gcc` の `-std=c99` オプションは独自拡張をやめるので、`M_PI` が使えなくなる。

第 4 章

条件分岐

これまで書いたプログラムは、どんな値であろうが同じような計算をするものでした。しかし、例えば絶対値を返す関数を作るにはどうすればよいでしょうか。0 以上の数値であればそのまま返して構いません。しかし、負の値の場合は符号を反転させなければなりません。符号を反転させるには、例えば変数 a に対しては $-a$ と書けば良いですが、正の値の場合はこのマイナスは要りません。

二次方程式の解の個数は、判別式の値によって判定できますが、その個数を出すにはどうすればよいでしょうか。

駐車場の自動清算機は、硬貨の中にも、1 円や 5 円のように、取り扱わないものもあります。駐車料金以上のお金を受け取ると、出庫できるようにします。どれも、状況によって行なうことが違います。

この章では、「ある条件を満たすときに限って実行する文」（条件分岐）について説明します。今までは実行時に関数を呼び出して別の場所に飛ぶことはありましたが、今回は「制御構文」と呼ばれる、実行の流れを変える方法に触れます。

キーワード

- 条件分岐, 多重分岐, 入れ子
- if - else if - else
- 論理型 (boolean), TRUE / FALSE
- ド・モルガンの法則
- 短絡評価

4.1 条件分岐の if else と論理型

まずは簡単な**条件分岐** (conditional branch) です。

if 文は、最初に**条件式** (conditional expression) を検査します。成立していれば、そのすぐ後のブロック (**then 節**^{*1}) を実行します。不成立であれば、else の後ろのブロック (**else 節**) を実行します。(else 節が省略されていれば何もしません。)

```
/* 文法 */
if (条件式) {
    成立時に実行する文; ...
} else { // else 以降は省略可
    不成立時に実行する文; ...
}
```

```
/* 実例 */
if (a >= 10) {
    printf("10以上です\n");
} else {
    printf("10以上ではない\n");
}
```

条件式では、式と式の比較をします。例えば変数 a と 10 を比較する if 文は右上の例のようになります。比較に使う不等号は**比較演算子** (comparison operator) あるいは**関係演算子** (relational operator) と呼ばれ、次の 6 通りあります。

== 等しい	>= 以上	> より大きい, 超えて
!= 異なる	<= 以下	< より小さい, 未満

- 2 文字の演算子は、2 文字目が = です。
- 2 文字の演算子の間には、空白を入れてはいけません。

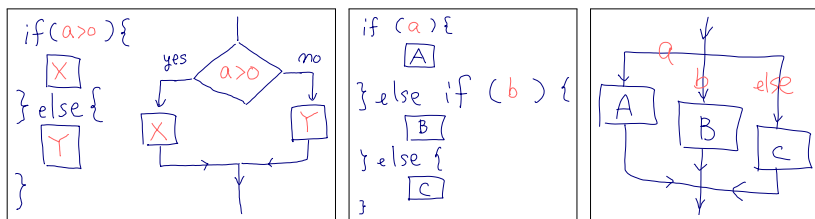
条件式の結果は、**真** (true) と**偽** (false) の 2 通りの値で表されます。比較演算子の大小関係が成立していれば真、そうでなければ偽です。このようなデータ型を、一般に**論理型** (logical datatype) とか**ブーリアン型** (boolean datatype) と呼びます^{*2}。

0 == 0 → 真	0 != 0 → 偽
1 <= 5 → 真	1 > 5 → 偽

数学では、不等号で変数の取りうる範囲を示したり、場合分けの条件を示すという使い方ができました。しかし C 言語では、その時点の変数の値を使って、値の大小関係が成立しているかどうかを検査して、真偽を決定するだけです。「x >= 10」と書いたところで、x が 10 以上の値に限定されるわけではありません。

^{*1} C 言語では then のキーワードは使いませんが、言語によっては if の条件と実行文の境目に then を挿入するので、「then 節」の用語を採用しました。

^{*2} C 言語では、本当の論理型は用意されていませんでした。(☞ 63 ページのコラム)



4.1.1 多重分岐

次は、分岐先が3つ以上の場合の書き方です。else if を繰り返します。右下の例は、年齢 x によって年代を分類しています。

```
/* 文法 */
if (条件1) {
    文1;
} else if (条件2) {
    文2;
} else if (条件3) {
    文3;
...           // 繰り返してよい
} else {      // else 以降は省略可
    文;
}
```

```
/* 実例 */
if (x <= 19) {
    printf("10代以下\n");
} else if (x <= 29) {
    printf("20代\n");
} else if (x <= 39) {
    printf("30代\n");
} else {
    printf("40代以上\n");
}
```

条件1が成り立てば、文1を実行します。それ以外で、条件2が成り立てば、文2を実行します。さらにそれ以外で、条件3が成り立てば、文3を実行します。この調子で、else if は好きなだけ続けて構いません。どの条件も成り立たなければ、最後の else 節を実行することになります。このように if - else if - else の構文は、**どれか1つの文を実行すること**を保証します。(else 節がなければ、最大で1つの文を実行します。)

コラム：ブロックの範囲を見やすくするインデント

インデント（字下げ）は、関数の範囲を見分けるのに役立ってきました。これからは、if のブロックの範囲を見分けるのにも活用します。

- { の次の行から、インデントを1段深くします。
- } の行で、深さを回復します。

```
int func(int x) {
→ if (x % 2 == 0) {
→   printf("xは偶数\n");
→   return x / 2;
} else {
→ printf("xは奇数\n");
→   return x + 1;
} ←
} ←
```

4.2 if の入れ子と論理演算

年齢 x 歳が 20 代であることを判定しましょう。条件を、20 歳以上で、しかも 30 歳未満と考えます。

```
/* if の入れ子 */
if (20 <= x) {
    if (x < 30) {
        printf("20代\n");
    }
}
```

```
/* 論理積 */
if (20 <= x && x < 30) {
    printf("20代\n");
}
```

- 左上の例では、2 個の if を組み合せました。if のブロックに、さらに if を入れても構いません。（このような「あるものの中に、同じ種類のものがもう一度含まれている構造」を入れ子 (nest) といいます。）
- 右上の例では、論理積 (and) 演算子 `&&` を用いました。`&&` の前後の 2 つの条件の両方が成立したときに限って、全体として成立したことになる（日本語の「かつ」に相当します）ので、if は 1 つですみます。

それでは 20 代以外はどのように判定すればよいでしょうか。条件としては、20 歳未満と、30 歳以上の、2 通りの可能性が考えられます。

```
/* 多重分岐 */
if (x < 20) {
    printf("20代以外\n");
} else if (x >= 30) {
    printf("20代以外\n");
}
```

```
/* 論理和 */
if (x < 20 || x >= 30) {
    printf("20代以外\n");
}
```

- 左上の例のように、多重分岐にする^{*3}と、まったく同じ printf() を 2 回書くことになります。これはあまりうまい方法とは言えません。
- 右上の例のように、論理和 (or) 演算子 `||` はうまく働きます。どちらか片方の条件でも成り立てば、全体として成り立ったことになる（日本語の「または」に相当します）ので、1 つの if で表現できて、printf() も 1 回ですみます。

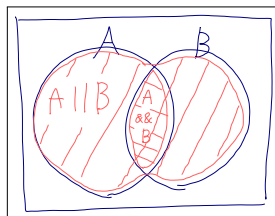
また、論理否定 (not) 演算子 `!` は真偽を反転するので、「～以外」の条件を直接的に表現できます。20 代の条件の直前に `!` を書きます。演算の優先順位が高いので、否定される条件式全体をカッコで囲います。

```
/* 論理否定 */
if (!(20 <= x && x < 30)) {
    printf("20代以外\n");
}
```

^{*3} 単純に 2 つの if を並べるだけだと、printf() が 2 回とも実行される場合があるので、else if は必須です。

表 4.1 論理演算子の真偽値

入力		演算結果		
A	B	A B	A && B	!A
偽	偽	偽	偽	真
	真	真		真
真	偽		真	真
	真	真		



論理演算（和・積・否定）の働きをまとめると、表 4.1 のようになります。

頻出ミス

比較の `a==1` は、`a=1` とよく書き間違えてしまいますが、これは代入です。しかも C 言語では、代入も演算子なので、文法上は if の条件式として正当です。

```
if (a == 1) { // ○ 比較
    printf("aは1でした");
}
```

```
if (a = 1) { // × 代入
    printf("aを1にしました!?");
}
```

比較の `!=` を `=!` と書き間違えると、これも代入と解釈されます。(4.5 節で述べるように、`!1` は 1 の否定ですから、0 です。)

```
if (a != 1) { // ○ 比較
    printf("aは1以外でした");
}
```

```
if (a =! 1) { // × 代入
    printf("aを!1にしました!?");
}
```

論理演算の `&&` や `||` も、1 文字の `&` や `|` にするとビット演算になって、エラーにならずに微妙に動作が変わります。(4.6 節)

このように、文法上はエラーにならず、気づきにくい間違いなので要注意です。gcc なら、`-Wall` のオプションで以下のような警告も出るので、活用しましょう。

```
source.c:9:5: 警告: 真偽値として使われる代入のまわりでは、
丸括弧の使用をお勧めします [-Wparentheses]
    if (a = 1) {
    ^
```

他言語に目を向けると、Java では、if の条件式を boolean (論理型) に限ることで、このような間違いをコンパイルエラーにしています。

Pascal 言語では、代入が `a:=1`、比較が `a=1` です。代入は右辺から左辺へという向きが重要ですから、コロンの向きを表現していると思えば、納得の文法です。

4.3 論理演算の組合せ

数学でよく目にする $0 \leq x < 10$ の表記は、残念ながら C 言語では通用しません。比較演算子は2つの値だけを扱えるので、 $0 \leq x$ と $x < 10$ に分解した上で、両方もも成立し欲しいので、論理積で連結して $0 \leq x \ \&\& \ x < 10$ とします。

頻出ミス

$0 \leq x < 10$ は、エラーにならずに、動作だけがおかしくなります^a。これも気づきにくい間違いですので、要注意です。

```
if (0 <= x && x < 10) { // ○
    printf("xは1桁の整数です");
}
```

```
if (0 <= x < 10) { // ×
    printf("常に成立します!?");
}
```

Java ではコンパイルエラーになります。Python では、何と正常に動作します。

^a $0 \leq x$ の演算結果は、4.5 節で説明するように、int 型の 0 か 1 になります。全体としては $0 < 10$ あるいは $1 < 10$ を評価することになり、どちらであっても成立です。

それでは、 x が「偶数か、あるいは 0 以上 10 未満」の条件は、どう記述すればよいでしょうか。「かつ」と「または」の両方が出てくるので**優先順位**が問題になります。

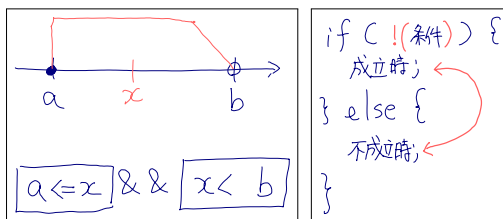
```
if (x % 2 == 0 || 0 <= x && x < 10) {...} // || より && が優先
if (x % 2 == 0 || (0 <= x && x < 10)) {...} // 同じ動作の安全策 (推奨)
```

文法としては、(積と和というだけあって) 論理積が先に計算されるので、上の 2 行は同じ動作をします。それでも、優先順位を混同する人が多いためか、覚えていなくても理解できるよう、カッコを使用することが推奨されています。

論理否定の ! は、(すでに述べていますが) 論理和や論理積よりも優先度が高いので、下の例のように「0 以上 10 未満」を否定するのにカッコが必要です。もしカッコを忘れて「 $!0 \leq \dots$ 」とするとどうなるかは、4.5 節を参考に考えてみてください。

```
if (!(0 <= x && x < 10)) {...} // 0 <= x < 10 の否定
```

論理否定で if の条件式の真偽を反転すると、then 節と else 節のブロックを入れ替えたのと同じ効果があります。これを利用して、then 節に行なうべき処理がなければ (then 節自体は省略できないので)、論理否定で真偽を反転して else 節を省略する、という使い方をよく見かけます。



4.3.1 ド・モルガンの法則

論理積や論理和に否定が組み合わさった論理式を変形するのに、便利な法則があります。

$$\overline{A \ \&\& \ B} \iff (\overline{A}) \ || \ (\overline{B})$$

$$\overline{A \ || \ B} \iff (\overline{A}) \ \&\& \ (\overline{B})$$

全体の真偽を反転する代わりに、個別の真偽を反転させた上で、論理積と論理和を入れ替えても同じ、ということです。この規則をド・モルガンの法則 (De Morgan's laws) といいます。これを覚えておくと、機械的な書き換えができます。例えば $x > 0 \ \&\& \ y > 0$ の否定は $x <= 0 \ || \ y <= 0$ とすぐわかります。

コラム：不等号の向き

条件式の「 $x > 0$ 」と「 $0 < x$ 」は、どちらもまったく同じ動作をします。このような不等号の向きをどちらにするのがよいか、考え方が何通りかあります。

```
if (x > 0 && y < 0) ...
// 変数 [不等号] 定数
```

```
if (0 <= x && x < 10) ...
// 小 [不等号] 大
```

変数・定数 左上の例のように、変数と定数の比較の場合、変数を左辺にします。

左辺のほうが主体というか、「検査対象である」という雰囲気が感じられます。両方とも変数なら、変化の頻度の多い変数を左辺にするでしょう。

数直線 右上の例のように、数直線上の大小関係を思い浮かべると理解しやすい場面では、不等号を $<$ か $<=$ のどちらかに限定して、定数を小さいものから並べます。

どちらが常によいともいえないので、場面ごとに使い分ければよいでしょう。

4.4 if を羅列する弊害

いくつも分岐があるときに、同じ条件式を（真偽の反対のものも含めて）何度も冗長に羅列していると、プログラムの構造が理解しにくくなります。else を適切に用いて、プログラムの構造を正しく表現しましょう。いくつかある文のうち「どれか1つを実行する」のか「複数個を実行する可能性がある」のかは、else のありなしでわかります。条件式を吟味してようやく動きがわかるような表記をしていると、バグの温床になります。

4.4.1 条件の網羅

x の絶対値を表示することを考えます。

```
/* if-else */ // ○
if (x > 0) {
    printf("%d", x);
} else {
    printf("%d", -x);
}
```

```
/* if の羅列 */ // ×失敗
if (x > 0) {
    printf("%d", x);
}
if (x < 0) { // !(x > 0)なら△
    printf("%d", -x);
}
```

- 左上の if-else の例では、正ならば x 、そうでなければ $-x$ を表示することで、絶対値を実現しています。2つの printf() は else で振り分けられているので、どちらか片方が必ず実行されることがわかります。つまり、(何かしらの) 値が1回だけ表示されることが、プログラムの構造から読み取れます。
- 右上の例は、else を使わずに2つの if を並べました。しかし問題が2つあります。
 - (a) どちらか片方の printf() が必ず実行されるとは、プログラムの構造からは読み取れないので、動作を理解するには、条件式を吟味する必要があります。
 - (b) 2つめの if に真偽逆の条件を書いたつもりでも、 $x = 0$ の場合が抜けています。結局のところ、冗長な記述のために条件を網羅しきれず、「 $x = 0$ のときに何も表示されない」というバグを生んでしまいました。

4.4.2 条件の網羅と関数

x の絶対値を返す関数を作ってみます。abs_ifelse(x) はもちろん正しく動作します。

abs_ifif(x) は、先ほどと同じ条件分岐にしたので、 $x = 0$ のときに return に到達しません。このため、Java であればコンパイルエラーになります。C 言語ではエラーにならず (コンパイルオプションによっては警告してくれる場合があります)、return に到達しないときに関数の返す値は不定です。(初期化忘れの変数と同じです。)

```

if (a) {
  A
} else if (b) {
  B
} else {
  C
}

```

それでは、 $x = 0$ の場合が抜けないように、例えば2つめの条件を $!(x > 0)$ としたら正しく動作するでしょうか。確かに、動作は正しくなるのですが、Java ならこれでもコンパイルエラー、C 言語でも警告は消えないでしょう。この関数が必ず `return` に到達するとは、プログラムの構造から判断できないからです。

```

int abs_ifelse(int x) { // ○
  if (x > 0) {
    return x;
  } else {
    return -x;
  }
}

```

```

int abs_ifif(int x) { // ×
  if (x > 0) {
    return x;
  }
  if (x < 0) { // !(x > 0)なら△
    return -x;
  }
}

```

右上の例では、2つめの `if` は書かずに、「`return -x;`」だけにしてしまえば、簡単に正しいプログラムになります。というのも、1つめの `if` が成立すれば `return` で関数ごと抜け出すので、その後の部分はそもそも `else` 節に入っているのと同じだからです。

4.4.3 条件に影響のある操作

x が負なら符号反転、そうでなければ2を加える、という操作をしてみましょう。この操作では x が変化するので、後続の条件判定に影響を及ぼすことに注意してください。

```

/* if-else */ // ○
if (x < 0) { // xが負なら
  x = -x; // 符号反転
} else { // そうでなければ
  x = x + 2; // 2を加える
}

```

```

/* if の羅列 */ // ×
if (x < 0) { // xが負なら
  x = -x; // 符号反転
}
if (x >= 0) { // 元が負でも
  x = x + 2; // 2を加える
}

```

左上の `if-else` の例は、もちろん正しく動作します。しかし、右上の `if` の羅列の例は、負のときに、符号反転の上に2まで加えてしまいます。一度の `if` で動作を確定してないために、両方の `if` が成立してしまうことになりました。

4.4.4 if の羅列と多重分岐

年度初めの年齢 a 才によって、身分を次の表に従って分類してみましょう。

年齢	6才～	9才～	12才～	15才～
身分	小学生低学年	小学生高学年	中学生	(義務教育修了)

右の if の羅列の例は、規則的で見通しは悪くないのですが、やはり弊害があります。「複数のものが選ばれることはない」というプログラム上の構造がすぐには読み取れません。そして、境界の年齢を変更する(あるいは、間違いに気づいて修正する)ときには2ヶ所を直す必要があるため、どちらかを直し忘れそうです。

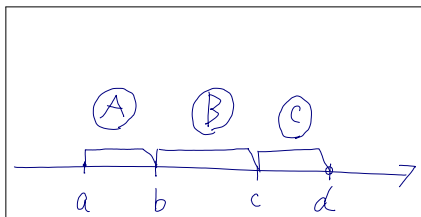
if の入れ子ではその点は改善されていて、境界の年齢の記述が1ヶ所です。(例えば「小学生高学年」の if が成立するときには、「中学生」が除外されていることに注意してください。)そして、else でつながっていることから「複数のものが選ばれることはない」という構造も(何とか)読み解けます。しかし、インデントが深くなり、見通しが悪くなりました。

やはりこのような場面では else if で多重分岐するのがよいでしょう。インデントの深さが一定に揃うので、同じ構文の繰り返しであることがすぐにわかりますし、分岐の数が増えても複雑になりません。ただし、15才以上のときに何もしないために、ブロックの中身を空にしたところは、少し技巧的です。書き忘れてないことを示すために、コメントを残しておきました。

```
/* if の羅列 */
if (12 <= a && a < 15) {
    printf("中学生");
}
if (9 <= a && a < 12) {
    printf("小学生高学年");
}
if (6 <= a && a < 9) {
    printf("小学生低学年");
}
```

```
/* if の入れ子 */
if (a < 15) {
    if (12 <= a) {
        printf("中学生");
    } else {
        if (9 <= a) {
            printf("小学生高学年");
        } else {
            if (6 <= a) {
                printf("小学生低学年");
            }
        }
    }
}
```

```
/* 多重分岐 */
if (15 <= a) {
    // 何もしない(義務教育修了)
} else if (12 <= a) {
    printf("中学生");
} else if (9 <= a) {
    printf("小学生高学年");
} else if (6 <= a) {
    printf("小学生低学年");
}
```



コラム：常にブロックを書くべし

これまで、if や else の後にはブロックを記述すると述べてきました。本当は、ブロックの中身がたった1つの文 (= 単文、つまりセミコロン (;) が1つだけ) であれば、{ } を省略して、単文だけにして構いません。とはいえ、実際のプログラム作成の現場では、作っているプログラムが刻々と姿を変えていきます。単文にしていると、後から処理を加えるときに { } で囲うのを忘れ、インデントに惑わされて制御をおかしくする、という事故が簡単に起こります。

```
/* 元のプログラム */
if (x < 0)
    x = -x; // △ if成立時のみ
printf("絶対値は%d\n", x);
```

```
/* printf() を書き加えると... */
if (x < 0)
    printf("符号を反転します\n");
    x = -x; // × ifと無関係
printf("絶対値は%d\n", x);
```

右上の例は、左上の if に printf() を書き加えたのですが、{ } で囲うのを忘れました。誤ったインデントに惑わされずに「x = -x;」が if と無関係に実行されると見抜けるでしょうか？

```
// 単文でもブロック
if (x < 0) { x = -x; } // ○
```

```
// 単文なら改行しない
if (x < 0) x = -x; // ○
```

最初から**単文でもブロック**にしておけば⁴、このような事態を防げます。少なくとも改行しなければ、ブロックが必要だと思出すでしょう。

ところで、C 言語の else if は、else 節を単文の if にして実現しているだけで、特別な文法があるわけではありません。それでも、多重分岐の構文は「else if」と覚えるのが簡便でしょう。Perl や Ruby のように、専用の構文 (elsif) が用意されている場合もあります。

⁴ Perl のように、そもそも単文が許されない言語もあります。

4.5 論理型の変数と関数

+ や / などの四則演算の演算子は、演算の結果を変数に代入できました。同じように、< や >= などの比較演算子も、結果を変数に代入できそうです。これまで、条件式の結果は論理型の「真」と「偽」のどちらかだと説明してきましたが、実際には、C 言語ではこの値を int 型の「1」と「0」にしています。ですから、代入する変数は int 型にします。

```
int x = 1, y = 2;
int c = x + y; // 四則演算を代入
```

```
int x = 1, y = 2;
int c = x < y; // 比較演算を代入
```

上のプログラムはどちらも正当です。ただ、= と < が連続して出てくるのには違和感があるので、書かなくてもよいカッコを書いて「c = (x < y);」として、条件式の結果を代入することを示す、という使い方もあります。

この性質を理解すると、次の2つのプログラムが、まったく同じ動作をすると理解できるでしょう。条件式を一旦 int 変数に代入しても、動きは同じということです。

```
if (x == 0) {
    printf("xはゼロです\n");
}
```

```
int c = (x == 0);
if (c) {
    printf("xはゼロです\n");
}
```

関数でも、変数と同じように、条件式の結果を返せます。return に書いた式が、そのまま関数呼び出しの部分に置き換わると理解しましょう。やはり右の is_zero() の return のように、書かなくてもよいカッコを書くことがあります。

```
int is_zero(int x) {
    return (x == 0);
}

if (is_zero(x)) { // (x==0) と同じ
    printf("xはゼロです\n");
}
```

条件式には、論理演算が入っても大丈夫ですから、右の is_in_range() のような関数も作れます。

```
int is_in_range(int a) {
    return (0 <= a && a < 10);
}
```

このように、論理型の変数や関数は、if の条件式に単独で現れて、比較演算のない、風変わりな表記になります。そこで、真偽値だとすぐにわかるよう、変数名や関数名に目印をつけます。つまり、is_+(形容詞) や has_+(過去分詞)、has_+(名詞) のような名前になります。この習慣を知っていれば、例えば <ctype.h> の isalpha() など^{*4}が論理型であるとわかります。

^{*4} 正確には、単語の区切りの _ (アンダースコア) が省略されています。ライブラリ関数は、昔の名残りで、文字数を節約しています。

真	TRUE (=1), 2, 3, ...
偽	FALSE (=0)

4.5.1 論理型の定数

関数の処理が複雑になると、真を返したい場面と、偽を返したい場面が、別々になることがあります。変数にどちらかの値を代入したい場合もあります。このとき、真を表すためには「1」ではなく、右上のようにマクロで定義⁵した「TRUE」を用いるのが習慣です。偽なら「0」ではなく「FALSE」です。これも、論理型であることを示す工夫です。

```
#define FALSE 0
#define TRUE 1

int is_zero2(int x) {
    if (x == 0) { // 0なら表示する
        printf("ゼロです\n");
        return TRUE;
    }
    return FALSE;
}
```

コラム：条件式の型は int 型

C 言語のように、条件式の型に `int` を流用するのは、古い言語ではよくあることでした。その C 言語でも、C99 からは 2 値に制限された `bool` 型が用意され、`true` と `false` の定数が使えるようになりました。既に C++ では `bool` 型、Java 言語では `boolean` 型が用意されていたので、C 言語もその流れに乗った形です。

本文では、この `bool` 型を紹介しませんでした。int 型で真偽値を返す関数はいくつもあるので、int 型の約束事は必須です。加えて `bool` 型の `<stdbool.h>` ヘッダの読み込みなどの互換性維持の仕掛け (☞ A.9 節) まで一度に学ぶのは重荷でしょう。言語の全体像がわかれば難しくないので、後から学んでも大丈夫です。

なお、一つの式で `bool` 型の要素を `int` 型の要素と混ぜて用いると、`int` 型に統一され、真が 1、偽が 0 に変換されます。これはこれで使い道もあるのですが、55 ページの頻出ミスのように、予期しない比較演算になることも多々あるため、Java のような混在防止の仕組みにならなかったのは残念です。

⁵ C99 より前は、真偽の定数が用意されてなかったので、プログラムで定義する必要がありました。ここでは `#define` の方法を紹介しましたが、列挙型 (`enum`) による方法もあります。

4.5.2 if に現れる論理型

ところで、論理型の変数や関数は、比較演算の結果だと理解してもらえたでしょうから、if の条件式の中で、これらの値をもう一度 TRUE や FALSE と比較するのはおかしいことにも気づいてもらえるでしょう。おかしいだけでなく、(条件式の型に int を流用しているため) TRUE と比較すると害があります。というのも、int には TRUE(=1), FALSE(=0) 以外にもとりうる値がたくさんあって、これらは条件式ですべて真と扱われることに決まっていますので、TRUE 一つだけと比較したのでは網羅しきれないからです。逆に、偽であることを判定したければ*6、直前に否定演算子 ! を置いて真偽を反転させます。

```
if (isalpha(c)) { // ○ 真の判定
    printf("cは英字\n");
}

if (!isalpha(c)) { // ○ 偽の判定
    printf("cは英字以外\n");
}
```

```
if (isalpha(c) == TRUE) { // ×
    printf("cは英字\n");
}

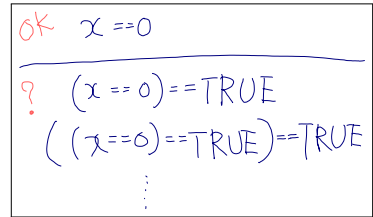
if (isalpha(c) == FALSE) { // △
    printf("cは英字以外\n");
}
```

<ctype.h> の is で始まる関数は、真として 1 以外の値を返すことで有名です。上記はその使用例で、左が正しい使い方です。ソースコード 4.1 にも使用例を挙げておきます。

ソースコード 4.1 文字の種類を見分ける

```
1 #include <stdio.h>
2 #include <ctype.h>
3
4 /* 真を返す関数名を表示する */
5 void print_true_func(int c) {
6     if (isalnum(c)) printf("isalnum('%c') ", c); // 英字か数字
7     if (isalpha(c)) printf("isalpha('%c') ", c); // 英字
8     if (islower(c)) printf("islower('%c') ", c); // 英字の小文字
9     if (isupper(c)) printf("isupper('%c') ", c); // 英字の大文字
10    if (isdigit(c)) printf("isdigit('%c') ", c); // 数字
11    if (isspace(c)) printf("isspace('%c') ", c); // 空白や改行等
12    printf("\n");
13 }
14
15 int main(void) {
16    print_true_func('A'); // isalnum('A') isalpha('A') isupper('A')
17    print_true_func('z'); // isalnum('z') isalpha('z') islower('z')
18    print_true_func('0'); // isalnum('0') isdigit('0')
19    print_true_func(' '); // isspace(' ')
20    return 0;
21 }
```

*6 FALSE との比較は、FLASE が 1 種類しかないため、(非推奨ですが) 正しく動作します。



ソースコード 4.2 に 1 桁の整数の大小比較をする関数 `print_comparison(m,n)` を作ってみました。2つの引数の範囲チェックを行なっていますが、比較の動作を 1つの関数 `is_in_range()` に共通化したのがうまいところです。

ソースコード 4.2 条件を表す関数の例

```

1 #include <stdio.h>
2
3 /* 一桁の整数なら TRUE を返す */
4 int is_in_range(int a) {
5     return 0 <= a && a < 10;
6 }
7
8 /* 一桁の数 m,n の大小比較 */
9 void print_comparison(int m, int n) {
10     if (!is_in_range(m)) { // !(0 <= m && m < 10) と同じ
11         printf("m=%d は一桁ではありません。\\n", m);
12     } else if (!is_in_range(n)) { // !(0 <= n && n < 10) と同じ
13         printf("n=%d は一桁ではありません。\\n", n);
14     } else if (m < n) {
15         printf("m=%d より n=%d の方が大きいです。\\n", m, n);
16     } else if (m > n) {
17         printf("m=%d の方が n=%d より大きいです。\\n", m, n);
18     } else {
19         printf("m と n は %d で等しいです\\n", m);
20     }
21 }
22
23 int main(void) {
24     print_comparison(-1, 5); // m=-1 は一桁ではありません。
25     print_comparison(-1, 13); // m=-1 は一桁ではありません。
26     print_comparison(0, 3); // m=0 より n=3 の方が大きいです。
27     print_comparison(1, 12); // n=12 は一桁ではありません。
28     print_comparison(10, -5); // m=10 は一桁ではありません。
29     print_comparison(4, 1); // m=4 の方が n=1 より大きいです。
30     print_comparison(1, 4); // m=1 より n=4 の方が大きいです。
31     print_comparison(1, 1); // m と n は 1 で等しいです
32     return 0;
33 }

```

4.6 論理積と論理和の短絡評価（発展的内容）

整数 $m, n (\geq 0)$ について、 m が n の倍数かどうかを判定してください。

普通に考えると、 m を n で割った余りが 0 かどうかで判断できそうです。しかし、 $n = 0$ のときは、割り算が実行できません（[2.6 節](#)）。このため、剰余計算の前に除数 n が 0 でないことを確かめる必要があります。この部分の処理を 2 通り書いてみました。

```
/* ifの入れ子 */
if (n != 0) {
    if (m % n == 0) { // OK
        printf("倍数です\n");
    }
}
```

```
/* 論理積 */
if (n != 0 && m % n == 0) { // OK
    printf("倍数です\n");
}
```

左上の if の入れ子の例は、もちろん思い通りの動きをします。 $n = 0$ ならば、2 つめの if は実行されないの、条件式の 0 の割り算を回避します。では、右上の論理積の例はどうでしょうか。驚いたことに、これも同じ動作をします。どういう仕組みになっているのでしょうか。

論理和と論理積は、全体としての結果が判明した時点で、それより後の式を検査しません。これを**短絡評価** (short-circuit evaluation) といいます。具体的な動作は次の通りです。

- A && B A が偽であれば、B を検査することなく、即座に全体として偽に決まります。
A が真であれば、B を検査して、その結果がそのまま全体の結果になります。
- A || B A が真であれば、B を検査することなく、即座に全体として真に決まります。
A が偽であれば、B を検査して、その結果がそのまま全体の結果になります。

「0 の倍数は 0 だけ」であることを加味すると、プログラムの全体はソースコード [4.3](#) のようになるでしょう。

- (A) `print_multi_nest()` は if の入れ子で、条件に無駄がありません。すべての場合分けがプログラム上に現れているので、後から処理を加えるのも簡単です。しかし、インデントが深くなって、行数も多いです。（論点が変わりますが、最初の if の条件に否定がありながら else 節もあるので、条件を反転して then 節と else 節を入れ替えたい人もいるでしょう。）
- (B) `print_multi_and()` は n の比較を 2 回行なうものの、論理積のおかげで行数が短く、しかも直後に起こる問題を回避しているというのが読み取りやすいです。ただし、一体化しすぎて、違う条件での処理を追加するには手数がかかるでしょう。

このように、どちらにも利点欠点があるので、必ずこうすべき、とは考えないでください。なお、似た演算にビット演算の `&` や `|` があります。これらは短絡評価をせず、必ず両辺

の値を得てから、ビットごとの積や和を求めます。つまり、ifの条件式で||を|と書き間違えるなどして、比較演算の結果でビット演算を行うと、この点で動作が異なります。

ソースコード 4.3 ifの入れ子と論理積の短絡評価

```
1 #include <stdio.h>
2
3 void print_multi_nest(int m, int n) { // (A) if の入れ子
4     if (n != 0) {
5         if (m % n == 0) {
6             printf("%d は %d の倍数です\n", m, n);
7         }
8     } else { // n==0 のみ
9         if (m == 0) { // 0の倍数は0だけ
10            printf("%d は %d の倍数です\n", m, n);
11        }
12    }
13 }
14
15 void print_multi_and(int m, int n) { // (B) 論理積
16     if (n != 0 && m % n == 0) {
17         printf("%d は %d の倍数です\n", m, n);
18     }
19     if (n == 0 && m == 0) { // 0の倍数は0だけ
20         printf("%d は %d の倍数です\n", m, n);
21     }
22 }
23
24 int main(void) {
25     print_multi_nest(10, 5); print_multi_and(10, 5);
26     print_multi_nest(2, 3);  print_multi_and(2, 3);
27     print_multi_nest(7, 0);  print_multi_and(7, 0);
28     print_multi_nest(0, 0);  print_multi_and(0, 0);
29     return 0;
30 }
```

コラム：ブロックのスタイル

ブロックのスタイルには、いくつかの流儀があります。自動的に整形してくれるツールもあるので、それほどこだわる必要もないのですが、意外なところに落とし穴があることを指摘しておきます。

本書では `abs1()` のように、ブロック開始の `{` を行末に配置するスタイルを採用しています。書籍の `K&R[1][2]` や、Java 公式のコーディング規約もこのタイプです。行数が節約できるので、大きさの限られたディスプレイ（あるいは紙面）で読むのに好都合です。ただし、ブロックの開始・終了の対応を探すのに少し慣れが必要なので、熟練者向けと思う人もいるでしょう。

`abs2()` のように、`{` を行頭に配置するスタイルもあります。ブロックの開始・終了の対応がわかりやすいので、初学者向けとも思えるのですが、授業を長年サポートしてきた経験からいえば、このほうが**致命的な問題**を引き起こします。

```
int abs1(int x) { // {が行末
    if (x < 0) {
        return -x;
    } else {
        return x;
    }
}
```

```
int abs2(int x) // {が行頭
{
    if (x < 0)
    {
        return -x;
    }
    else
    {
        return x;
    }
}
```

右下の `abs3()` は、`abs1()`、`abs2()` と同じく、絶対値を求めようとしているのですが、間違いがあります。初学者だけでなく、指導役の熟練者までもが見抜けずに、困り果てている姿を何度も見てきて、早く行末スタイルに慣れてもらうのがよいと思うようになりました。

答えは、行末スタイルに書き換えてもらえば、すぐにわかるでしょう。

```
int abs3(int x) // ×間違い
{
    if (x < 0);
    {
        x = -x;
    }
    return x;
}
```

4.7 練習問題

1. [1 要素による分岐]

ある美術館の入館料は 1 人 1000 円であるが、20 人以上の団体の場合は、10% 割引かれて 1 人 900 円になる。次の関数を作れ。

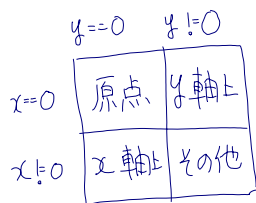
- `int total_charge(int n)` は、 n 人の団体の入館料の総額を返す。

2. [2 要素による分岐]

あるお化け屋敷は、6 才以上の子供なら 1 人で入場できるが、そうでなければ 1 人以上の大人の付き添いが必要である。いま、 m 才の子供に n 人の大人が付き添っている ($m \geq 0, n \geq 0$)。入場できるかどうかを判定せよ。

3. [多重分岐]

点 (x, y) を、「原点」「 X 軸上」「 Y 軸上」「軸上ではない」の 4 通りに、3 個の `if` と 3 個の `else` で分類して表示せよ。なお、 X 軸上と Y 軸上は、どちらも原点を除くものとする。論理積を使わない方法と、`if` を入れ子にしない方法の、2 通り作れるとよい。



4. [多重分岐]

年度初めの年齢 a 才をもとに、6 才以上なら「小学 m 年生」、12 才以上なら「中学 n 年生」、15 才以上なら「高校 n 年生」と表示せよ。年齢の上限を適切に判断して $1 \leq m \leq 6$, $1 \leq n \leq 3$ を満たせ。また `if` の個数が膨れ上がらないように工夫せよ。

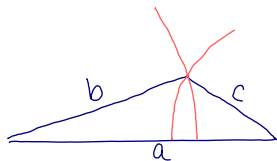
5. [条件の列举]

じゃんけんの手を、0 (=グー)、1 (=チョキ)、2 (=パー) で表す。次の関数を作れ。

- `void print_winer(int a, int b)` は、A 君の手が a 、B 君の手が b のとき、勝敗を判定して「勝者は A 君」「勝者は B 君」「あいこ」のいずれかを表示する。

6. [論理型の関数]

3 辺の長さが a, b, c (> 0) の三角形は、「 $a < b + c$ かつ $b < c + a$ かつ $c < a + b$ 」のときに存在し、そうでなければ存在しない。これを判定する、次の関数を作れ。



- `int exist_triangle(int a, int b, int c)` は、3 辺の長さが a, b, c の三角形が存在すれば `TRUE`、そうでなければ `FALSE` を返す。簡単のため $a, b, c > 0$ を前提とする。`TRUE` と `FALSE` は必要に応じて定義せよ。

7. [0 以外]

x 円を何枚かの硬貨で払いたい。次の関数を作れ。

- `void print_coins(int x)` は、合計が x 円になる、各硬貨の枚数を表示する。
 - 使用する硬貨は 500 円, 100 円, 50 円, 10 円の 4 種類
 - x は 10 以上 990 以下の 10 の倍数
 - 各硬貨の使える枚数は十分にある
 - 0 枚のものは表示しない
 - 硬貨の合計枚数が最小となる払い方を選ぶ

(ヒント)

- 高額のものから枚数を決定すれば、自動的に合計枚数が最小になる。
- 枚数を決定したら、 x からその金額を引く。次の硬貨で (残った) x 円を支払うと考える。これを、4 通りの金額で、4 回繰り返せばよい。
- 金額部分を変数にすると、完全に同じ処理になる。

実行例

40 円を支払います。
10円硬貨 4 枚

50 円を支払います。
50円硬貨 1 枚

90 円を支払います。
50円硬貨 1 枚
10円硬貨 4 枚

100 円を支払います。
100円硬貨 1 枚

490 円を支払います。
100円硬貨 4 枚
50円硬貨 1 枚
10円硬貨 4 枚

500 円を支払います。
500円硬貨 1 枚

コラム：境界値分析

プログラムの間違いを探すための効率的な手法の一つに、**境界値分析** (boundary value analysis) というものがあります。ある条件の成立する (あるいは成立しない) 境界近くのギリギリの数値を重点的に試すというものです。if で使われているであろう条件式のイコールの有り無しが正しいかをあぶり出すわけです。

例えば 4. の問題であれば、11 才で「中学 0 年生」、12 才で「小学 7 年生」のような間違いが起きないかを疑うと、効率的にバグを見つけられるでしょう^a。

7. の問題では、例えば 40 円と 50 円では、使用する硬貨がすっかり切り替わります。このような切り替わりの境界をしっかりと試してみましょう。

^a これには前提があって、例えば「小学 m 年生」の 6 学年分の処理が共通であることを期待しています。学年ごとの if が 6 回列挙されているとすれば、もっと細かく検査せねばなりません。

第5章

繰り返し処理 (1)

駐車場には、何台も車が停められて、利用者が来るたびに、同じように清算処理をする必要があります。同じ処理を繰り返すことは、機械やコンピュータの得意な動作です。

プログラムで、状況によって行なうべき動作を切り替える方法は、もうすでに知っています。もう一歩、プログラム上の実行すべき場所をジャンプする方法がわかれば、繰り返し処理が実現できます。

この章では、ある条件を満たす限り同じような処理を繰り返す構文について説明します。同じメッセージを何度も表示したり、規則的に変わっていくメッセージを表示したり、ということが簡単にできるようになります。

キーワード

- for ループ・while ループ
- ループ変数
- インデント
- 個数・合計・漸化式

5.1 while 文による繰り返し

条件を満たす限り実行する繰り返し処理は、while 文で次のように書けます。

```
/* 文法 */
while (条件式) {
    文;
    ...
}
```

```
/* 実例 */
while (x < 10) {
    printf("%d\n", x);
    x++;
}
```

if とよく似ていて、while の後に、条件式と、実行するブロック { } を書きます。if では条件式が成立したときにブロックを1度だけ実行しましたが、while では条件式が成立する限り、ブロックを何度も繰り返して実行します。繰り返し実行することを「ループ (loop) する」といいます。

ソースコード 5.1 while で 2 の累乗を列挙

```
1 #include <stdio.h>
2
3 int main(void) {
4     int i = 1;
5     while (i < 100) {
6         printf("%d ", i);
7         i = i*2;
8     }
9     printf("\nループ後のiの値は%dです\n", i);
10    return 0;
11 }
```

while の簡単な例 (ソースコード 5.1) で 2 の累乗を表示してみましょう。

ソースコード 5.1 の実行結果

```
1 2 4 8 16 32 64
ループ後のiの値は128です
```

5 行目の while の条件が成立すれば、6-8 行目を実行した上で、また 5 行目に戻ります。条件が不成立ならば 9 行目にジャンプします。変数 i の値に着目してみましょう。4 行目で 1 に初期化され、7 行目を実行するたびに 2 倍になります。while の条件は 100 より小さいことですから、6-8 行目を何度か実行した後にループを抜け出して、9 行目に達するのは i が 128 になったときです。抜け出すときには、もう 6 行目の表示が行われていないことに注意してください。

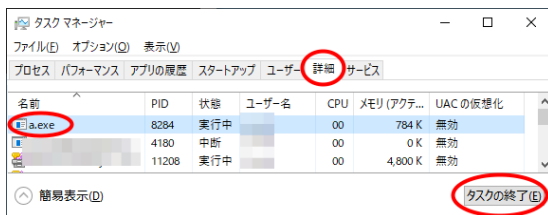


図 5.1 Windows のタスクの終了方法

次に「 x を越える最小の自乗数」を求め関数を作ってみましょう。ソースコード 5.2 のようになります。min_square() は、引数 x を受け取り、 $k = 1, 2, 3, \dots$ について k^2 と x の値を比較して、 x を越えたところでループを終了して、最後に k^2 を返します。

ソースコード 5.2 x を越える最小の自乗数 (while 版)

```

1 int min_square(int x) {
2     int k = 1;
3     while (k*k <= x) {
4         k++;
5     }
6     return k*k;
7 }

```

コラム：プログラムの強制終了

ループ処理は、ちょっとしたミスで終わらなくなることがよくあります。この手の無限ループを実行してしまうと、プログラムを強制的に停止させる必要があります。どのような操作をするかは OS や環境に依存するのですが、Cygwin や Linux のターミナルでは **Ctrl+C** (**Ctrl**を押したまま**C**を押す) です。Visual Studio のような統合開発環境では、停止のためのボタンが GUI 上に用意されています。

それでも停止してくれないプログラム (プロセス) には、最後の手段があります。Windows ならタスクマネージャ、Unix 系では kill コマンドや top コマンドの使い方を調べてみてください。

頻出ミス

Cygwin のターミナルで **Ctrl+Z** を押すと、プログラムは終了したように見えても休止しているだけで、再開を待っています。Windows では休止している実行ファイルを上書きできないので、gcc がおかしいエラー (Device or resource busy) を出すようになります。fg コマンドでプログラムを再開して、**Ctrl+C** で停止しなおすか、図 5.1 のようにタスクマネージャで a.exe を終了しましょう。

5.2 決まった回数の繰り返しとループ変数

プログラムを作る上で、「同じ処理を10回繰り返す」という場面がよくあります。whileで実現すると、次のようになるでしょう。

```
int i = 0;
while (i < 10) {
    処理;
    i++;
}
```

変数*i*の働きに着目してみましょう。初期値は0で、処理を1回するごとに1増えるという、規則的な動きをします。そのため、*i*が10になれば、目的の処理を10回実行したと判断できるので、*i*が10以上でループ処理をやめる(*i*が10未満でループする)ことで10回ループが実現できます。このように、変数*i*はループの回数を制御する重要な役割をしているので、このループの**ループ変数** (loop variable) と呼ばれます。

5.3 for 文による繰り返し

「同じ処理を10回繰り返す」ために、もう一つの手段があります。for文です。

```
for (int i=0; i<10; i++) {
    処理;
}
```

for文の文法は、次の通りです。whileで実現した場合と対比してみます。for文の()の内側の3つの式は、いずれも空欄でも構いません。空欄にしても2個の;(セミコロン)は残したままにします。

```
for (初期化式; 条件式; 増分式) {
    文;
    ...
}
```

```
初期化式;
while (条件式) {
    文;
    ...
    増分式;
}
```

forとwhileは、上のような対応関係をみると相互に書き換えができそうです。対応するものがなければ空欄にしてもよいのですから^{*1}。では、なぜ同じような働きをする構文が2つもあるのでしょうか。

^{*1} 1点だけ例外があって、whileの条件式は空欄にはできません。このことは6.2.1節に影響があります。

for では、ループ変数の出現する式がまとまっているという特徴があります。1ヶ所を見るだけでループの動きがすぐにわかります。そして同じループ変数が3回出現することで、間違いのないループであることを視覚的にとらえることができます。逆に言えば、for (i=0; j<10; k++) はおかしなループだと、反射的に判断できるわけです。

while では、増分式をループ処理の最後に書きます。ループ処理が長くなるとループ変数の登場する3つの式が離れてしまうので、書き忘れたり変数名を間違えたりしがちです。ですから典型的なループでは for が好まれます。

while は長い条件式を書くのに適しています。あるいは、ループ変数が明確に決まっていないとか、ループ回数が事前に決まっていらないような時によく使われます。

プログラマがループ処理を書く時に、最初に考えることは「ループ変数を何にするか」です。特別の理由がなければ i, j, k といった短い変数名が使われます。^{エス}1は1と見間違えるので避けましょう。

コラム：インクリメントの順序

前置の「++x;」と後置の「x++;」は、単独なら1加えるという、同じ働きです。式の中に現れると、その「値」がインクリメントの後か前か、どちらになるかが違います。

<pre>// 増加が前 y = ++x; ↓ ++x; y = x;</pre>	<pre>// 増加が後 y = x++; ↓ y = x; x++;</pre>
---	---

しかしC言語には、2項演算子の2つのオペランドや、関数の引数⁴など、評価順序に保証のない場面も多くあります。処理系依存の動作になりがちなので、単純な使い方にとどめておきましょう。

```
int x = 0; /* 処理系依存の例 */
int y = (++x) - (++x); // × 1-2(=-1) または 2-1(=1)
printf("%d:%d", ++x, ++x); // × "3:4" または "4:3"
```

⁴ 関数の引数の区切りのカンマは、カンマ演算子（順次評価）ではありません。

5.4 ループ処理の実例

ここでは、典型的なループの使用例を見ていきましょう。

5.4.1 偶数を表示

1 から n の整数の中から、偶数のみを表示してみましょう。ソースコード 5.3 に3つのパターンを用意してみました。

ソースコード 5.3 偶数を表示する

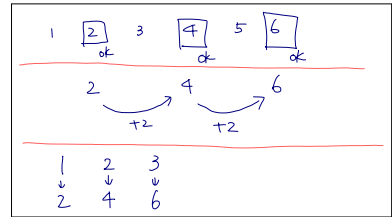
```
1 void print_even(int n) {           // (A)
2     for (int i=1; i<=n; i++) { // 通常のループ
3         if (i % 2 == 0) {         // 偶数を選び出す
4             printf("%d ", i);
5         }
6     }
7 }
8
9 void print_even2(int n) {          // (B)
10    for (int i=2; i<=n; i+=2) { // 2ずつ増やす変則ループ
11        printf("%d ", i);
12    }
13 }
14
15 void print_even3(int n) {          // (C)
16    for (int i=1; 2*i<=n; i++) { // 2*i で偶数を作り出す
17        printf("%d ", 2*i);      // 表示も 2*i
18    }
19 }
```

(A) `print_even()` では、 n 回ループの中で、条件分岐で偶数を選び出しています。

(B) `print_even2()` では、少し変則的なループで実現しています。ループ変数を2から始めて、2ずつ増やすことで、偶数だけを生成しています。 n が奇数でも偶数でも、どちらでも正しく動きます。

(C) `print_even3()` では、計算式で偶数を作り出すことにしました。 i が整数なら $2*i$ は偶数です。ですから i の代わりに $2*i$ と書けば偶数になります。 $2*i$ と書くところは、ループの条件と、表示の2ヶ所あります。

今はこの式が単純なのでよいですが、複雑な式だと、複数の場所に同じ式を書く手間違えそうになるので、多用するのは考えものです。



頻出ミス

下の `print_even_bad()` のような処理はすすめられません。この `for` ループは、1 から $(n-1)$ までの $(n-1)$ 回のループをしているように見えますが、ループ処理中でループ変数の `i` を変更して偶数を作り出しています。つまりループ回数は $(n-1)$ ではありません。

```
void print_even_bad(int n) { // 非推奨
    for (int i=1; i<=n-1; i++) { // 通常のループのように見えて
        if (i % 2 == 1) i++; // 奇数なら1増やす(!)
        printf("%d ", i);
    }
}
```

これでは `for` ループの「ループ変数がまとまって現れて、どのようにループするかが一目で分かる」という特徴を台無しにしています。

ループ変数を変更するなどという不可解な動作は、くれぐれも避けましょう。

コラム：手書きのメモ

プログラムの動きが複雑になってくると、いくらパソコンの画面上でソースコードと睨めっこしていても埒が明きません。

紙の上にメモを書いて、最初は具体的な値で、変数の変化の様子を追いかけてたり、プログラムの制御の具合を書いてみたりして、徐々に法則性を見出しましょう。この作業はとても重要です。

5.4.2 個数を数える

整数 n の約数の個数を数えてみます。 n の約数とは、1 から n の間の整数のうち、 n を割り切るものですから、1 から始まる n 回のループの中で、それぞれ割った余りを計算することで約数かどうかわかります。約数を見つけるたびに 1, 2, 3, ... とカウントアップしていきましょう*2。ループ回数が明確に決まっているので for を用いることにして、ループ変数は i にします。そして、**個数を数えるための変数**が別に必要になります。

ソースコード 5.4 約数の個数

```
1 #include <stdio.h>
2
3 int n_divisor(int n) {
4     int count = 0; // 個数を数える変数
5     for (int i=1; i<=n; i++) { // 総当たりする
6         if (n % i == 0) { // 割り切るなら
7             count = count + 1; // 個数を1増やす (count++と同じ)
8         }
9     }
10    return count; // 数えた個数を返す
11 }
12
13 void print_n_divisor(int n) {
14     printf("%dの約数は%d個あります\n", n, n_divisor(n));
15 }
16
17 int main(void) {
18     print_n_divisor(7); // 2個 (1,7)
19     print_n_divisor(15); // 4個 (1,3,5,15)
20     return 0;
21 }
```

ソースコード 5.4 では、個数を記憶するのに変数 `count` を用意して (4 行目)、最初はまだ約数を見つけていないので 0 で初期化しておきます。5-9 行目のループでは、 n を割り切る数値 i を見つけるたびに (6 行目)、`count` を 1 増やします (7 行目)。ループが終わると (10 行目)、すべての約数を見つけているはずなので、その個数を返します。

*2 素因数分解 $n = 2^a 3^b 5^c \dots$ から $(a+1)(b+1)(c+1)\dots$ 個と計算する方法もありますが、ここでは愚直に総当たりで数えることにします。


count = 0	0 ← 0
count ++	0 → 1 ++
count ++	1 → 2 ++
count ⇒ 2	2

ソースコード 5.4 の実行結果

```
7の約数は2個あります
15の約数は4個あります
```

実行結果は、7は素数で、約数は1と7だけですから、正しく2個と求まっています。15は1, 3, 5, 15が約数で、4個です。

関数名 `n.divisor()` の接頭辞 `n_` は、ソースコード上では **number of** の意味でよく使われます。全体で「divisor の個数」を表している、プログラマなら誰でもわかります*3。

 頻出ミス

ソースコード 5.4 の4行目の `count` 変数の初期化（ここでは0の代入）を忘れると、とんでもないおかしな値になります。運が悪い(?)と正常に動作することもあります。何度も関数を呼び出しているとそのうちおかしくなります。未然に防ぐために、コンパイラに警告オプションをつけるなどの工夫をしましょう。

複雑な処理になると、変数を使う直前に初期化することも重要になります。忘れていないことを簡単に確認できるようにするためです。

コラム：インデント（字下げ）

ソースコード 5.4 のような入り組んだ処理になってくると、インデントの重要性が増してきます。5行目の `for` のブロックの終わりが9行目だということがすぐにわかるように、6-8行目は行の先頭に4文字ほどスペースを多めに入れて字下げしておきます。6行目の `if` のブロックの終わりも8行目ですから、その間はさらに4文字余分に空けます。このような整形は機械的に行えるので、テキストエディタの編集支援機能を使いこなしたいものです。

*3 英文的には `number of + 名詞の複数形` とすべきところですが、プログラム上は `n_ + 単数形` もよく見かけます。

5.4.3 合計を計算する

次に、個数を数える代わりに、約数の合計を計算してみます。

ソースコード 5.5 約数の和

```
1 #include <stdio.h>
2
3 int sum_divisor(int n) {
4     int sum = 0;
5     for (int i=1; i<=n; i++) {
6         if (n % i == 0) {
7             sum = sum + i;
8             printf("i=%d, sum=%d\n", i, sum); // デバッグライト
9         }
10    }
11    return sum;
12 }
13
14 void print_sum_divisor(int n) {
15     printf("%dの約数の和は%dです\n", n, sum_divisor(n));
16 }
17
18 int main(void) {
19     print_sum_divisor(7); // 8 (1,7)
20     print_sum_divisor(15); // 24 (1,3,5,15)
21     return 0;
22 }
```

ソースコード 5.5 では、ソースコード 5.4 とループの構造は同じですが、count の代わりに、合計値を累積するための変数 sum を用意します (4 行目)。約数を見つける前なので、やはり 0 で初期化します。約数が見つかると、sum を約数の値だけ増やします (7 行目)。

ソースコード 5.5 の実行結果

```
i=1, sum=1
i=7, sum=8
7の約数の和は8です
i=1, sum=1
i=3, sum=4
i=5, sum=9
i=15, sum=24
15の約数の和は24です
```


$$\begin{array}{l}
 \text{sum} = 0 \quad \boxed{0} \leftarrow 0 \\
 \text{sum} += 1 \quad \boxed{0 \rightarrow 1} + 1 \\
 \text{sum} += 7 \quad \boxed{1 \rightarrow 8} + 7 \\
 \text{sum} \Rightarrow 8 \quad \boxed{8}
 \end{array}$$

実行結果は、「和は 24 です」だけでは正しいかどうかはわかりにくいので、8 行目で変数の値も表示させてみました。約数が見つかるたびに、*i* が見つけた約数、*sum* がそれまでの和として表示されます。これを見ると、*sum* は突然得られるわけではなく、ループ処理の中で徐々に計算が進むことがわかります。この変化の過程をよく理解しておきましょう。

コラム：デバッグライト

ソースコード 5.5 の 8 行目のような変数表示のコードは**デバッグライト** (debug write) と呼ばれます。条件が成立したり、変数の変化するタイミングなどで表示させて、プログラムが思い通りに動作していることを確かめる (あるいは、どちらかという、思い通りに動作しないときに原因を探る^{a)}) ためのものです。

プログラムが完成してしまえば無用にも思えるのですが、次に改造することがあれば役立ちますので、消してしまわずに、**コメントにして残しておきます**。

表示形式には「**関数名: 変数名=値, 変数名=値**」のようなパターンが好まれます。行頭の関数名は、通常の出力と違うことの日印でもあります。いちいち関数名を書いていると間違えそうですが、C99 からは、現在の関数名を表す `__func__` という特殊な変数が用意されたので、以下のように簡便に書けるようになりました。

```
printf("%s: i=%d, j=%d\n", __func__, i, j);
```

^a このようなデバッグ手法は**プリントデバッグ** (print debug) と呼ばれます。

5.4.4 平方根で3乗根を求める (漸化式)

電卓の中にはルート (平方根・2乗根・自乗根) の計算のできるものがあります。しかし3乗根まで計算できるものは、まず見つかりません。そのような電卓でも3乗根の近似値を計算する方法がありますので、プログラムで実現してみましょう。

正の実数 x に対して、次の漸化式で与えられる数列 $\{a_n\}$ を考えてみます。

$$a_1 = 1, \quad a_n = \sqrt{\sqrt{x} \cdot a_{n-1}} \quad (n = 2, 3, 4, \dots)$$

このとき、 a_∞ は $\sqrt[3]{x}$ に収束して、しかも収束が早いので a_4 くらいでも良い近似値になります。 a_4 の計算は、電卓にルートとメモリ機能があれば簡単な操作で行えます。

この計算を C 言語に翻訳してみます。ソースコード 5.6 に3通りの関数を作りました。ルート計算には `<math.h>` の `sqrt()` を使います。

(A) `cbrt4()` では、電卓での a_4 の計算をそっくり再現しました。

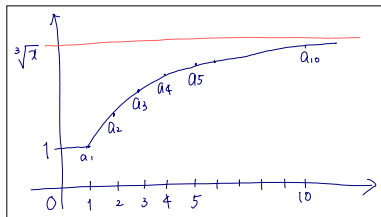
ところで、変数 `a3` や `a4` の役目をよくよく考えると、 a_4 を求めるのに必要なのは1つ前の a_3 の項のみで、それより前は不要です。それならば、異なる変数を用意する必要はなく、すべて同じ1つの変数を使いまわしできそうです。

(B) `cbrt10()` では、数列に同一の変数を使いまわせることに気づいたので、ループを使いました。数列の次の項を求めるところが、ループ処理の1回分に対応します。繰り返しはプログラムの得意とするところですから、 a_4 とわずかに a_{10} を求めてみましょう。このように、**回数が決まったループには for** が適しています。

(C) `cbrt_eps()` では、ループのもう一つの考え方を採用しました。今、3乗根を求めようとしているのですから、理想的な値が手に入れば、その値の3乗が元の x と等しくなります。始めのうちは誤差がありますが、計算を続ければより正しい3乗根に近づきます^{*4}。それならば、許容できる x との誤差をあらかじめ設定しておき、誤差がそれ以上の間は計算を繰り返す、という作戦も考えられます。この場合のような、**条件によるループには while** が適しています^{*5}。23行目の `while` の条件式の `fabs()` は絶対値を返す数学関数で、やはり `<math.h>` で定義されています。20行目で許容誤差をあらわす `EPS` というマクロを定義しています。暫定解 a の3乗と x の差の絶対値が許容誤差を超える間、つまり $|a^3 - x| > EPS$ の間にループします。

^{*4} この漸化式はうまく収束してくれますが、どんな漸化式でもある値に収束するわけではありません。

^{*5} 許容誤差を小さくしすぎると、その精度が達成できずに無限ループになる可能性があるため、実用的にはループ回数に上限を設けます。



ソースコード 5.6 3乗根を近似的に求める

```

1 #include <stdio.h>
2 #include <math.h>
3 // 以下は電卓での操作 ([MR]はメモリ呼び出し)
4 double cbrt4(double x) { // (A) // xをメモリに記憶
5     double a1 = 1.0; // [1]
6     double a2 = sqrt(sqrt(x * a1)); // [×][MR][=][√][√]
7     double a3 = sqrt(sqrt(x * a2)); // [×][MR][=][√][√]
8     double a4 = sqrt(sqrt(x * a3)); // [×][MR][=][√][√]
9     return a4;
10 } // よく考えるとa1,a2,a3,a4は同じ変数でもよい
11
12 double cbrt10(double x) { // (B) // xをメモリに記憶
13     double a = 1.0; // 初項a1 // [1]
14     for (int i=2; i<=10; i++) {
15         a = sqrt(sqrt(x * a)); // [×][MR][=][√][√]
16     }
17     return a;
18 }
19
20 #define EPS 0.001 // 許容誤差
21 double cbrt_eps(double x) { // (C)
22     double a = 1.0;
23     while (fabs(a*a*a - x) > EPS) { // aの3乗とxの差の絶対値を比較
24         a = sqrt(sqrt(x * a));
25     }
26     return a;
27 }
28
29 int main(void) {
30     double x = 8.0;
31     printf("cbrt=%f, cbrt4=%f, cbrt10=%f, cbrt_eps=%f\n",
32           cbrt(x), cbrt4(x), cbrt10(x), cbrt_eps(x));
33     return 0;
34 }

```

ソースコード 5.6 の実行結果

```
cbirt=2.000000, cbirt4=1.978456, cbirt10=1.999995, cbirt_eps=1.999979
```

実行結果には、`<math.h>` の本物の 3 乗根関数 `cbirt()` の値も表示させてみました。8.0 の 3 乗根を求めているので、正確には 2.0 ですが、いずれも良い近似値が得られていることがわかります。

コラム：ノイマン型アーキテクチャあるいはプログラム内蔵方式

現在のようなコンピュータ言語の登場する前には、歯車による機械式計算機や、真空管やリレーによる計算機がありました。一種の電卓のようなもので、ボタンやレバーうまく操作することで望みの計算を実現していました。このコンピュータ（+人間）の動作は、ソースコード 5.6 の `cbirt4()` と似ているように思えます。

その後、ノイマン型アーキテクチャとよばれる計算機が登場しました。プログラム内蔵方式とも呼ばれ、人間の行っていた操作をデータのようにメモリに記憶しておきます。そして状況次第で動作を切り替えるという、条件分岐に相当する機能も実現されました。これが大きな進歩で、`cbirt_eps()` のような動作ができるようになりました。

現在のほとんどのプログラミング言語は、この系譜に則っています。「プログラム上に現在実行中のところがあって、だんだんと進んでいく」という考え方は、ここからきています。

コラム：0 オリジン vs. 1 オリジン

0 から数え始めることを、日本人のプログラマはよく **0 オリジン** といますが、英語では `zero origin` よりも `zero-based` のほうが通じやすいようです。1 始まりの **1 オリジン** (`one-based`) も同様です。

日常生活では、1 番目 2 番目と、1 オリジンで数えることが多いですが、C 言語の配列 (☞ 8 章) を 0 から数え始めるため、0 オリジンもよく出てきますし、計算にはこちらが向いています。

0 オリジンに慣れてしまえば、まったく難しくないので、日常生活の 1 オリジンの数字とギャップが出てくるため、この違いを埋め合わせるのに苦勞するのは、プログラマの宿命でしょう。

5.5 良いループと悪いループ

10 回ループの書き方は何通りもあるのですが、良いものと悪いものがあります。

```
for (i=0; i<10; i++) { /* 10回 */ } // ○
for (i=0; i<=9; i++) { /* 10回 */ } // ×
for (i=1; i<=10; i++) { /* 10回 */ } // ○
for (i=1; i<11; i++) { /* 10回 */ } // ×
```

×のついているものがなぜ悪いのか、一目瞭然ですね。確かに 10 回のループではありますが、ソースコード上には「10」という数字が現れていませんから、これではプログラムの書き手の意図がわかりません。

では、「10」という数字をソースコード上に素直に書くためには、どのようなループを書けばよいでしょうか。2 通りのパターンがあります。

- (0 ~ 9) ループ変数を 0 から始めて、条件は < (イコールなし)
- (1 ~ 10) ループ変数を 1 から始めて、条件は <=

いま仮に、本当に (n+1) 回のループをさせたい場面があったとしましょう。それならばソースコードには (n+1) と表記すべきであって、n に = をつけたり取ったりして調節すると誤解を招くということです。(n+1) 回ループの例を挙げます。

```
for (i=0; i<n+1; i++) { /* (n+1)回 */ } // ○
for (i=0; i<=n; i++) { /* (n+1)回 */ } // ×
for (i=1; i<=n+1; i++) { /* (n+1)回 */ } // ○
```

0 から始める (0 オリジン) と、1 から始める (1 オリジン) のどちらにするかは、場面によって判断します。このあとに学ぶ配列を扱うなら 0 オリジンが向いていますし、日常生活の数を扱うなら 1 オリジンが簡単です。

5.6 練習問題

1. [ループの実行順序]

ソースコード 5.1 の 6 行目と 7 行目を入れ替えると、実行結果はどのように変化するか。

2. [デバッグライト]

ソースコード 5.6 にデバッグライトを追加して、変数の変化を観察してみよ。cbrt_eps(x) のループ回数も確かめてみよ。

3. [条件を満たすものを表示]

376 は特別な数である。2 乗した値 $376^2 = 141376$ の下 3 桁が、元の数に一致する。このような 3 桁の整数を、総当たりですべて求めて表示せよ。

4. [条件を満たすものを表示]

1 から 50 までの整数のうち、3 の倍数と、一の位あるいは十の位に 3 が現れるものを、小さいものから順に「3 6 9 12 13 15 ... 30 31 ...」のように表示せよ。(☞ 2 章の練習問題 6.) 同じ数が何度も表示されないよう、条件を工夫せよ。

5. [回数の決まったループ]

次の関数を作れ。いずれもループで積算せよ。

- int sum(int n) は n 以下の正の整数の合計を返す。
- int sum_odd(int n) は n 以下の正の奇数の合計を返す。

6. [ループ処理中で自作関数を呼び出す]

約数の合計と、その数自身が一致するものを完全数という。ただし、約数の合計からはその数自身を除くものとする。ソースコード 5.5 を参考に、10000 以下の完全数 (4 個ある) をすべて求めて表示せよ。

_____ 完全数の最初の 2 個は 6, 28 である。5 個目を求めるには、かなりの工夫が必要である。現在のところ 50 個ほどだけが知られている。_____

7. [数列の和]

S_n ($n = 0, 1, 2, \dots$) を以下で定義する。

$$S_n = \sum_{k=0}^n \frac{(-1)^k}{2k+1} = \frac{1}{1} - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \dots$$

$n = 0, 1, 2, \dots, 100$ について、 $4 \cdot S_n$ を表示せよ。なお、ライプニッツの公式によって $4 \cdot S_\infty$ が π (円周率) に収束することが知られている。

_____ $(-1)^k$ の計算には、累乗関数を用いるまでもない。 k の偶数奇数で +1 と -1 を使い分けてもよい。符号を記憶する変数 `sign = 1` を用意して、ループごとに `sign = -sign;` と反転させてもよい。_____

第6章

繰り返し処理 (2)

ここでは繰り返し処理のより高度な使い方を学びます。複数の繰り返しを入れ子にしたり、途中で終了したり、途中から次の繰り返しへ飛ばす制御が可能です。

世の中の問題は、数学のように美しく解けるものばかりではありません。愚直に総当たりで数えたり、条件に合うものを探し出したりする場面があるかもしれません。そのような問題に対して、繰り返し処理を使いこなせば、強力な武器となるでしょう。

キーワード

- 2重ループ
- 無限ループ
- break・continue
- ある?ない?

6.1 2重ループ

for や while のループ処理のブロックの中では、入れ子（ネスト）にして、さらに for や while のループ処理が行えます。

```
for (int i=1; i<=3; i++) { // 外側のループ (1,2,3)
    for (int j=5; j<=7; j++) { // 内側のループ (5,6,7)
        printf("i=%d, j=%d\n", i, j);
    }
}
```

このように入れ子になった2つのループを**2重ループ**といい、最初のループを「外側のループ」、ループの中のループを「内側のループ」などと呼びます。ループ変数は**ループ毎に別のも**のを用意する必要があります。

右の実行例を見ると、2重ループの中で i と j の値の組合せ（ここでは9通り）がすべて網羅されることがわかります。そしてループ変数は**内側のほうが早く変化**することも理解しておきましょう。

```
i=1, j=5
i=1, j=6
i=1, j=7
i=2, j=5
i=2, j=6
i=2, j=7
i=3, j=5
i=3, j=6
i=3, j=7
```

もしも2重ループでループ変数を共有すると、ループの条件判定がおかしくなってしまいます。次の例を見てみましょう。

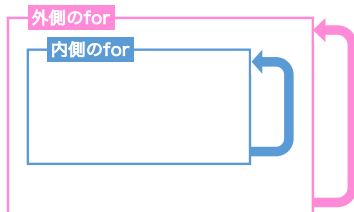
```
int i; // 共通のループ変数! ?
for (i=1; i<=3; i++) { // 外側のループ (1,2,3)
    for (i=5; i<=7; i++) { // 内側のループ (5,6,7)
        printf("i=%d, i=%d\n", i, i);
    }
}
```

右の実行例では、外側のループが実行されていないように見えます。外側のループ変数が、内側のループで上書きされています。

```
i=5, i=5
i=6, i=6
i=7, i=7
```

このように、内側のループでは新しいループ変数を用意する必要がありますが、重複しても**エラーにも警告にもなりません**。初めの例のような、for の初期化式に変数定義も含めるスタイルが安心^{*1}です。

^{*1} 安心の理由は、(残念ながら) 変数名の重複を検査できるわけではなく、(驚いたことに) 重複しても正しく2重ループの動作をすることにあります。重複してもよい理由は7.2.3項でわかります。



6.1.1 for のループ変数のスコープ

すでに活用していますが、C99 からの新しい機能で、for の初期化式で変数定義を行えるようになりました。この変数のスコープは、for ループのブロック終了まで^{*2}です。

```

for (int i=0; i<n; i++) { // 初期化式で i を定義
    ...;
}
// ここでは上の i は無効
for (int i=0; i<n; i++) { // i を繰り返し使える
    ...;
}

```

↑ i のスコープ

↑ i のスコープ

このおかげで、別の for ループを続けて書くときに、同じループ変数名を繰り返して使えます。「スコープは狭く」の精神 (C42 ページのコラム) にも合致しますし、C++ や Java といった多くの言語でも同様に実現されている機能なので、ぜひ活用したいものです。

コラム：多重ループ

2重ループと同じようにして、3重、4重、... のような、**多重ループ**も実現可能です。ただし動作がわかりにくくなるので、実際には4重、5重のような深いループはあまり使われません。もしそのような動作をさせたいならば、見通しがよくなるよう、例えば内側の2重ループ部分を関数に分離して、3重ループにあたる場所では関数を呼び出す、などの工夫を行って、表面的には深い多重ループにならないようにするとよいでしょう。

^{*2} このスコープには別の考え方もあります。過去にはある有名 C++ コンパイラが、「その for ループの一つ外側のブロックまで通用する」としたことがありました。確かに 6.3.4 節のような場面でメリットはあったのですが、続けて書く for のループ変数名は異なるものを使う必要があります。このデメリットは大きく、後に撤回され、今では C99 と同様になっています。

6.2 ループを抜ける・次のループへ切り上げる

ループを制御する文が2つあります。

break 文 ループを抜け出します。

continue 文 ループ中の処理を途中で切り上げて、次のループにとりかかります。

どちらも **for** や **while** のいずれのループでも使えます。多重ループの場合は、まだ抜けていない一番内側のループに作用します。

```
for (int i=0; i<1000; i++) { // 結果的に1000には意味がない...
    if (i <= 2) continue; // i=0,1,2 でループの先頭に戻る
    if (i == 5) break; // i=5 でループを抜け出す
    printf("%d\n", i); // ここに到達するのは i=3,4 のみ
}
```

break も **continue** も便利そうな機能に思えますが、ブロック中の実行の流れをジャンプさせるため、動作がわかりにくくなるという弊害もあります。そのため、多用すべきでないというのが主流の考え方です。使用は、例外的な事象に、最小限に留めておくのがよいでしょう。

6.2.1 無限ループ

ループ処理は、ちょっとしたミスで終了しなくなることがありますが、そうではなくて、わざと終了しないループ=無限ループ (infinite loop) を作ってみましょう。**for** でも **while** でも実現できます。

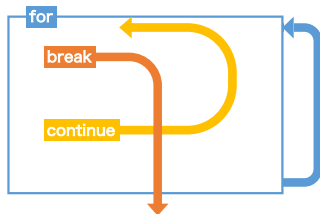
```
for (;;) {
    ...;
}
```

```
while (1) {
    ...;
}
```

```
while (TRUE) {
    ...;
}
```

for 条件式は省略できるので、省略するだけで無限ループになるのですが、括弧内のほかの式も不要になるので、3つとも空欄にします。

while 何かしらの条件式を書く必要があるなので、常に成立する比較式として $0==0$ を書くこともあります。もっとも、この比較式の演算結果は **int** 型の **1** になるので、最初から **1** (あるいはマクロで定義した **TRUE**) と書くことが多いです。



```
int i = 0;
while (1) {
    if (i == 10) break;
    ...; // 10回ループ
    i++;
}
```

無限ループと言っても、break で中断できますから、左のようになりますと 10 回ループを実現できます。ただし、メリットは感じられません。

無限ループが役立つのは、ループの条件判定が単純には書けず、例えば変数へ代入してからようやく判断できる、というような場合です。典型例は、外部からの入力を受け取るときです。詳しくは 6.4.3 項で紹介します。

6.2.2 continue の活用例

ここではインデントの深さを節約する continue 文の使い方を紹介しましょう。ループ処理の範囲は、視覚的にとらえられるようにブロック全体をインデントしますが、そのループ処理の中に条件文などが入れ子になってくると、どんどんとインデントが深くなって読みにくくなります。

例えば左下の例のように、条件 A が成り立ったときに処理 X を行い、それ以外に行うべき処理がないとしましょう。処理 X が長く、この中でも分岐があったりするとますます読みにくくなります。

そこで右下のように、条件 A が成り立たなかった場合に continue して、それ以外の場合に処理 X を行う、と書き換えてみましょう。else 節を書く必要がないので、処理 X のインデントを 1 つ浅くできます。

元々の if の後に処理すべきものがまったくないことが大前提ですが、このように continue で処理を飛ばしたい場面は、現実によく現れます。

```
while (...) {
    if (条件A) {
        ←→ 処理X; // インデント深い
    }
    // ここに処理がないとする
}
```

```
while (...) {
    if (!(条件A)) continue;
    ←→ 処理X; // インデント浅い
}
```

6.3 繰り返し処理にまつわる話題

6.3.1 ループ処理が空

ソースコード 5.2 を for で実現しなおしてみます。ループ内での処理がなくなりましたが、ブロックをあらわす `{ }` (あるいはセミコロン) は残す必要があります。そして、書き忘れたくないことを示すのにコメントを書いておきましょう。

ソースコード 6.1 x を越える最小の自乗数 (for 版)

```
1 int min_square(int x) {
2     int k; // スコープを広げるためにここで定義する必要がある
3     for (k=1; k*k<=x; k++) {
4         // 何もしない
5     }
6     return k*k;
7 }
```

for ループの後にもループ変数 `k` を使いたいの、スコープを広げるために、for ループの前に `k` を定義しています。

6.3.2 少なくとも 1 回は実行するループ

for も while も、ループの条件式を最初から満たさなければ、ループ処理をまったく行わないことになります。しかし状況によっては、少なくとも 1 回はループ処理を実行して、ループを続けるかどうかは後から考えたい、ということもあります。その場合に適しているのが do-while 文です。ループを続けるかどうかを、ループ処理の最後で判断します。

```
do {
    処理;
} while (条件式);
```

現実のプログラムに現れるループの内訳でいえば、for が圧倒的に多数、while がその次、do-while はたまたま目にする、といった具合です。(本書でも 1 度だけ、6.4.3 項で利用します。) でも「少なくとも 1 回は実行する」という場面では do-while のことを思い出してあげてください。

6.3.3 逆順ループ

10, 9, 8, ..., 1 と、ループ変数の値の減っていく、逆順ループも、for と while のどちらでも実現することはできます。

```
for (int i=10; i>0; i--) {
    printf("%d ", i);
}
```

```
int i = 10;
while (i > 0) {
    printf("%d ", i);
    i--;
}
```

しかし、条件に = をつけるかどうかはわかりにくいので、普段どおりのループにしておいて、表示する値を 10-i のように加工するほうがわかりやすいことも多々あります。

6.3.4 break したかどうかを後から判定する

for ループを、break で終了したのか、それともループを最後まで回りきったのか、ループの直後に知りたい場合があります。もちろん論理型変数を作って記憶させればわかることです。多用途の手段ではありませんが、ループ変数の動きをよく理解していれば、ループ後の値を検査することでも区別がつかます。ただし、変数定義を for の括弧内で行うと、ループのブロックの外でその変数が使えないことに注意してください。

```
int i; // スコープを広げるためにここで定義する必要がある
for (i=0; i<10; i++) {
    if (...) break;
}
if (i < 10) printf("breakしました\n");
```

この例では、ループを回りきったのならループ条件の $i < 10$ を満たさなくなっているの、 i は 10 になっていることでしょう。逆に break していればループ条件をまだ満たしているの、ループの外でもう一度ループ条件と同じ検査をすると break したとわかります。

コラム：終わる（と思われる）繰り返し

コンピュータができて70年以上、計算理論という分野が出てきて100年以上が経っていますが、不思議なことに未だに未解決の問題がたくさんあります。

右の関数は、引数として与えられた正の整数から始めて、偶数なら半分に、奇数なら3倍して1を加える、という計算を繰り返します。1になると、 $1 \rightarrow 4 \rightarrow 2 \rightarrow 1$ を繰り返すので、1で終了とします。

さて、この簡単に書けてしまう関数はどんな正の整数に対しても終了するのでしょうか？これはコラッツ予想^aと呼ばれる問題で、実は未だに決着がついていません。

様々な値で試した結果としては、万、億、兆の範囲ではすべて終了することが確認されています。予想としてはどんな値でも終了すると考えられていますが、証明はされていませんし、反例も見つかっていません。単純に書かれたものは、簡単に見えてもそうとは限らないという一つの例でしょう。

折角なので、この関数を for 文で次々呼び出して50くらいまでの結果を見てみるのも良いでしょう。10000くらいまで試すと表示が多すぎて見えないと思いますが、まだまだ終了することが分かります。値によっては大きくなった結果、オーバーフローする場合がありますので、注意しましょう。例えば int 型が32ビットの場合でも113383から始まる系列が表示できないようです。開始時点の数値からするとずいぶん大きくなるようですね。これもどのような動きをしているか出力してみると面白いでしょう。

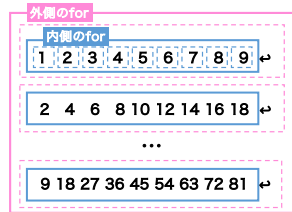
ソースコード 6.2 コラッツ予想

```

1 void print_collaz(int n) {
2     while (n > 1) {
3         printf("%d -> ", n);
4         if (n % 2 == 0) {
5             n = n / 2;
6         } else {
7             n = 3 * n + 1;
8         }
9     }
10    printf("1 -> ... \n");
11 }

```

^a 他にも様々な名前があります。日本人の名前を使う場合もあり、^{かくたに}角谷の問題、米田の予想、とも呼びます。



6.4 より高度なループ処理の実例

6.4.1 九九の表

次のような九九の表を表示してみましょう。2重ループで実現できそうです。

ソースコード 6.3 の実行結果

```
1  2  3  4  5  6  7  8  9
2  4  6  8 10 12 14 16 18
3  6  9 12 15 18 21 24 27
....
9 18 27 36 45 54 63 72 81
```

ソースコード 6.3 九九の表

```
1 #include <stdio.h>
2
3 int main(void) {
4     for (int i=1; i<=9; i++) { // 縦に9行
5         for (int j=1; j<=9; j++) { // 横に9列
6             printf("%2d ", i*j); // 積の表示 (81回)
7         }
8         printf("\n"); // 改行して左端に戻る (9回)
9     }
10    return 0;
11 }
```

1回 9回 81回

ここで注意したいのが、ソースコード 6.3 の 8 行目の改行の動作です。printf で表示した文字は、通常の文字なら左から右に進んでつながっていきます。改行文字 ("\n") を表示すると、表示位置が次の行（つまり下方）に進んで左端に戻ります。画面には何も表示されず、表示位置のみが変化します。これを 1 行の表示の最後に行っています。

この 8 行目の printf は、内側のループには含まれず、外側のループに入っているため、9 回だけ実行されることを理解しておきましょう。内側のループは、ループ自体が 9 回実行され、6 行目の printf はその内側に入っているため、のべ 81 回実行されます。

6.4.2 コンマで区切って列挙

1 から n までの数字を「1, 2, 3, ..., n 」のように、コンマで区切って表示してみましょう。ちょっと考えてみてください、ループで簡単に実現できるでしょうか。数字は n 個、コンマは $(n-1)$ 個と、**個数が一致しない**ので、何かしら例外処理が必要になります。

- (A) 先頭の 1 を無条件に表示して、残りを",%d"で表示する。
- (B) $(n-1)$ までを"%d,"で表示して、最後の n を無条件に表示する。
- (C) 数値とコンマを常に別々に表示する。数値を"%d"で表示してから、最後以外で", "を表示する、を繰り返す。

ソースコード 6.4 1 から n までの数をコンマ区切りで表示

```

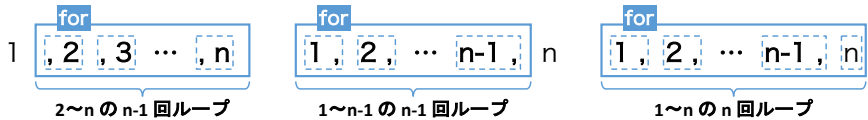
1 void join1(int n) { // (A)
2     printf("%d", 1); // 先頭を特別扱い
3     for (int i=2; i<=n; i++) {
4         printf(",%d", i);
5     }
6     printf("\n");
7 }
8
9 void join2(int n) { // (B)
10    for (int i=1; i<=n-1; i++) {
11        printf("%d,", i);
12    }
13    printf("%d\n", n); // 末尾を特別扱い
14 }
15
16 void join3(int n) { // (C)
17    for (int i=1; i<=n; i++) {
18        printf("%d", i);
19        if (i != n) { printf(","); } // ループの最後以外で, を表示
20    }
21    printf("\n");
22 }

```

このいずれの方法でも、ソースコード 6.4 の join1(), join2(), join3() のように実現可能です。しかしここで注意したいのは、例外的に行っている動作が、極端な条件下でどうなるか、です。

実は n の値の範囲をまだ明確にはしてませんが、例えば $n=0$ にしたときの動作はどうか、なればよいでしょうか。「想定外なので、どんな動きをしても知りません」という作り手の立場もあるでしょう。使い手の立場で言えば「0 個の数字が表示される = 数字は表示されない」と思ってるかもしれません。

実際、join1(0) と join2(0) は数字が表示されます。表示されないように修正するのを練習問題としておきます。



話は変わりますが、ソースコード 6.4 の 10 行目は良いループ (☞ 5.5 節) でしょうか。

- `for (int i=1; i<=n-1; i++)` は良いループです。1 から始まる (n-1) 回ループです。i=n に例外がありそうなことが、ループだけからも読み取れます。
- `for (int i=1; i<n; i++)` でも動作は同じですが、ここには不適切です。0 オリジンで i=0 に例外がありそうにも見えます。

6.4.3 キーボードから正しい値を受け取るまで繰り返す

キーボードから入力された数値が、想定している範囲に入らなかったとします。正しい値を入力してもらうよう、再入力を促しましょう。

13.8.2 項で紹介するような手段で、`input_int()` がキーボードから `int` の数値を受け取るものとします。

少なくとも 1 回はキーボードから値を受け取りますので、`do-while` (☞ 6.3.2 項) の出番です。

```
int a;
do {
    printf("0以上10未満の数を入力してください => ");
    a = input_int();
} while (!(0 <= a && a < 10));
```

確かに、以下の実行例のように、範囲外の値 (100 と -1) を入力すると、また入力の場面に返るのですが、利用者からすると、同じメッセージが表示されるだけなので、再入力を促されてるのかどうかわかりにくいです。

実行例: 「100」と「-1」を入力した場合

```
0以上10未満の数を入力してください => 100
0以上10未満の数を入力してください => -1
0以上10未満の数を入力してください =>
```

そこで、入力された数値が範囲外であれば、そのことを表示してみます。

```
int a;
do {
    printf("0以上10未満の数を入力してください => ");
    a = input_int();
    if (a < 0) {
        printf("%dは0未満です。再入力してください。 \n", a);
    }
    if (a >= 10) {
        printf("%dは10以上です。再入力してください。 \n", a);
    }
} while (!(0 <= a && a < 10));
```

実行例: 「100」と「-1」を入力した場合

```
0以上10未満の数を入力してください => 100
100は10以上です。再入力してください。
0以上10未満の数を入力してください => -1
-1は0未満です。再入力してください。
0以上10未満の数を入力してください =>
```

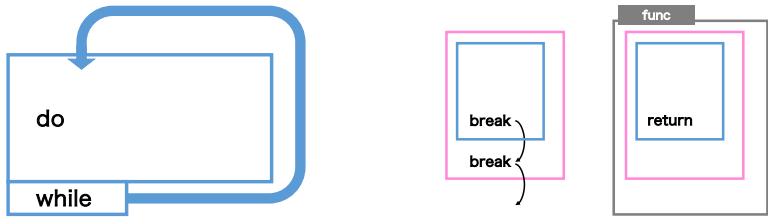
親切になったのですが、このプログラムには欠点があります。上限値の10を変更したくなったら、注意深く2カ所の条件式(に加えてメッセージの内容)を書き直す必要があります。つまりメッセージを表示する条件と、ループを継続する条件が別々になっています。これは崖っぷちでバランスをとってるようなもので、バグの温床になります。

それでは、メッセージ表示とループ継続が、必ずセットになる条件分岐にしましょう。こうなると、もうdo-whileは使えず、無限ループの出番です。以下のようにすると、どちらのifが成立しても、メッセージを表示した上でbreakしません。

```
int a;
for (;;) {
    printf("0以上10未満の数を入力してください => ");
    a = input_int();
    if (a < 0) {
        printf("%dは0未満です。再入力してください。 \n", a);
    } else if (a >= 10) {
        printf("%dは10以上です。再入力してください。 \n", a);
    } else {
        break; // 正しく入力された
    }
}
```

breakはエラーなどの例外的なものに対して使うことが多いので、正しい入力の際にbreakで中断するのは気持ちが悪いのですが、ここではいたしかたありません。

このように、人間の不規則な入力に対処するためのエラー処理は複雑になりがちです。人間に親切にすればするほど、長いプログラムになっていきます。



6.4.4 2重ループの中断

`break` によるループの中断は、直近のループにだけ作用します*3。そのため、2重ループを抜け出すには工夫が必要です。

まずは論理型の変数（フラグ）を使う方法です。下の例では、内側のループを `break` で抜ける前に、変数 `finished` を真にします。抜けた直後に `finished` を検査して、真であればもう一度 `break` します。これで外側のループから抜け出せます。

```
int finished = FALSE;
for (int i=0; i<10; i++) {
    for (int j=0; j<10; j++) {
        if (...) {
            finished = TRUE;
            break; // 内側ループ用
        }
    }
    if (finished) break; // 外側ループ用
}
```

もうひとつの方法は、2重ループを関数に入れておいて、`return` で関数ごと抜け出すというものです。

```
void func(void) {
    for (int i=0; i<10; i++) {
        for (int j=0; j<10; j++) {
            if (...) return; // ループはもちろん、関数ごと抜け出す
        }
    }
}
```

関数に分離すればインデントを浅くとどめられるという効果もあるので、実用的な手法だと思います。

*3 言語によっては、どのループまで抜け出すかを指定できるものもあります。

6.4.5 素数判定

与えられた2以上の整数 n が素数かどうかを判定することを考えてみます。数学の定義によると、1と n 自身を除いて、約数のないものが素数です。これから作る関数は、素数ならば論理型の TRUE、そうでなければ FALSE を返すことにします。例外処理を省くために、与えられる n は2以上であることを前提にします。

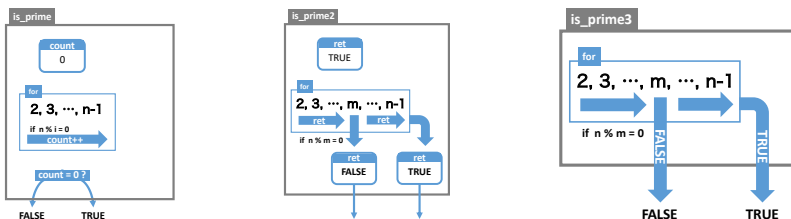
ソースコード 6.5 を見ていきましょう。3つの関数がありますが、下線部の処理は共通ですので、下線部以外の違いをよく理解してください。

ソースコード 6.5 素数判定

```

1 #define FALSE 0
2 #define TRUE 1
3
4 /* nが素数なら TRUE を返す */
5 int is_prime(int n) { // (A)
6     int count = 0; // 約数の数
7     for (int i=2; i<=n-1; i++) { // 1とnを除いてループする
8         if (n % i == 0) { count++; } // 約数が見つかったら1増やす
9     }
10    if (count == 0) return TRUE; // 約数が0個なら素数
11    else return FALSE;
12 }
13
14 int is_prime2(int n) { // (B)
15     int ret = TRUE; // 素数であることを前提にスタート
16     for (int i=2; i<=n-1; i++) {
17         if (n % i == 0) {
18             ret = FALSE; // 約数が見つかったら素数ではない
19             break; // ループを続ける必要もない
20         }
21     }
22     return ret;
23 }
24
25 int is_prime3(int n) { // (C)
26     for (int i=2; i<=n-1; i++) {
27         if (n % i == 0) {
28             return FALSE; // 約数が見つかったら偽を返す
29         }
30     }
31     return TRUE; // 最後まで約数が見つからなければ素数
32 }

```



コラム：ある?ない?

約数に限らず、「ない」と判定するのは、難しいことです。逆に「ある」と判断するのは簡単です。それを見つければ動かめ証拠になります。

ループ処理中の if で、見つけたいものを探しているとします。if が成り立てば（見つければ）「ある」と即断できます。逆に、ループ中に 1 回ぐらい if が成り立たなかったからといって、すぐに「ない」とは決められません。「ない」と判断できるのは、ループが終わってからです。そしてループ中に、いくつ見つかったのか、個数を数えておくとうまく判断できます。個数の 0 は「ない」ことを示します。

- (A) まず `is_prime` 関数では、定義どおり 2 から $(n-1)$ のループで約数がいくつあるか数えてみます (7-9 行目)。その結果、0 個であれば素数だとわかるので `TRUE` を返します (10 行目)。そうでなければ `FALSE` を返します (11 行目)。
- (B) しかしよく考えると、約数が 1 つでも見つかる、もっとたくさん見つかって、素数でない (`FALSE` を返す) ことには変わりありません。つまり約数の個数は重要ではないので、見つかったかどうかを論理型で覚えておけば十分です。そこで `is_prime2` 関数では、変数 `ret` を用意して、ひとまず `TRUE` にします (15 行目)。約数が見つかったと `ret` を `FALSE` にします (18 行目)。こうすることで、`ret` 変数が関数の戻り値としてそのまま使えます (22 行目)。
- ところで、ループを続けて約数をいくつも見つけると、何度も `ret` に `FALSE` を代入することになりますが、一度代入すれば十分ですので、そこでループを中断しても、結果は変わりません (19 行目)。こうすることで計算時間を節約できます。
- (C) ここまで、ループ中の計算結果を変数に反映してきましたが、関数の中での完結した処理なので、変数を使わなくても呼び出し元に影響はありません。そこで `is_prime3` 関数では、素数ではないという結果が出れば、その場で `FALSE` を返して (28 行目)、ループはもちろん関数ごと抜け出すことにしました。結果を保存する変数が不要になり、見通しよくなります。ループの最後に到達すると、素数だと判明するので `TRUE` を返します (31 行目)。

6.5 練習問題

1. [2重ループ、回数の変化するループ]

2重ループを用いて「*」の文字を並べて、縦横 n 文字の長方形を表示せよ。また同様に、縦横 n 文字の直角三角形を表示せよ。右は $n = 5$ の出力例である。

```
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
```

2. [3重ループ or 1重ループ]

153 は特別な数である。各桁の数の3乗の和 $1^3 + 5^3 + 3^3 = 1 + 125 + 27 = 153$ が、元の数に一致する。このような3桁の整数を、総当たりですべて求めて表示せよ。

3. [例外処理]

ソースコード 6.4 の `join3(0)` では数字が表示されない。しかし `join1(0)` と `join2(0)` では、1個の数字が表示される。数字が表示されないように修正せよ。

4. [複雑なループ]

ソースコード 5.6 の `cbrt_eps()` 関数の `while` ループに、ループ回数の上限を設けよ。`break` を使っても構わないし、ループの条件を増やしてもよい。`for` ループに書き換えてもよい。

5. [逆ループ or 計算]

ソースコード 6.3 の九九の表を、上下反転して表示せよ。

6. [例外処理]

九九の表の数字を、スペース区切りではなくコンマ区切りで表示せよ。行末の無駄なコンマは除去せよ。

7. [2重ループ or 1重ループ]

1 から 30 までの数字を、7 個おきに改行をはさんで、月めくりカレンダーのように表示せよ。また、1 の開始位置を 7 ヶ所のどこにでも自由にずらせるようにせよ。

8. [数列の和]

数列 $\{a_k\}$ を以下で定義する。 a_k と a_{k-1} の関係に着目して a_0, a_1, \dots, a_{10} を求めよ。

$$a_k = \frac{1}{k!} \quad (k = 0, 1, 2, \dots)$$

そして、数列 $\{a_k\}$ の和 S_n を以下で定義する。 S_0, S_1, \dots, S_{10} を求めて表示せよ。

$$S_n = \sum_{k=0}^n a_k = \sum_{k=0}^n \frac{1}{k!} = \frac{1}{0!} + \frac{1}{1!} + \frac{1}{2!} + \dots + \frac{1}{n!} \quad (n = 0, 1, 2, \dots)$$

- $0! = 1$ である。 $k = 0$ で例外処理が必要かもしれない。
- S_∞ はネイピア数 $e (= 2.71828 \dots)$ に収束することが知られている。

9. [キー入力に対するループ] [📖 12.1.1 項](#)

第7章

関数 (2)

本書では、一般的な入門書とは異なり、関数の作り方を早い段階で紹介しました。そして、手間のかかる約束事の紹介を後回しにしました。これは、言語にかかわらず通用する、大きく複雑な作業を、小さな関数に分割して単純化して解決するという体験を、プログラミングの早い段階で経験してもらいたかったからです。

さてここでは、後回しにしていた約束事を紹介します。C言語に特有のことも多いため、難易度は高く感じるかもしれませんが、少しずつ身につけていけば大丈夫ですので、安心してください。

キーワード

- プロトタイプ宣言
- 暗黙的な宣言
- 標準ライブラリ関数のヘッダ
- スコープ・寿命
- グローバル・ローカル・static
- スネークケース・キャメルケース

7.1 プロトタイプ宣言

これまでのプログラムでは、ソースコード上での位置でいうと、呼び出される関数を先に記述して、後から呼び出していました。そして、最初に実行される main 関数を、ソースコードの最後に書いていました。この順序制約は、関数の間の呼び出し関係が明らかならば気にするほどでもないでしょうが、長いプログラムになってきたり、2つの関数がお互いに呼び出しあっているなど、逃れたいくなる場面もあります。

ソースコード 7.1 プロトタイプ宣言なし

```

1 #include <stdio.h>
2
3 // プロトタイプ宣言なし
4
5 int main(void) {
6     printf("正方形の面積は%fです。 \n", square(1.0));
7     return 0;
8 }
9
10 double square(double x) { return x * x; } // 関数定義

```

例を見てみましょう。ソースコード 7.1 は順序制約を破っているので、コンパイル時に次のような警告とエラーが出ます。

ソースコード 7.1 のコンパイル出力 (gcc の場合)

```

square.c: 関数 ‘main’ 内: 6 行目 5 文字目の意味
square.c:6:5: 警告: 関数 ‘square’ の暗黙的な宣言です
printf("正方形の面積は%fです。 \n", square(1.0));
    ^
    5 文字目の位置の目印
] 1つの警告メッセージ

square.c: トップレベル: 10 行目 8 文字目の意味
square.c:10:8: エラー: ‘square’ と型が競合しています
double square(double x) { return x * x; }
    ^
    8 文字目の位置の目印
] 1つのエラーメッセージ

square.c:6:50: 備考: 前の ‘square’ の暗黙的な宣言はここです
printf("正方形の面積は%fです。 \n", square(1.0));
    ^
    競合相手の場所

```

このメッセージは、意味がわかりにくいのですが、6行目で square() を呼び出すまでに、この関数の戻り値や引数の型が判明していない、ということの間接的に表現しています。(メッセージの意味は 7.1.1 項で読み解きます。)

関数の、引数などを含めた型のことを関数プロトタイプ (function prototype) といいます。関数呼び出しの段階で、引数の数が合致してるかなどの検査をするためには、関数プロトタイプが必要になります。コンパイラは、ソースコードを上から順番に解釈してゆき

ますから、後から記述してあることを知らなくてもよいように、ソースコード上の記述順序に制約^{*1}を設けてあるのです。

この順序制約を実質的に取り扱う方法があります。後から出てくる関数の「予告」を事前に書けばよいのです。予告のことを関数の**プロトタイプ宣言** (prototype declaration) といいます。ソースコード 7.2 では、3 行目を書き加えたことで、エラーや警告が解消されました。3 行目のプロトタイプ宣言は 10 行目の関数定義とほとんど同じ、違いは関数ブロックをセミコロンで置き換えてあることだけです。

ソースコード 7.2 プロトタイプ宣言あり

```
1 #include <stdio.h>
2
3 double square(double x); // プロトタイプ宣言
4
5 int main(void) {
6     printf("正方形の面積は%fです。 \n", square(1.0));
7     return 0;
8 }
9
10 double square(double x) { return x * x; } // 関数定義
```

自分のプログラムでプロトタイプ宣言を書くときは、手でタイプしなおすと間違えるので、完成している関数の先頭部分をテキストエディタでコピーします。そして、関数ブロックを消して、代わりにセミコロンを書きます。プロトタイプ宣言を書く場所は、関数呼び出しよりも前であればどこでもよいのですが、中途半端なことはせず、`#include` のすぐ下の行に書くのが習慣です。

なお、関数定義（本体）の現れた後は、正しく関数呼び出しができますから、関数定義もプロトタイプ宣言の役目を含んでいることがわかります。

コラム：プロトタイプ宣言は無駄？

プロトタイプ宣言を「同じことを 2 度も記述して無駄」とか「自動生成できてしかるべき」と思う人は、プログラミングの才能があります。現実の作業として、プログラムを作っているうちに引数が足りないことに気づいて後から追加する、などという場面は頻繁にあって、そのたびにプロトタイプ宣言まで修正する（コピーしなおす）のは煩雑かつミスが入りやすいです。ですから、関数の記述順序を調節して、**プロトタイプ宣言を書かずにすませる**という流儀もよく見かけます。

^{*1} この制約は、C 言語が登場した頃のコンピュータの能力からくるもので、今の進歩したコンピュータの能力や高性能なコンパイラ技術から考えると、コンパイラを甘やかせ過ぎているようにも感じられます。実際、Java や C# のような新しい言語には記述順序に制約はありません。

ただし、分割コンパイル (1つのプログラムを複数のソースコードで構成する) を行うときには、ファイルを越えた呼び出しを許す (つまり公開する) 関数だけを選び出して、あるファイル^aにプロトタイプ宣言を書き並べる必要があります。この選び出す作業の存在が、自動生成を難しくしています。

^a あるファイル (=共有するヘッダファイル) で関数のスコープを制御するという仕組みは、単純かつ効果的ではありますが、時代を感じずにはられません。ちなみに Java 言語では、プロトタイプ宣言もヘッダファイルもなく、メソッド (≈関数) に `public` という修飾子をつけることで、クラス (≈ファイル) を越えた呼び出しを許したことになります。

7.1.1 暗黙的な宣言

ここで紹介するのは C99 で廃止されたはずの機能ですが、一部のコンパイラになぜかまだ残っています。また、廃止されたコンパイラでのエラーメッセージを理解するのにも役立つ知識です。

プロトタイプ宣言を忘れると何が起るのでしょうか。ソースコード 7.1 のコンパイル出力をもう一度見てみてみましょう。関数を呼び出そうとした時点ではまだエラーにならず、**暗黙的な宣言** (implicit declaration) を行ったという警告^{*2}になっています (6行目)。これまでに明示的なプロトタイプ宣言がなかったので、事前に決められたプロトタイプで代用するというのがこの機能です。事前に決められたプロトタイプとは `int func();` という、関数は `int` 型を返し、引数は (ないわけではなく) **型検査をしない恐ろしいもの**^{*3}です。

構文解析が進んで、関数本体のプロトタイプが、初めの暗黙的な宣言と矛盾したので、めでたくエラーとして検出されました (10行目)。この長いメッセージで、矛盾する宣言がどこにあったのかまで教えてくれています。

しかし、運悪くプロトタイプが矛盾しなければ、この段階は警告になりませんし、全体を通じてエラーが起りません。運悪くといっても、関数が `int` 型でさえあれば**引数は検査の対象外**ですから、簡単に起ることです。ですから、最初の警告を見逃さないようにすることが重要です。

^{*2} 暗黙的な宣言は、C 言語の初期の規格 (K&R) で書かれたソースコードをエラーにしないために、言語規格の 2 世代目 (C89) で考案された巧妙な仕組みです。このため「プロトタイプ宣言がない」との直接的なメッセージになりません。3 世代目の C99 では廃止されたので、正しく準拠すればエラーになるのですが、それでも「暗黙的な宣言は無効」のような、回りくどいメッセージになることもあります。

^{*3} プロトタイプ宣言の引数は、() と空にすると型検査の対象外になります。引数がないのであれば、いつもの `main` 関数のように、(void) と書いて示す必要があります。

7.1.2 標準ライブラリ関数のプロトタイプ宣言

C 言語で用意されている標準ライブラリ関数であっても、呼び出す前にはプロトタイプ宣言が必要です。このようなプロトタイプ宣言は、システムの用意するヘッダファイル (header file) に記述されているので、`#include` 命令^{*4}で読み込みます。`printf()` のプロトタイプ宣言は `<stdio.h>` というファイルに書かれているので、これまでのプログラムの先頭には「`#include <stdio.h>`」を書いてきました。数学関数を使う場面では、「`#include <math.h>`」も追加しましたが、このような仕組みになっていたのです。

コラム：標準ライブラリ関数の調べ方

C 言語には言語規格書なるものがあって、標準ライブラリ関数のプロトタイプや、関数がどのような動作をするのかも決められています。今やインターネットで検索すれば、これに準じるような文書が簡単に見つかります。

インターネットにつながってなくても、Unix 系 (Cygwin を含む) なら `man` というコマンドがあって、`man 3 sin` を実行すると、`sin` 関数の説明が表示されます。(閲覧中の操作は、スペースキーで次のページ、`q` で終了です。)

`man` コマンドには各種の説明文 (マニュアル) が集約されていて、セクション番号によって分類されています。“3”というのは C 言語のライブラリ関数を指し、Unix コマンドなら “1” にします。名前に重複がなければ省略しても構いません。詳細は `man man` で調べましょう。

7.1.3 プロトタイプ宣言における省略と重複

コンパイラが関数の型検査を行うときに、引数の「型」が何であるかは重要ですが、引数の「名前」は関係ありません。ですから、プロトタイプ宣言では、引数の名前は何であってもエラーになりませんし、省略することすら可能です。しかも、矛盾さえなければ、同じ関数のプロトタイプ宣言を何度行ってもエラーになりません^{*5}。以下は文法上、正しいプログラムです。

```
double square(double a); // プロトタイプ宣言 (関数定義と引数名が異なる)
double square(double);  // プロトタイプ宣言 (引数名を省略)
double square(double x); // プロトタイプ宣言 (3 度めでも OK)
double square(double x) { return x * x; } // 関数定義
```

^{*4} # で始まるので、プリプロセッサ命令です。コンパイルの最初期段階で処理されます。

^{*5} 関数定義 (本体) もプロトタイプ宣言の機能を含んでいますので、言語機能上、重複は禁止できません。

しかし、このように引数を省略したり、何度も宣言する積極的な理由は思いつきません。むしろ引数の名前からは役割が連想されて、関数を呼び出すプログラムの役に立ちそうです。したがって、これらのことは文法的な知識にとどめておいて、プロトタイプ宣言は単純に関数本体からコピーして、一度だけ記述すればよいでしょう。

コラム：テキストエディタの行選択とプロトタイプ宣言

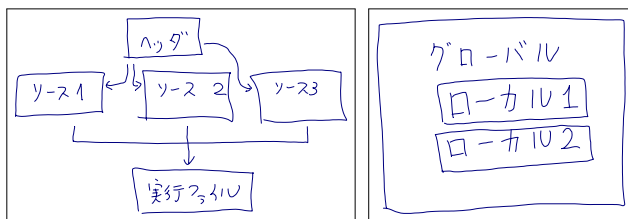
プロトタイプ宣言を書くのには、テキストエディタで関数定義の最初の行をコピーします。この作業に、マウス操作の得意(?)な初学者が、1文字単位で時間をかけて選択しているのをよく見かけますが、見ているだけでまだるっこしいです。

多くのエディタは、たとえマウスで操作しても、**行番号の数字の上をクリック**するだけで、1行(行頭から行末まで)が選択されます。トリプルクリック(マウスのダブルクリックの、もう一度多くクリックする)でも簡単に選択できるでしょう。さらにキーボードなら2~3操作(例えば **Home** **Shift**+**↓**)で同様のことができますから、コピー&ペーストは、間違えることなくすぐに終わります。普段からマウスでちまちま1文字単位で選択していると、関数の引数を修正するときに、面倒くさがってコピーし直さず、プロトタイプ宣言まで手作業で編集して、しかも間違えてコンパイルエラーで右往左往するという**絵に描いたような苦行**が待っています。書かずにすませることもできるプロトタイプ宣言で、わざわざ険しい道に行くわけです。

近頃は、選択した**領域の移動**(カット&ペースト)ができずに、**コピー&ペースト&コピー元削除**、という操作をする人が増えています。もちろん削除するのにコピー元をマウスで選択しなおすので、1文字単位で範囲を間違えて、あっという間にコンパイルエラーです。早く「カット」を覚えましょう。

テキストエディタにもよるのですが、行選択すれば、**Tab**と**Shift**+**Tab**でインデントの深さを**選択領域**まるごと調節できることがあります。ぜひ使いこなしたい機能です。

まずは「カット」**Ctrl**+**X**、「コピー」**Ctrl**+**C**、「ペースト」**Ctrl**+**V**、そして「上書き保存」**Ctrl**+**S**、「アンドゥ」**Ctrl**+**Z**、「リドゥ」**Ctrl**+**Y**をマスターしましょう。これは最低限です。そしてプロトタイプ宣言を書くのは、行選択が素早くできるようになってからです。



7.2 変数のスコープ・変数の寿命

これまで、変数の定義は関数のブロックの中を書いてきました。これを関数ブロックの外（トップレベル）で行うとどうなるでしょうか。実はエラーにならずに、ちゃんと使えます。

この変数のスコープ（有効範囲）はどうなるでしょうか。さらに考えると、今まで気にならなかった、変数の誕生や消滅の瞬間がいつなのか問題になってきます。順番に考えていきたいのですが、一つ頭の隅に覚えておいて欲しいことがあります。

本書のプログラムは、実行プログラムを1つのソースコードから生成しています。main関数のある.cファイルが1つです。しかし一般のC言語プログラムは複数のソースコードで構成され、ファイルごとに別々にコンパイルする分割コンパイルが行われているということです。

7.2.1 ローカル変数・グローバル変数

関数の中で定義された変数のスコープは、3.5節で述べたように、関数のブロックの中だけでした。この変数は、スコープの狭さから、その関数の局所変数あるいはローカル変数 (local variable) と呼ばれます。もちろん、仮引数もローカル変数です。そして関数ブロックの入れ子になった内側のブロックで定義されていれば、スコープはそのブロックのみと、より狭くなるのでした。

関数の外で定義された変数のスコープは、プログラム全体に広がります。手続き^{*6}さえ踏めば、分割コンパイルされる別のファイルからでも参照できます。この変数は、スコープの広さから、大域変数あるいはグローバル変数 (global variable) と呼ばれ、どの関数に属するかという概念がなくなります。

^{*6} 本書では扱いません。関数にプロトタイプ宣言があるように、変数にも宣言のみ（実体を伴わない）があります。

表 7.1 スコープの種類

変数の種類	変数を定義する場所	実現されるスコープ	スコープの名称
グローバル	関数外	プログラム全体	プログラムスコープ
static グローバル	(トップレベル)	1つのソースファイル	ファイルスコープ
ローカル,	関数内, 仮引数	1つの関数	関数スコープ
static ローカル	関数内のブロック内	関数内のブロック内	ブロックスコープ
	関数内のブロックのさらに内側の...	関数内のブロックのさらに内側の...	

7.2.2 時間的な有効期間＝寿命と static

ここでは、プログラムの実行する部分が進むのに応じて、変数がどのタイミングで必要になり、どのタイミングで不要になるのかを見ていきます。つまり、変数の誕生から消滅までの、**寿命** (lifetime) を考えてみます。

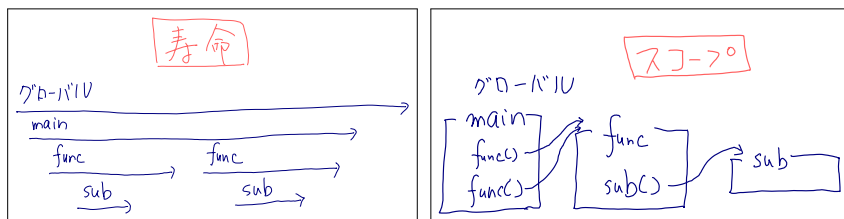
ローカル ローカル変数は、定義されているブロックを実行するときに誕生します。(記憶するための領域が確保されます。)そして、そのブロックを抜け出すときに、不要になるので消滅します。(領域が開放されます。)ブロックをもう一度実行しなせば、ローカル変数はまた新たに作り直されます。今まで気にすることはなかったかもしれませんが、このような動きになっていました。そして初期化は自動では行われず、プログラムで代入するまで、どんな値になっているのか保証がありません。

グローバル グローバル変数は、いつどの関数から使われるかわかりませんから、消滅するわけにはいきません。プログラム開始と同時に誕生して、プログラム終了まで存続します。ローカル変数とは異なり初期値は0です*7が、0であっても明示的に初期化(定義と同時に代入)するのがよいでしょう。

static ローカル ローカル変数でも、計算結果を残しておくなど、次の関数呼び出しまで値を保ちたいこともあるでしょう。そんなときは、ローカル変数の定義に **static** (静的) というキーワードをつけて「`static int a = 5;`」のように書きます。スコープが狭い特徴はそのまま、グローバル変数相当の寿命を獲得して、プログラム終了まで存続します。初期値もグローバル変数と同じく0です*8し、初期化(定義と同時に代入)もプログラム開始直後の1回だけになります。

*7 ローカル変数と字面が同じで区別がつきにくいからでしょうか、あまり理解されていないようです。

*8 static だと字面が違うためか、よく理解されているようです。



7.2.3 空間的な有効範囲=スコープと static

7.2.1 項に続いて、もう一度スコープ、つまりプログラムの実行している場所とは無関係に、変数の定義のされかたによって、参照を許す範囲がどう変わるかをまとめてみます。

ローカル ローカル変数は、既に述べた通り、ブロックの中だけで有効です。(引数や関数本体のブロックで定義されるものを関数スコープ、さらに内側のブロックならばブロックスコープと区別したり、まとめてローカルスコープと呼ぶこともあります。)

グローバル グローバル変数は、やはり既に述べた通り、スコープがプログラム全体にわたります。(グローバルスコープとかプログラムスコープと呼ぶこともあります。) 有効範囲があまりにも広いので、グローバル変数の使用は最小限にとどめるべしとの戒めとともに、たとえ使うとしても、変数名の重複が起こらないよう、変数名を長くしたり、接頭辞や接尾辞で目立たせるなどの工夫が推奨されます。

static グローバル グローバル変数のスコープを少し狭くする方法があります。ここでも **static** をつけます*9。寿命は長くなりようがありませんが、スコープがそのファイルだけに限定されます。(ファイルスコープと呼びます。) 変数名の重複も、1つのファイルだけ気にすればよくなります。

以上をまとめると表 7.1 のようになります。なお、スコープは変数以外にも考えられます。

関数 関数にも、どこから呼び出せるかというスコープを考えることができ、通常の間数はすべてのファイル、**static** をつけるとそのファイルだけになります。つまりグローバル変数と同じです*10。

マクロ マクロのスコープは、ブロックとは無関係に、そのファイルの終わりまでです。**static** もつけられません。

*9 C 言語は少ないキーワード(予約語)でやりくりしているので、文脈によって意味が変わることがよくあります。**static** も **void** に並んでバリエーションの多いキーワードです。

*10 普通の間数がグローバルだとすると、ローカル関数(関数内の関数)が作れそうに思えるかもしれませんが、C 言語の規格にはないのですが、gcc のように、コンパイラの独自拡張で使える場合もあります。

7.2.4 static の使用例

ソースコード 7.3 では、関数が呼び出されるたびに、1, 2, 3, ... と、呼び出された回数を返す関数を作ろうとしています。4 通りの変数を使ってみました。

- (A) `call_num_local()` では通常のローカル変数を使いました。関数呼び出しが起こるたびに `n` が 0 で初期化されるため、返す値は常に 1 となり、目的の動作をしてくれません。関数の処理が終わった後でも、値を保持する変数が必要になります。
- (B) `call_num_global()` ではグローバル変数を使いました。グローバル変数はスコープが広いので、この変数を使う関数名を接頭辞にして、長い変数名にしてみました。初期化 (0 の代入) が行われるのはプログラム開始直後の 1 回だけで、関数の処理が終わっても値を保持し続けて、望みの動作をしてくれます。ひとまず目標達成です。しかし、変数名をいくら長くしたところで、プログラムの別の場所から値を (意図せず) 変更してしまう危険が付きまといます。
- (C) `call_num_static_local()` ではローカル変数に `static` 属性をつけてみました。変数のスコープはローカル変数と同じで、この関数専用です*¹¹。変数名が短くても安全です。変数の寿命は `static` によってグローバル変数と同等になり、0 の代入もプログラム開始直後の 1 回だけ起こります。望みの動作をしてくれますし、変数名も短くできるので、この方法が一番適しているでしょう。
- (D) `call_num_static_global()` ではグローバル変数に `static` 属性をつけてみました。変数のスコープはこのソースファイルのみと、少し狭くなるので、少し短い変数名にしました。今回は当てはまりませんが、複数の関数からアクセスする必要がある場合には、この方法がよさそうです。

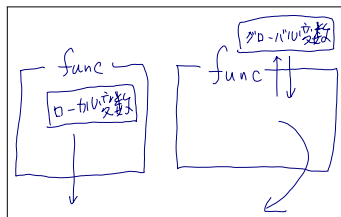
コラム：main 関数のプロトタイプ宣言

コンピュータ言語によっては、初めから用意された関数が特別扱いされていて、「組み込み関数」として何の準備もなく呼び出せることがよくあります。

ところが C 言語では、ライブラリ関数であってもプロトタイプ宣言が必要と、徹底しています。それでは main 関数のプロトタイプ宣言は必要でしょうか。

実は main 関数だけは特別で、引数のパターンが複数あるため、プロトタイプ宣言を書くことができません。 `int main(int argc, char *argv[])` という引数をとるタイプもよく使われます。(☞ 11.4 節)

*¹¹ もっともマルチスレッドになると、これでも占有したことになります。



ソースコード 7.3 呼び出された回数を数える関数

```

1 int call_num_local(void) { // (A) 失敗
2     int n = 0; // ローカル変数 (短い変数名)
3     n++;
4     return n; // 1, 1, 1, ...
5 }
6
7 int call_num_global_counter = 0; // グローバル変数 (長い変数名)
8 int call_num_global(void) { // (B)
9     call_num_global_counter++;
10    return call_num_global_counter; // 1, 2, 3, ...
11 }
12
13 int call_num_static_local(void) { // (C)
14     static int n = 0; // staticなローカル変数 (短い変数名)
15     n++;
16     return n; // 1, 2, 3, ...
17 }
18
19 static int count = 0; // staticなグローバル変数 (少し長い変数名)
20 int call_num_static_global(void) { // (D)
21     count++;
22     return count; // 1, 2, 3, ...
23 }

```

コラム：単体テスト・結合テスト

関数をつつ作るごとに、その関数の単独の動作テストをしましょう。この作業は**単体テスト** (ユニットテスト・unit test) と呼ばれます。関数の引数 (入力データ) を自由に操作できるので、境界値分析 (☞ 70 ページのコラム) も行いやすいです。

単独の動作が正しいとなれば、次はプログラムに組み込んでの**結合テスト** (統合テスト・integration test) です。結合テストは、プログラムが大きくなって、テストそのものにも手間がかかりますし、動作不良があっても原因を探るのが難しくなっています。単体テストをちゃんとやっておかないと、後で苦労します。

7.2.5 変数名や関数名の重複

これまで、2つの変数の名前を同じにして、コンパイルエラーになったことがあると思います。変数名は重複できないことを経験的に理解されたことでしょう。

この制限には、正確には「同一ブロックで」という但し書きがつきます。つまり内側にブロックを作れば、外側のブロックと同じ変数名でもよくなります。参照するときには、まだ実行中のブロックのうち、もっとも内側のものが優先して使用され、それより外側のブロックにある同名の変数は隠蔽され、参照できなくなります。(参照できなくなることをシャドーイング (shadow) といいます。) もっとも、同名の変数があるとややこしいので、避けたほうがよいでしょう*12。

```

void x(void) { printf("x"); }
void func(void) {
    int x=1; // ここに定義があるので
// int x=2; // コンパイルエラー
    {
        int x=3; // ブロックの内側なら OK
        {
            int x=4; // さらに内側のブロックなので OK
            printf("%d", x); // "4"
        }
        // このブロックにはxがないので直近のxを使う
        printf("%d", x); // "3"
    }
    printf("%d", x); // "3"
}
// printf("%d", x); // "1"
// x(); // コンパイルエラー (case C)
// int x=0; // コンパイルエラー (case B)
// void x(void) { printf("x"); } // コンパイルエラー (case A)

```

ここで注意したいのは、C 言語では識別子 (名前) に関して、関数と変数は区別せずに扱われる*13ことです。そして関数とグローバル変数は、スコープが同じであったことも思い出してください (☞ 7.2.3 項)。したがって、次のようになります。

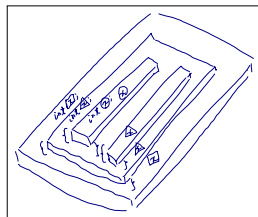
(case A) 同じ名前の関数を2つ作ることは (処理が同一でも) できません。

(case B) 関数と同じ名前のグローバル変数も作れません。

(case C) 関数と同じ名前のローカル変数を作ると、関数を呼び出せなくなります。

*12 Java のようなオブジェクト指向言語では、クラスに直属のメンバ変数がシャドーイングされても参照する方法があって、わざと同名の変数 (仮引数) にすることがよくあります。

*13 C 言語では、関数は、特別な型の変数として扱われています。関数の引数に、関数を渡すことも可能です。



コラム：return に用いる変数名

戻り値を格納する変数名を、関数名と同じにする初学者を散見します。その関数自身を呼び出すこと（再帰呼び出し）がシャドーイングでできなくなるだけで、ほとんど実害はないのですが、関数名と変数名が重複しないよう心がけている身からすると、違和感を覚えてしまいます。

return に用いる変数名は、どんな関数でも**とりあえず ret** にしている人もいます。

C 言語と同時期に誕生した Pascal という言語には return 文がありません。戻り値は、関数と同名の変数に代入することで示します。初学者が自然にやりそうなことを言語規格に取り入れてあったのだと気づくと、この仕組みを思いついた人の洞察力には感謝してしまいます。

コラム：複数の値を返す関数?

C 言語の関数は、引数ではたくさんの値を受け取ることができるのですが、返す値は最大でも 1 つだけに制限されています。コンパイラ的设计を楽にするための、一種の割り切りとも思えます。

しかし現実には、複数の値を返したい場面もあります。そのような時の工夫がいくつかありますが、多くは今後学ぶ知識が必要になるので、今の段階では、そのような方法もあるのだ、ぐらいにとどめておいてもらえば構いません。

- 2 つの関数に分離する。
- 構造体を返す。(10 章)
- 変数の配列 (ポインタ) を受け取って、関数側で値を書き換える。(9 章)
次の方法では失敗しますので、注意が必要です。(ヒント：寿命)
- 関数で確保したローカル変数の配列 (ポインタ) を返す。(8 章)

表 7.2 単語の連結戦略

名称	スネークケース (snake case)	キャメルケース (camel case)
方法	単語の間に_をはさむ	途中の単語の先頭大文字
例	snake_case get_string_length	camelCase getStringLength
使用例	(小文字) C 言語 (大文字) 各言語の定数	Java, C# 先頭大文字の使い分けあり



7.3 変数や関数の命名規則や習慣

多くのコンピュータ言語では、変数名や関数名（識別子）にスペースもハイフンも含めることが許されません。そこで、複数の単語からなる変数名を、どのように表現するのか、工夫が凝らされています。この習慣は言語によっても異なります。このような違いが生まれたのは、その言語の標準関数名の影響が大きいように思われます。世間でよく使われるのは、次の2通りです。

スネークケース (snake case) `get_string_length` のように、単語の間にアンダースコア (.) をはさみます。C 言語ではこれが主流で、変数と関数はすべて小文字、マクロはすべて大文字にする習慣があります。ほかの言語でも、定数には大文字のスネークケースがよく用いられます。

キャメルケース (camel case) `getStringLength` のように、途中の単語の先頭 1 文字だけ大文字にします。Java や C# でよく使われます。先頭も大文字にした **アップーキャメルケース** に異なる意味を持たせる場合もあります。

もちろん、この規則が素直に適用できない場合もあります。GCM のような元から大文字の単語を使おうとすれば、`get_GCM` か `get_gcm` かどちらにしてよいのかもわかりませんから、その場その場で柔軟に考えればよいでしょう。

並べる単語の順番は、「動詞 + 目的語」*14 にすることが多くなってきました。論理型なら「is+ 形容詞」「has+ 過去分詞」などにする習慣 (☞ 4.5 節) があります。

*14 オブジェクト指向言語の習慣が影響しているように思えます。

表 7.3 変数名の例

悪い名前の例	問題点	良い名前の例	改善点
× flag	いつ TRUE になるのか?	○ is_first	最初なら TRUE
× table	何を変換する?	○ binary2ascii	内部表記を表示用に変換
× no_data	否定は !no_data	○ exist_data	否定は !exist_data

長い単語の省略パターンは、initialize を init にするなど、業界標準的なものもあります。number を num とするのは、本書でも採用していますが、時代とともに長い単語のまま使う場面も増えてきています。first ↔ last、upper ↔ lower のような、よく使われる対義語のペアもあります。

ちょっと変わったところでは、deg2rad のように、変換するものの名前に現れる“2”は、数字 (two) の意味ではなく、発音の同じ“to”の意味を借用しています。

逆に避けるべきといわれているのは、flag や table のように、いつ TRUE になるフラグ (論理型変数) か、何を変換するテーブル (対応表) なのかわからない名前です。良い名前は、例えば is_first とすれば最初が TRUE と誰にでもわかります。binary2ascii とすれば内部表記 (バイナリ値) を表示用に変換するテーブルだろうと想像できます。

no_data のような、否定の入った論理型変数もやめておきましょう。逆の条件が !no_data と、二重否定になって読みにくくなります。真偽を逆にして exist_data の名前にしておくと、否定が一回ですみます。

このような習慣はたくさんあります。他人のプログラムを読みながら、少しずつ覚えていきましょう。

7.4 練習問題

1. [ループ vs. 公式]

1 から n (> 0) までの整数の和 $\sum_{k=1}^n k$ を求めたい。次の 2 通りの方法で実装せよ。

- int sum(int n) はループで積算せよ。
- int sum2(int n) は等差数列の和の公式 $\frac{n+1}{2}n$ を用いよ。

この 2 つの関数の値が $n = 50000$ 程度で一致するか調査せよ。そして、同じ結果を返すはずの関数を複数の方法で実装することが、信頼性にどのように影響するのか考察せよ。

2. [信頼性]

正の整数 x, y の最大公約数を返す関数を作りたい。

- int gcm(int x, int y) は、ループで共通する約数の最大値を求めよ。
- int gcm2(int x, int y) は、ユークリッドの互除法で実現せよ。

3. [1組の値を返すペアの関数]

分数の足し算 $\frac{b}{a} + \frac{d}{c}$ の結果を表す分数 $\frac{bc+ad}{ac}$ を、約分後の状態で得たい。簡単のため $a, b, c, d > 0$ とする。

C言語の関数の戻り値は1つだけなので、分数を表すためにペアで使う2つの関数を作ることにする。つまり、結果を表す分数の、

- 分子を返す `int add_numer(int a, int b, int c, int d)`
- 分母を返す `int add_denom(int a, int b, int c, int d)`

を作れ。これらの関数は、それぞれ $bc+ad$ と ac を、この2つの最大公約数で割ってから返せばよい。最大公約数を得るのに2. で作った関数を用いてよい。

4. [getter/setter]

以下のプログラム (大きなプログラムの一部分) を指示に従って完成させ、作った2つの変数と4つの関数をどのように運用すれば目標が達成できるか、考察せよ。

背景 作ろうとしている大きなプログラムは、指定された年月に関する処理をする。

目標 プログラム全体で共通する年と月を保持したい。ただし、個々の関数で範囲外のエラー処理を省くため、常に定められた範囲内の値にしておきたい。

指示 年と月のそれぞれについて、以下の関数をそれぞれ作れ^{*15}。

(`get.???`) 保持している値を返す。(必ず定められた範囲内の値を返すこと。)

(`set.???`) 引数で指定された値を保持する。(範囲外なら何もしなくてよい。)

範囲 年は1970から2100の整数をとる。月は1から12の整数をとる。

```
static int year = 2000;
static int month = 1;

/* getter */
int get_year(void) { return year; }
int get_month(void) { /* ここを作る */ }

/* setter */
void set_year(int y) {
    if (1970 <= y && y <= 2100) {
        year = y;
    }
}
void set_month(int m) {
    /* ここを作る */
}
```

5. [関数の役割分担] 📖 12.1.1 項

^{*15} 一般に、内部で使う変数の、値を取り出す関数を `getter`、値を代入する関数を `setter` と呼びます。慣用語としては、これらを対にして作ります。

第 8 章

配列

駐車場には駐車スペースが多数あるでしょう。駐車開始時刻を変数で覚えるとしたら、駐車スペースの数だけの変数が必要です。

隣の駐車スペースでも、同じ駐車場内であれば料金計算などの方法は共通でしょう。プログラム上でも共通の処理で表現したいところです。しかし、変数名が違うものになっていると、何度も似たようなことを書き並べることになって大変です。

違う駐車場では、駐車スペースの数が異なることでしょう。駐車場ごとにプログラムを修正するとしたら、これまた手間がかかります。

この章では、複数並んだ変数を定義する配列について説明します。配列は、同種のデータを並べて規則的に扱えることに意味があります。

キーワード

- 配列の定義・初期化子
- 要素数
- 添字（要素番号）
- マクロ定数
- 0 始まり・1 始まり
- 多次元配列・配列の配列
- 参照渡し

8.1 配列の定義と初期化

複数の変数をまとめて扱うための仕組みが**配列** (array) です。配列を定義するには、変数定義のときの変数名に続けて、角括弧 [] と、変数の個数を書きます。右下の例のように [3] と書けば3個の変数ができあがります。できあがった個々の変数を、配列の**要素** (element) といいます。つまり [] の中には**要素数** (number of elements) を書きます。

```
/* 文法 */
型名 変数名[要素数];
```

```
/* 実例 */
int a[3];
```

配列の要素1つを指すには、何番目の要素かを示す必要があります。このときにも [] と要素の番号を書きます。この要素番号を**添字** (subscript) とか**インデックス** (index) といいます。添字は、C言語では0から数え始めることになっているので、3個の要素を用意したら、使える添字は0, 1, 2です。そして関数内で定義すると、配列要素の初期値は、普通の変数と同じく不定ですから、代入してから使います。

```
/* 普通の変数 */
int a0, a1, a2;
a0 = 0;
a1 = 10;
a2 = 20;
```

```
/* 配列変数 */
int a[3];
a[0] = 0;
a[1] = 10;
a[2] = 20;
```

初期化 (変数定義と同時の代入) は、以下のように、値を書き並べてブロックで囲みます。この列挙された個々の値を**初期化子** (initializer) といいます。配列の要素数を初期化子の個数に合わせるなら、要素数を省略できます。

```
/* 普通の変数 */
int a0 = 0, a1 = 10, a2 = 20;
```

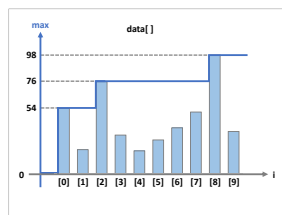
```
/* 配列変数 */
int a[3] = { 0, 10, 20 };
int b[] = { 0, 10, 20 }; //省略可
```

頻出ミス

(1) 初期化子は、変数定義と分離できません。(2) 配列同士の代入もできません。配列全体をコピーするには、要素を一つずつ代入します。(☞ 8.4.3 項)

```
int a = 1, b;
b = 1; // OK
b = a; // OK
```

```
int a[3] = {1,2,3}, b[3];
// b[3] = {1,2,3}; // NG(1)
// b = a; // NG(2)
```

配列は、名前（変数名）ではなく、番号（添字）で変数を区別するものです。これを利用すると、例えば、いくつもの変数の合計をループで求められます。

```
/* 普通の変数 */
int sum = a0 + a1 + a2;
```

```
/* 配列変数 */
int sum = 0;
for (int i=0; i<3; i++) {
    sum = sum + a[i];
}
```

普通の変数では、3 個版、4 個版と、個数に応じて違う式を用意せねばなりません、配列なら、ループ回数を変えるだけの同じ処理ですみます。努力と根性ではなく、うまい仕組みを使って簡単にすませましょう。

頻出ミス

`int a[3];` と用意した配列で `a[3]` の要素は使ってはいけません。しかも、もし使ったとしてもコンパイルエラーにもならなければ、警告も出ないのが普通です。

8.1.1 配列要素の最大値

配列要素の最大値を求めてみましょう。合計を求めるのに、先ほどはループ中に暫定の合計値を更新していきました。同じ発想で、暫定の最大値を更新しましょう。

ソースコード 8.1 では、100 点満点のテストの点数のデータ 10 個を `data[]` に用意しました。`max` を暫定の最大値として使うので、0 で初期化します。初期値は 0 でも -100 でも、小さい値なら何でもよくて、`max` はループ中に徐々に大きくなります。

ループ内では、`data[i]` を順に調べて、`max` より大きいときに `max` を更新します。変数の変化の経過を表示したので、動きをよく理解しておきましょう。

ソースコード 8.1 の実行結果

```
data[0] = 54, max = 54
data[1] = 20, max = 54
data[2] = 76, max = 76
data[3] = 32, max = 76
data[4] = 19, max = 76
data[5] = 28, max = 76
data[6] = 38, max = 76
data[7] = 51, max = 76
data[8] = 98, max = 98
data[9] = 35, max = 98
max = 98
```

ソースコード 8.1 配列の最大値

```

1 #include <stdio.h>
2
3 int main(void) {
4     int data[10] = {           // 0から100の点数のデータ
5         54, 20, 76, 32, 19, 28, 38, 51, 98, 35
6     };
7     int max = 0;              // 暫定最大値
8     for (int i=0; i<10; i++) {
9         if (max < data[i]) { // より大きな値を見つければ
10            max = data[i]; // 暫定値を更新する
11        }
12        printf("data[%d] = %d, max = %d\n", i, data[i], max); // 経過
13    }
14    printf("max = %d\n", max); // 最終結果
15    return 0;
16 }

```

8.2 要素数とマクロ定数

ところで、ソースコード 8.1 には気になるところがあります。「10」という数値が2ヶ所にあります。4行目の配列要素数と、8行目のループ回数です。この2つが食い違っていると、プログラムは正しい動作ができません。もっと言うと、5行目のデータの個数を数えて、その値を採用せねばなりません。今は絶妙にバランスをとっていますが、将来、簡単に破綻することでしょう。これはプログラムの構造上の欠陥です。

この「10」のような、恣意的で、それだけでは意味のわからない数値のことを、マジックナンバー (magic number) と呼んで、プログラマは忌み嫌います。欠陥の隠れているサインだからです。マジックナンバーでなくすためには、意味がわかるように名前をつけます。例えば下の例のように、N_DATA のような名前のマクロを定義して、「10」を置き換えます。すると、マクロで2ヶ所の数値がいつでも同じになるので、欠陥が解消されます。

```

#define N_DATA 10 /* 要素数をマクロで定義 */
int data[N_DATA] = { 54, 20, 76, 32, 19, 28, 38, 51, 98, 35 };

for (int i=0; i<N_DATA; i++) { ... }

```

配列要素数は、定数 (コンパイル時に決まる値) で指定する必要があるので、このようにマクロによる定数 (マクロ定数) を使います。N_ の接頭辞は、number of の意味で、個数を表すのによく使われます。

コラム：マクロの定数式

マクロ定数には、定数を用いた式を定義しても構いません。ただし、マクロはソースコード上の文字の置き換えを指示するものなので、式の中で使うと、優先順位が考慮されません。ですから、定義する式の全体をカッコで囲うのが安全です。

```
#define SIZE 1+1 // ×
int a[SIZE * 2]; // 1+1*2 = 3
```

```
#define SIZE (1+1)//○
int a[SIZE * 2]; // (1+1)*2 = 4
```

頻出ミス

マクロ定義の行末にセミコロン (;) を書いても、文字の置き換え指示としては正なので、エラーになりません。マクロを使ったところでエラーになりますが、原因がわかりにくいので要注意です。

```
#define SIZE 10; // 原因
// int a[SIZE]; // ここでエラー ← // int a[10;]; // NG
```

コラム：可変長配列

C99 では、配列がローカル変数ならば、要素数が変数でも大丈夫です。変数の値に応じて長さが変わる可変長配列 (variable-length array) が実現されています。

```
/* 配列がグローバル変数 */
int n = 10;
// int a[n]; // コンパイルエラー
```

```
/* 配列がローカル変数 */
int func(int n) {
    int a[n]; // C99のみOK
```

とはいえ、出来上がった配列の長さを、後から伸縮できるわけではありません。また、C11 ではオプション機能に格下げになり、C++ には取り込まれていない機能なので、使ってよい場面は限られています。

8.2.1 初期化子の個数

先ほどのソースコード 8.1 には、まだ問題が残っています。マジックナンバーをマクロで置き換えたとしても、データを追加あるいは削除すると、個数を数えなおさねばなりません。手作業だと間違えそうなので、プログラム（コンパイラ）にやらせましょう。

配列要素数を省略すると、初期化子の個数ぴったりの配列になります。この配列の要素数を計算する慣用句があります。sizeof 演算子は、変数の占める領域の大きさ（バイト数）を教えてください*1。「sizeof(data)」が配列全体の大きさ、「sizeof(data[0])」が要素 1 個の大きさなので、この 2 つの割り算が要素数になります。8 行目のループの回数をこの要素数で置き換えると、個数を気にせずデータを書き並べてよくなります。

```
int data[] = { 54, 20, 76, 32, 19, 28, 38, 51, 98, 35 };
int num = sizeof(data)/sizeof(data[0]); /* 要素数を計算で求める */

for (int i=0; i<num; i++) { ... }
```

ところで、配列の最後の初期化子の後にコンマは不要ですが、書いてもエラーになりません。初期化子を行単位で入れ替えたり追加削除するような場面では、すべての行末にコンマを書いた方が統一的ですので、気にせず書いておきましょう。

```
int data[] = {
    54, 20, 76, 32, 19,
    28, 38, 51, 98, 35, // 最後のコンマは余分だが、書いてもよい
// 0, // ここを有効にしたときに、上の行の最後のコンマが役立つ
};
```

コラム：初期化子の過不足

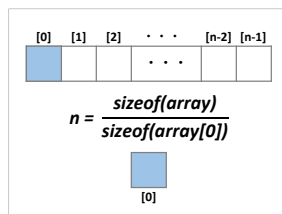
変数定義で、配列の要素数と初期化子の両方が指定されていると、先頭の初期化子から順に消費されていき、足りなければ 0 が補われます。ただし、なぜか初期化子を空にすることは言語規格上は認められていません^a。したがって、すべての配列要素を 0 で初期化する慣用句は、初期化子をひとつだけ書く「int a[100] = {0};」^bです。

逆に、初期化子が余ると、コンパイルエラーか警告になります。

^a コンパイラにとっては、初期化子が空でも害がないのか、警告すらされないことが多いです。

^b C 言語の言語規格に強く依存した手法なので、ループで 0 を代入するのもよいでしょう。

*1 sizeof 演算子は、2.9 節では型に使用しましたが、変数でも計算できます。型の場合は「sizeof(int)」のように () が必要ですが、変数では「sizeof data」と省略しても構いません。



8.3 関数に配列を渡す

配列を、関数の引数に渡すことができます。次のようにします。

仮引数 受け取る関数の仮引数には、変数定義のときのように、変数名に続けて [] を書きます。要素数は、後で述べますが、省略できる場合があります。

実引数 呼び出し側は、配列の変数を用意して、実引数には**変数名だけ**を書きます。

```
/* 引数が普通の変数 */
void func(int x);

int main(void) {
    int x = 0;
    func(x); // xはint
    ...
}
```

```
/* 引数が配列 */
void func2(int a[5]);

int main(void) {
    int a[5] = {0,0,0,0,0};
    func2(a); // aはint配列
    ...
}
```

頻出ミス

配列を渡すのに、変数名に要素数までつけて a[5] のようにすると、要素 1 つを取り出したこととなります。(しかも添字の範囲外アクセスです。)[] の役目は、変数定義では**要素数**、式に現れると**添字** (何番目) と違うので、要注意です。

```
void func(int x);
...
int a[5] = {0,0,0,0,0};
func(a[0]); //OK, a[0]はint
```

```
void func2(int a[5]);
...
int a[5] = {0,0,0,0,0};
//func2(a[5]); //NG, a[5]はint
```

ここで注意したいのは、そもそもの要素数の取り扱いです。例えば 5 個しか用意していない配列要素の 10 番目を使っても、C 言語には検出機能がなく、コンパイルエラーにも、ひょっとすると実行時エラーにもなりません。

そして関数に配列を渡しても、関数側では要素数を感知できません。つまりプロトタイプ宣言での配列要素数は、いくつを指定しても同じことで、省略すら可能です*2。そのため、要素数が必要なときの慣用句は、右の例のように、別の引数で渡します。

```
void func3(int num, int a[]);

int main(void) {
    int a[5] = {0,0,0,0,0};
    func3(5, a);
    ...
}
```

8.3.1 配列は参照渡し

関数とのやりとりに配列を用いると、次のような動作になります。

引数 配列自体をコピーせず、メモリ上のアドレス（ポインタ）が渡されます。参照渡し（call by reference）相当になって、関数で、呼び出し元の配列を変更できます。

戻り値 配列自体は返せません。return 文に配列名を与えると、アドレスが返されます。

これまでの変数はコピーされましたが、配列では、全要素の値をコピーすると非効率ですから、このような動作になっています。関数が配列を返したくなったら、本来は入力であるはずの引数を出力に転用します。例えば、与えられた配列の n 個の要素を 0 で埋める関数は右のようになります。

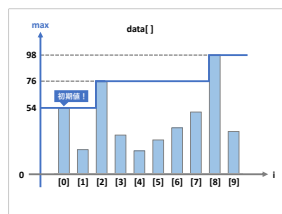
```
/* n個の要素を0にする */
void clear(int n, int a[]) {
    for (int i=0; i<n; i++) {
        a[i] = 0;
    } // ↑呼び出し元の配列が変化
}
```

また、3.9.2 項でできなかった、関数でのスワップ（値の入れ替え）が、配列なら可能です。

```
/* a[] の i, j 要素を入れ替える */
void swap(int a[], int i, int j) {
    int tmp = a[i]; a[i] = a[j]; a[j] = tmp;
}
```

もっとも、関数でスワップするには、無理に配列に頼らず、9 章で学ぶポインタを使います。あくまでも、配列は「ポインタのような性質を持っている」と理解してください。

*2 正確には、多次元配列では最初の 1 次元の要素数だけ省略できます。（☞ 8.6.1 項）



8.4 配列を順番に操作する

8.4.1 配列の最大値を返す関数

ソースコード 8.1 を関数版に書き換えてみましょう。

ソースコード 8.2 では、最大値を返す関数 `max_array()` を作ってみます。引数は、配列サイズと `int` 配列です。関数内の処理を見てみましょう。暫定最大値 `max` を小さな値で初期化したいのですが、配列がテストの点数とは限らない状況を思い浮かべると、0 でも大きすぎる可能性があります。そこで5行目では、配列の先頭要素を使ってみました。これは慣用句です。6行目のループは `i=0` を除いてよくなります。ただし `n=0` だと都合が悪いので、本来なら検出してエラー処理すべきところです。

ソースコード 8.2 配列の最大値（関数版）

```

1 #include <stdio.h>
2
3 /* x[0]..x[n-1] の最大値を返す (n>0) */
4 int max_array(int n, int x[]) {
5     int max = x[0];           // 初期値は先頭要素
6     for (int i=1; i<n; i++) { // i=0 は不要
7         if (max < x[i]) {    // より大きな値を見つければ
8             max = x[i];      // 暫定値を更新する
9         }
10    }
11    return max;
12 }
13
14 int main(void) {
15     int data[] = { // 0から100の点数のデータ
16         20, 54, 76, 32, 19, 28, 38, 51, 98, 35, // 末尾のコンマは有益
17     };
18     int num = sizeof(data)/sizeof(data[0]); // 要素数を計算で求める
19     printf("max = %d\n", max_array(num, data));
20     return 0;
21 }

```

8.4.2 0 始まり・1 始まり

毎月の日数を配列に保存しましょう。(簡単のため、閏年は除外して平年だけを考えます。) 12ヶ月分のデータが必要ですから、配列の要素数を12にしてみます。添字は0始まりなので、1月は0番めです。月名と日数を表示しようとする、ソースコード 8.3 のように、月の数字^{*3}と、配列の添字がずれるので、+1 や -1 の調整項が必要になります。

ソースコード 8.3 月ごとの日数 (0 始まり)

```

1 #include <stdio.h>
2
3 #define N_MONTH 12
4 int days[N_MONTH] = {
5     31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31
6 }; //1月 2月 3月 4月 5月 6月 7月 8月 9月 10月11月12月
7
8 int main(void) {
9     for (int i=1; i<=N_MONTH; i++) {
10        printf("%d月は%d日あります\n", i, days[i-1]);
11    }
12    return 0;
13 }
```

そこで、せっかく確保される0番めの要素は捨ててしまって、配列要素を1オリジン(1始まり、☞84ページのコラム)で使ってみましょう。確保する要素は13個に増えます。ソースコード 8.4 では、0番めの初期化子には-1と、明らかに無効な値を書きおきました。1回だけの配列定義には+1の調整項がありますが、何度も使うループ処理はすっきりとします。

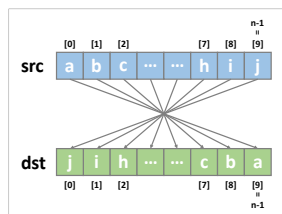
ソースコード 8.4 月ごとの日数 (1 始まり)

```

1 #include <stdio.h>
2
3 #define N_MONTH 12
4 int days[N_MONTH + 1] = {
5     -1/*1オリジン*/, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31
6 }; //1月 2月 3月 4月 5月 6月 7月 8月 9月 10月11月12月
7
8 int main(void) {
9     for (int i=1; i<=N_MONTH; i++) {
10        printf("%d月は%d日あります\n", i, days[i]);
11    }
12    return 0;
13 }
```

*3 英語圏では、月を名前 (Jan., Feb., ...) で呼ぶので、0 始まりで違和感がないようです。

i の式	dst[] の添字		src[] の添字		
	n の式	$n = 10$	$n = 10$	n の式	n, i の式
i	0	0	9	$n-1$	$(n-1)-i$
i	1	1	8	$n-2$	$(n-1)-i$
i	2	2	7	$n-3$	$(n-1)-i$
\vdots	\vdots	\vdots	\vdots	\vdots	\vdots
i	$n-3$	7	2	2	$(n-1)-i$
i	$n-2$	8	1	1	$(n-1)-i$
i	$n-1$	9	0	0	$(n-1)-i$



8.4.3 配列の正順・逆順コピー

配列をコピーしたければ、要素を一つずつ代入します。配列を関数に渡すと参照渡しになるので、ソースコード 8.5 の `array_copy()` のような関数が作れます。コピー元の `src` は source (源泉、情報源)、コピー先の `dst` は destination (目的地) の省略形です。

逆順にコピーするには、添字をうまく計算すれば大丈夫です。まず、 $n = 10$ のような具体的な値で `dst[0]=src[9], ..., dst[9]=src[0]` を実行したい、と考えます。次に、`dst[]` に i を使うことにします。そして、`src[]` の添字を n と i で表せれば、ループ処理の完成です。逆順の処理にどうしても -1 の調整項が出てくることは、覚えておいてもよいでしょう。

ソースコード 8.5 配列要素のコピー

```

1 #include <stdio.h>
2
3 void array_copy(int n, int dst[], int src[]) {
4     for (int i=0; i<n; i++) {
5         dst[i] = src[i]; // 1要素ずつコピー
6     }
7 }
8
9 void array_reverse(int n, int dst[], int src[]) {
10    for (int i=0; i<n; i++) {
11        dst[i] = src[(n - 1) - i]; // 逆順にコピー
12    }
13 }
14
15 int main(void) {
16    int orig[] = { 5, 2, 7, 5, 9, 1, 5, 7, 3, 0 };
17    int num = sizeof(orig)/sizeof(orig[0]);
18    int copy[num];    array_copy(num, copy, orig); // コピー
19    int reverse[num]; array_reverse(num, reverse, orig); // 逆順
20    for (int i=0; i<num; i++) { // 結果の表示
21        printf("%d %d %d\n", orig[i], copy[i], reverse[i]);
22    }
23    return 0;
24 }

```

8.5 配列をランダムに操作する

これまでの例では、配列にはデータが並んでいるだけで、添字がいくつであっても、あまり意味はありませんでした。ここでは、添字の値に意味のある操作をしてみます。

8.5.1 エラトステネスのふるい

素数とは、2 以上の整数のうち、1 と自分自身以外に約数を持たないものです。

6.4.5 項では 1 つの整数が素数かどうかを判定しましたが、ここでは 100 までのすべての素数を列挙してみましょう。これには、割り算も掛け算も使わず、足し算と、素数かどうかの表だけで求める**エラトステネスのふるい** (sieve of Eratosthenes) というアルゴリズムが有用です。詳細な手順は 131 ページのコラムを参照してください。

このアルゴリズムをプログラムで実現してみます。数を並べた表は、配列で表現して、**数を添字に対応付け**ます。つまり、2 が表に残っているのなら、`is_prime[2]` は `TRUE` です。4 を消すには `is_prime[4]` を `FALSE` にします。このように、配列の添字の値に意味を持たせるところが、新しい使い方です。

ソースコード 8.6 では、`is_prime[]` を 7 行目で確保します。表に数が並んでいる状態にするために、10 行目で `TRUE` に初期化します。11 行目では、小さな数から素数を探のですが、0 と 1 は除外して、2 以上をループで調べます。12 行目で `i` が素数だとわかれば、14–16 行目で倍数を消します。`i` の倍数を作り出すのに、`i` ずつ加算しています。

コラム：添字の型

配列の添字には、整数型を使います。浮動小数点型 (`double` や `float` など) では、たとえその数値の小数部分が 0 であっても、コンパイルエラーになります。

```
int a[10];
double x = 1.0;
// a[x] == 5; // コンパイルエラー
```

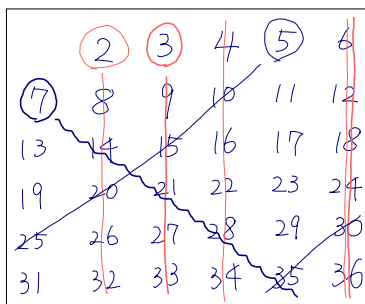
C 言語では、ループ変数は整数型にしておくのが無難です。浮動小数点型だと、配列の添字に使えないだけでなく、0.1 ずつの足し算をすると、誤差が積もってループ回数が 1 回ずれるかもしれないからです。(☞ 13.7.1 項)

コラム：エラトステネスのふるいのアルゴリズム

100 までの素数を求めるには、まず表に (1 は素数ではないので) 2 から 100 までの数を書き並べます。そして、小さい数から順に調べてゆき、消されていない数 (= 素数) を見つけるたびに、「その素数自身は残して、倍数を消す」を繰り返します。

具体的な操作は、次のようになります。

- 2 は表に残っているので、素数です。4, 6, 8, ..., 100 を消します。
- 3 も表に残っているので、素数とわかります。6, 9, 12, ..., 99 を消します。
- 4 は既に消されているので、素数ではありません。何もせずに次に進みます。この操作を最後まで続けると、素数だけが表に残ります。



ソースコード 8.6 エラトステネスのふるい

```

1 #include <stdio.h>
2
3 #define FALSE 0
4 #define TRUE 1
5
6 #define N 100
7 int is_prime[N + 1]; // i が表にあるなら is_prime[i]=TRUE
8
9 int main(void) {
10     for (int i=2; i<=N; i++) is_prime[i] = TRUE; // 初期化
11     for (int i=2; i<=N; i++) { // 小さい数から調べる
12         if (is_prime[i]) { // 素数を発見
13             printf("%d\n", i);
14             for (int j=i+i; j<=N; j+=i) {
15                 is_prime[j] = FALSE; // 倍数を表から消す
16             }
17         }
18     }
19     return 0;
20 }

```

8.5.2 度数分布（ヒストグラム）

与えられたデータの値ごとの出現回数（度数）を数えてみます。この統計値は**度数分布** (frequency distribution)、グラフに描くと**ヒストグラム** (histogram) といいます。

何人分かの点数のデータがあるとしましょう。もし手作業で出現回数を数えるのなら、点数ごとに「正」の字を書くところです。つまり、先頭のデータから順に調べて、その点数に応じた「正」に1画ずつ書き加えていきます。「正」の字を書く場所は、**点数のとりうる種類の数**だけ必要ですが、このおかげで同じデータを何度も読み返さずに済みます*4。

この作業をプログラムで実現しましょう。ソースコード 8.7 では、6-8 行目で点数データを `data[i]` に格納し、点数の最大値を3行目のマクロ定数 `POINT_MAX` に定義しました。

出現回数を数える配列は10行目の `hist[]` です。「正」の字を書く場所に対応します。`hist[0]` は0点の人数、`hist[1]` は1点の人数というように割り当てることにして、すべて0で初期化しておきます（13行目）。

`data[i]` には、先頭から5, 2, 7, ... が格納されているので、「正」の1画を書き加えることに対応する操作は `hist[5]++`; `hist[2]++`; `hist[7]++`; ... です。これをループで実現したのが16-17行目です。この2行はまとめて「`hist[data[i]]++`」と短く書くこともできます。

プログラムには2種類の配列（データと出現回数）が現れるので、一見すると煩雑ですが、**配列ごとにいつも同じループ変数**を使っていることに着目すると、規則的に見えてきます。ここでは下の表のように、`data[i]` と `hist[j]` の組み合わせにしています。

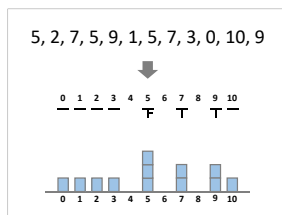
ソースコード 8.7 の実行結果

```
0点が1人います
1点が1人います
2点が1人います
3点が1人います
4点が0人います
5点が3人います
6点が0人います
7点が2人います
8点が0人います
9点が2人います
10点が1人います
```

目的	配列名	ループ変数	ループ変数の範囲
データ	<code>data</code>	<code>i</code>	$0 \leq i < \text{num}$
出現回数	<code>hist</code>	<code>j</code>	$0 \leq j \leq \text{POINT_MAX}$

なお、12行目と19行目の `j` のループは、一見すると良いループ（☞ 5.5 節）から外れて、0始まりなのにイコール付きですが、10点満点なら11通りの点数があるので、これで正解です。

*4 逆に、場所を節約したければ、データを何度も読み直すことになって、0点の人を見つければ終わったら、次はまた先頭から1点の人を探す、という方法もあります。（☞ 133 ページの頻出ミス）



ソースコード 8.7 ヒストグラム

```

1 #include <stdio.h>
2
3 #define POINT_MAX 10
4
5 int main(void) {
6     int data[] = {           // データ：添え字はiにする
7         5, 2, 7, 5, 9, 1, 5, 7, 3, 0, 10, 9
8     };
9     int num = sizeof(data)/sizeof(data[0]);
10    int hist[POINT_MAX + 1]; // 出現回数：添え字はjにする
11
12    for (int j=0; j<=POINT_MAX; j++) { // 出現回数の初期化
13        hist[j] = 0;
14    }
15    for (int i=0; i<num; i++) {        // 出現回数を数える
16        int d = data[i]; // data[i] の出現回数を1増やす
17        hist[d]++;        // 2行をまとめて hist[data[i]]++; でもよい
18    }
19    for (int j=0; j<=POINT_MAX; j++) { // 結果表示
20        printf("%d点が%d人います\n", j, hist[j]);
21    }
22    return 0;
23 }

```

頻出ミス

10-21 行目を、以下の 2 重ループの処理に置き換えると、配列は不要になりますが、計算時間が飛躍的に増えるので (POINT_MAX 倍)、実用的ではありません。

```

for (int j=0; j<=POINT_MAX; j++) {
    int count = 0;
    for (int i=0; i<num; i++) {
        if (data[i] == j) count++;
    }
    printf("%d点が%d人います\n", j, count);
}

```

8.5.3 覆面算 (発展的内容)

たとえば $\frac{\text{SEND} + \text{MORE}}{\text{MONEY}}$ のようなアルファベットに数字を割り当てて、 $\frac{9567 + 1085}{10652}$ のように式を成り立たせる、という古典的なパズルがあります [11]。覆面算といいます。同じアルファベットには同じ数字を、異なるアルファベットには異なる数字を割り当てます。最上位桁 (この例では S と M) は 0 を除外します。正しい割当が複数存在する問題もありますが、この例では上記が唯一の割当です。

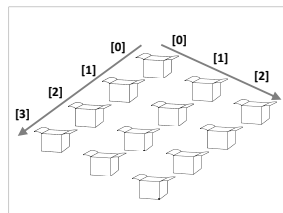
プログラムで探索させると、ソースコード 8.8 のようになります。異なる数字を割り当てるために、数字が使用済みかどうかを論理型の配列で管理しています。インデントは褒められたものではありませんが、8重ループなので、いたしかたないでしょう。

ソースコード 8.8 覆面算

```

1 #include <stdio.h>
2 #define FALSE 0
3 #define TRUE 1
4
5 int val(int a, int b, int c, int d, int e) { // 桁ごとの数をintに変換
6     return 10000*a + 1000*b + 100*c + 10*d + e;
7 }
8 int main(void) {
9     int used[10]; // 数字iが使用済みなら used[i]はTRUE
10    for (int i=1; i<10; i++) { used[i] = FALSE; } // 初期化
11    for (int s=1; s<10; s++) { if (!used[s]) { used[s] = TRUE; }
12    for (int e=0; e<10; e++) { if (!used[e]) { used[e] = TRUE; }
13    for (int n=0; n<10; n++) { if (!used[n]) { used[n] = TRUE; }
14    for (int d=0; d<10; d++) { if (!used[d]) { used[d] = TRUE; }
15    for (int m=1; m<10; m++) { if (!used[m]) { used[m] = TRUE; }
16    for (int o=0; o<10; o++) { if (!used[o]) { used[o] = TRUE; }
17    for (int r=0; r<10; r++) { if (!used[r]) { used[r] = TRUE; }
18    for (int y=0; y<10; y++) { if (!used[y]) { used[y] = TRUE; }
19        if (val(0,s,e,n,d) + val(0,m,o,r,e) == val(m,o,n,e,y)) {
20            printf(" %d%d%d%d\n+ %d%d%d%d\n-----\n %d%d%d%d%d\n",
21                s,e,n,d,      m,o,r,e,      m,o,n,e,y);
22        }
23        used[y] = FALSE; }} // 9567
24        used[r] = FALSE; }} // + 1085
25        used[o] = FALSE; }} // -----
26        used[m] = FALSE; }} // 10652
27        used[d] = FALSE; }}
28        used[n] = FALSE; }}
29        used[e] = FALSE; }}
30        used[s] = FALSE; }}
31        return 0;
32    }

```



8.6 2次元配列

配列は直線上に並んだ多数の箱のようなものでした。多数の箱が、平面に2次元の広がりできき詰めることもできるように、配列も2次元の添字を使うことができます。変数定義でも、式中でも、変数名に続けて [] を2回繰り返します。

```
/* 1次元配列 */
int a[6];
a[0] = 1; a[1] = 2;
a[2] = 3; a[3] = 4;
a[4] = 5; a[5] = 6;
```

```
/* 2次元配列 */
int a[3][2];
a[0][0] = 1; a[0][1] = 2;
a[1][0] = 3; a[1][1] = 4;
a[2][0] = 5; a[2][1] = 6;
```

初期化子は、2次元だとブロックが入れ子になります。要素数は**最初の次元だけ**が省略できます。

```
/* 1次元配列 */
int a[6] = {
    1, 2,
    3, 4,
    5, 6,
};
```

```
/* 2次元配列 */
int a[3][2] = {
    { 1, 2 },
    { 3, 4 },
    { 5, 6 },
};
```

2次元配列を、関数の引数として渡すこともできます。関数プロトタイプの変数名では、要素数は**最初の次元だけ**が省略できます。呼び出す際の実引数には、1次元のときと同じく、変数名のみを書いて、[] をつけません。

```
/* 1次元配列 */
int func(int a[6]);

int a[6];
func(a); // 実引数は変数名のみ
```

```
/* 2次元配列 */
int func2(int a[3][2]);

int a[3][2];
func2(a); // 実引数は変数名のみ
```

コラム：配列の配列

C 言語の 2 次元配列は、内部では添字を 1 次元相当に変換しています。M×N 要素の配列だと、以下のように、`[i][j]` は `[i*N+j]` のような動作をします。

```

/* 1次元配列 */
int a[M * N];
a[i * N + j] = 0;

/* 2次元配列 */
int a[M][N];
a[i][j] = 0;

```

`[i*N+j]` の計算式に N が必要なので、関数プロトタイプの仮引数で、2 次元目の要素数が省略できないことが理解できるでしょう。

メモリ上では、N 要素の int 配列が M 個並んでいるようなものですから、このタイプの 2 次元配列を「配列の配列」、また長さが揃っていることから「長方形配列」ということもあります。

なお、C 言語の 2 次元配列には「ポインタの配列」もあって、長さの揃わない配列を作ることできます (☞9.4 節)。

8.6.1 多次元配列

2 次元配列を作った方法を応用して、`[]` を繰り返すと 3 次元、4 次元のような**多次元配列**になります。初期化子や、関数との受け渡しの方法も、これまでの配列と同様です。省略できる要素数も、**最初の 1 次元だけ**です。

もっとも、多次元になってくると、変数の占める領域が大きくなるので、ローカル変数としては確保できなくなって*5、`static` をつけたり、グローバル変数にしてしまうこともよくあります*6。関数に渡すにしても、次元数を揃える必要があるため、ほぼ専用関数になって、引数で渡す意味が薄くなりがちです。本格的に利用するには、ポインタを駆使します (☞9.4 節)。

*5 ローカル変数を割り当てるメモリ (スタックメモリ) は、通常は数 MB 程度の領域しか確保されていません。さらに、領域をあふれた場合に、それを検出する機能は C 言語規格にはありません。つまり、コンパルエラーにも、実行時エラーにもならず、動作がおかしくなるだけの場合があります。

*6 割り当てられるメモリの種類が異なるため、数 MB のようなサイズの制限を受けませんが、それでも無尽蔵ではないので、最終的にはメモリを動的に確保することになります。

8.6.2 2次元配列の縦横合計

3人の5科目の点数データから、人ごとの合計と、科目ごとの合計を求めましょう。

人と科目の2種類のループが入り乱れるので、煩雑になりますが、ソースコード 8.9 では、ループ変数を、人には p 、科目には s と決めたので、少しは整理されているでしょう。添字の変数がいつでも同じになって、 $data$ なら $[p][s]$ 、人ごとの合計 sum_p なら $[p]$ 、科目ごとの合計 sum_s なら $[s]$ です。

ソースコード 8.9 2次元配列の縦横合計

```
1 #include <stdio.h>
2
3 #define PERSON 3
4 #define SUBJECT 5
5 int data[PERSON][SUBJECT] = {
6     { 80, 70, 40, 60, 80 }, // sum_p[0]
7     { 50, 90, 60, 40, 30 }, // sum_p[1]
8     { 70, 40, 70, 60, 50 }, // sum_p[2]
9 }; // sum_s[0] [1] [2] [3] [4]
10
11 void calc_sum(int data[PERSON][SUBJECT], int sum_p[], int sum_s[]) {
12     for (int p=0; p<PERSON; p++) sum_p[p] = 0; // 初期化(人)
13     for (int s=0; s<SUBJECT; s++) sum_s[s] = 0; // 初期化(科目)
14     for (int p=0; p<PERSON; p++) {
15         for (int s=0; s<SUBJECT; s++) {
16             sum_p[p] += data[p][s]; // 加算(人)
17             sum_s[s] += data[p][s]; // 加算(科目)
18         }
19     }
20 }
21
22 int main(void) {
23     int sum_p[PERSON], sum_s[SUBJECT];
24     calc_sum(data, sum_p, sum_s); // 合計を計算する
25     for (int p=0; p<PERSON; p++) {
26         for (int s=0; s<SUBJECT; s++) {
27             printf(" %2d ", data[p][s]); // データ表示
28         }
29         printf("| %d\n", sum_p[p]); // 合計表示(人)
30     }
31     for (int s=0; s<SUBJECT; s++) {
32         printf("%d ", sum_s[s]); // 合計表示(科目)
33     }
34     printf("\n");
35     return 0;
36 }
```

8.7 練習問題

1. [最大値]

ソースコード 8.2 の 7 行目の if の条件式の $<$ を \leq に置き換えると、どのように動作が変化するか考察せよ。また最小値を求める関数を作れ。

2. [最大値の添字]

ソースコード 8.2 を参考に、次の関数を作れ。

- `int max_index(int n, int x[])` は `x[]` の最大値の要素の添字を返す。

最大値をとる要素が複数あった場合の動作は、各自で考えよ。暫定最大値との比較に等号を含めるかどうかで、どのように動作が変化するか考察せよ。

3. [添字の範囲チェックと短絡評価]

m 月 d 日が正しい日付かどうかを調べたい。ソースコード 8.3 やソースコード 8.4 を参考にして次の関数を作れ。(閏年は考慮しなくてよい。)

- `int is_valid_date(int m, int d)` は、平年のカレンダーに m 月 d 日が存在すれば TRUE、そうでなければ FALSE を返す。

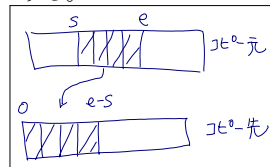
配列の添字範囲あふれに注意し、短絡評価 (☞ 4.6 節) の活用を検討せよ。

4. [配列の一部をコピーする]

ソースコード 8.5 を参考に、次の関数を作れ。

- `void slice(int dst[], int src[], int start, int end)` は、`src[start]` から `src[end-1]` までの $(end-start)$ 個の配列要素を、`dst[0]` から `dst[end-start-1]` までにコピーする。

$0 \leq start \leq end \leq (\text{src の要素数})$ と、 $(end-start) \leq (\text{dst の要素数})$ を前提としてよい。`src[end]` はコピーしないことに注意せよ。



5. [ヒストグラム]

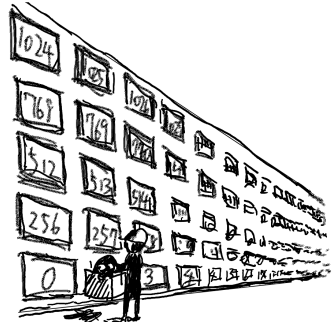
ソースコード 8.7 を参考に、100 点満点の成績データを作り、0 点から 10 点刻みのヒストグラム (0-9 点?人, 10-19 点?人, ...) を作れ。(ヒント: 100 点はどの区分に入るのかをよく検討せよ。)

6. [2 次元配列]

ソースコード 8.9 を改造して、3 人 5 科目 (のべ 15 科目) の点数の合計を求めて表示せよ。`calc_sum()` 関数の型を `void` から `int` に変更し、2 重ループで合計を求めて、戻り値とせよ。

第9章

文字列とポインタ



コンピュータ言語の「文字列」とは、複数の「文字」から構成されるもので、長さがあります。10文字の場合も、100文字の場合もあるでしょう。長さが1文字だと、単なる「文字」と同じに思えるかもしれませんが、コンピュータ言語にとっては、たまたま長さが1文字であっただけで、「文字列」と「文字」とは、決定的に異なるものとして扱います。さらに長さが0文字だと「空文字列」などと呼んで、文字列ではあっても中身が何もない、というものもよく登場します。

C言語の文字列は、専用の型があるわけではなく、char型の配列がその役目を担います。文字列のための特別な文法が少しあって、文字列操作のための標準ライブラリ関数がありますが、基本的には配列であるので、お世辞にも便利とは言えません。

とは言え、画面表示などには、やはり必要な機能ですから、配列操作のよい例題にもなるので、深入りせずに使い方を学びましょう。

配列と表裏一体のポインタについても、少しだけ触れておきます。

キーワード

- ヌル文字
- 文字列リテラル=初期化子
- <string.h>
- strlen(), strcpy(), strcat(), strcmp()
- ポインタ=アドレス
- const, *, &
- ポインタの配列

9.1 文字列=char の配列

C 言語の 1 文字は char 型で扱い、文字定数は 'A' のようにシングルコーテーションで囲って示しました。ここでは、複数の文字からなる**文字列** (string) を取り上げます。

文字列は、char の並んだものとして扱い、終端を表すために**ヌル文字** (null character) という、特別な文字 (文字コード 0) を配置します。つまり、文字列自体にはヌル文字を含められませんが、それ以外の char の並んだものなら、長さに制限なく扱えます。ヌル文字は、エスケープシーケンスの 8 進数コードを使って '\0' と表します。整数の「0」でも働きは同じですが、文字定数にすることで「文字である」という意図が読み取れます。

文字列は char の配列に格納して扱います。配列要素に 'A' のような文字定数を代入できますし、初期化子に文字定数を並べて構いません。

```
/* int配列 */
int a[4]; a[0]=10; a[1]=20; ...
int b[4] = { 10, 20, 30, 40 };
int c[] = { 10, 20, 30, 40 };
```

```
/* char配列 */ // △
char a[4]; a[0]='T'; a[1]='h'; ...
char b[4] = { 'T', 'h', 'e', '\0' };
char c[] = { 'T', 'h', 'e', '\0' };
```

頻出ミス

「The」の**3文字の文字列**を格納するのに、char 配列は**4要素**が必要です。終端のヌル文字を忘れがちですので、右上の例の初期化子は**実際には用いません**。

初期化子には、文字列専用の文法があります。ダブルコーテーションで囲った**文字列リテラル**が初期化子のブロックの役目を果たし、しかも終端のヌル文字を含みます。これなら編集も容易ですし、要素数を省略すれば**ヌル文字分を忘れない**ので、実用的で安全です。

```
/* int配列 */
// 対応する機能なし
```

```
/* char配列 */
char d[4] = "The"; // △3ではない
char e[] = "The"; // ○推奨
```

変数定義の後に代入したくなくても、初期化子ブロックが代入できないのと同じように、文字列リテラルも代入できません。もちろん 1 文字ずつの代入はできますが、実用的ではありません。通常はこの後に紹介する、標準ライブラリの文字列コピー関数を使います。

```
int f[4];
// f[4] = { 10, ... }; // NG
// f = { 10, ... }; // NG
f[0] = 10;
f[1] = 20;
f[2] = 30;
f[3] = 40;
```

```
char f[4];
// f[4] = "The"; // NG
// f = "The"; // NG
f[0] = 'T';
f[1] = 'h';
f[2] = 'e';
f[3] = '\0'; // 忘れそう
```

9.2 文字列操作の標準ライブラリ関数

文字列 `str` を画面に表示するには、`<stdio.h>` の `printf("%s", str)` あるいは `puts(str)` を使います。 `puts()` は `str` を表示した後に、さらに改行します。

ほかにも、`<string.h>` に次のような標準ライブラリ関数が用意されています。

```
長さ      size_t strlen(const char *s);
コピー    char *strcpy(char *dest, const char *src);
連結      char *strcat(char *dest, const char *src);
書式変換*1 int sprintf(char *str, const char *format, ...);
           int snprintf(char *str, size_t size, const char *format, ...);
比較      int strcmp(const char *s1, const char *s2);
```

見慣れない型がいくつかありますが、意味は次の通りです。

`size_t` `sizeof` 演算子の返す符号なし整数型で、標準ライブラリでは、長さやバイト数を表すときに用いられます。(実体は `unsigned int` や `unsigned long` などのいずれかが、処理系依存で選ばれています。) 本書では符号なしの型を積極的に用いませんから、`strlen()` と `int` 型変数との間で代入や比較をすることになって、厳密には `int` へのキャストが必要です。もっとも、C 言語のいい意味のいい加減さのおかげで、警告されない場面も多いです。

`char*` 「`char` へのポインタ」です。今は「`char` の配列」と同じと理解してください。

`const char*` 「`char` へのポインタ」ではありますが、配列の中身を書き換えませんが、という意味表明だと思ってください。 `const` は `constant` (不変の、一定の) の意味、つまり変数が変化しないことを示す修飾子です*2。

次節からは、標準ライブラリを実装してみます。文字列操作の理解を深めましょう。

*1 `s(n)printf()` は正確には `<stdio.h>` の関数です。

*2 `const` で修飾された変数に代入すると、通常はコンパイルエラーになるでしょうが、かつては単に `const` を読み飛ばすだけのコンパイラも存在しました。

9.2.1 文字列の長さ

`strlen()` は、文字列の長さを返す標準ライブラリ関数です。この動作を理解するために、自作しておしてみたのがソースコード 9.1 です。

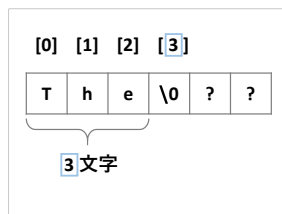
`str_length(str)` 配列の先頭から `'\0'` を探しています。 `str[i]` が `'\0'` になったら、そのときの `i` が文字列の長さになります。配列の添字が 0 始まりのおかげで、-1 のような調整項が不要で、うまく計算できます。

コラム：ヌル文字の向こうに

文字列を扱う標準ライブラリ関数は、ヌル文字より後の配列要素を参照しません。文字列としては、何が入っていても関係がありません。

ソースコード 9.1 文字列の長さ

```
1 #include <stdio.h>
2 #include <string.h> // strlen()
3
4 /* strlen()の自作関数：文字数を数える */
5 int str_length(char str[]) {
6     int i; // ループ後にも必要なので、ここで定義
7     for (i=0; str[i]!='\0'; i++) {
8         // dummy
9     }
10    return i;
11 }
12
13 void test_str_length(char s[]) {
14     printf("%s', %d, %d\n",
15           s, (int)strlen(s), str_length(s));
16 } // (int) でキャストする代わりに %zu で表示してもよい(C99)
17
18 int main(void) {
19     char str[] = "The";
20     printf("%d\n", (int)sizeof(str)); // 4 ('\0'を含むバイト数)
21     test_str_length(str);           // 3 (文字列の長さ)
22
23     test_str_length("The"); // 3 (文字列リテラルを引数に渡せる)
24     test_str_length(""); // 0 (長さ0の空文字列)
25     test_str_length("あ"); // 3(UTF-8の場合) または 2(SJISなど)
26     return 0;
27 }
```



`test_str.length(s)` 標準ライブラリの `strlen(s)` と、自作の `str.length(s)` について、比較するために両方の値を表示しています。同じ値が表示されることを確認しましょう。`strlen()` の型は `size_t` ですので、`printf()` で表示するには工夫が必要です。C99 で新設された、`size_t` 用の "%zu" という書式文字列もありますが、`int` にキャストしてしまえば、使い慣れた "%d" で大丈夫です。

`main()` 文字列 "The" の占めるバイト数 (=4) を表示してから、"The" をはじめとする何通りかの文字列に対して `test_str.length()` を呼び出しています。まずは `sizeof` と `strlen()` の食い違いに注意してください。

`test_str.length()` の引数は `char` 配列ですが、23–25 行目のように、文字列リテラルを直接与えても大丈夫です。ここからわかるように、文字列リテラルの型は `char` 配列 (正確には `char` へのポインタ) です。変数に結び付けなくても `char` 配列を作り出す、C 言語に昔からある、特別な文法です。

空文字列 (empty string) (長さが 0 文字の文字列) のような極端な例は、間違いを見つけやすいので、積極的に試しておくべきです。ここでは 24 行目で実施しています。境界値分析 (70 ページのコラム) にも通じる手法です。

頻出ミス

`strlen(str)` は、`sizeof(str)` と同じと思いがちですが、`'\0'` を含めずに数えるので、1 小さくなります。コピー先の領域確保の際には、要注意です。

コラム：マルチバイト文字の入った文字列リテラル

ダブルコーテーション (") に囲まれた文字列リテラルに、マルチバイト文字を含めた場合、必要な配列要素数 (バイト数) がいくつになるかは、文字エンコードによって変化するので、「`char a[]="あいうえお";`」というような場面で、要素数を省略するのは理にかなってません。

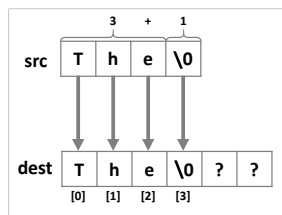
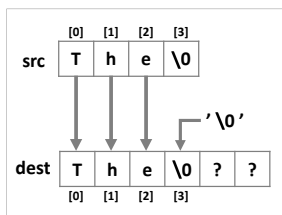
9.2.2 文字列のコピー

`strcpy()` は、文字列をコピーする標準ライブラリ関数です。やはり動作を理解するために、自作しなおしてみたのがソースコード 9.2 です。

`str_copy(dest, src)` ソースコード 9.1 の `str_length()` と同じ for ループで、1文字ずつ `src[]` から `dest[]` にコピーします。src は source (源泉、情報源)、dest は destination (目的地) の省略形で、それぞれコピー元とコピー先を表す変数名としてよく使われます。ループ終了後に、`dest[]` の末尾に終端のヌル文字を書き込みます。ループ変数 `i` は、最後の `dest[i]` への代入後にも 1 増えているので、ループ後のヌル文字の代入 (10 行目) で `i` に +1 のような調整項が不要と、うまくできています。ヌル文字忘れは、このあとのコラムで説明するように、大事故につながります。

ソースコード 9.2 文字列をコピー

```
1 #include <stdio.h>
2 #include <string.h> // strlen(), strcpy()
3
4 /* strcpy() の自作関数：文字列をコピーする */
5 void str_copy(char dest[], char src[]) {
6     int i; // ループ後にも必要
7     for (i=0; src[i]!='\0'; i++) {
8         dest[i] = src[i];
9     }
10    dest[i] = '\0'; // [i+1] ではない
11 }
12
13 /* strcpy() の自作関数：文字列をコピーする (その 2) */
14 void str_copy2(char dest[], char src[]) {
15     int len = strlen(src);
16     for (int i=0; i<len+1; i++) { // +1 は '\0' のため
17         dest[i] = src[i];
18     }
19 }
20
21 int main(void) {
22     char buff[100];
23     strcpy(buff, "abcdef"); printf("%s\n", buff); // "abcdef"
24     str_copy(buff, "ABCD"); printf("%s\n", buff); // "ABCD"
25     str_copy2(buff, "123"); puts(buff); // "123"
26     return 0; // puts() は文字列用の表示関数
27 }
```

`str_copy2(dest, src)` 同じようにコピーする関数です。こちらでは、あらかじめ `strlen()` で `src[]` の長さを調べておいて、回数の決まったループを使いました。この手段のほうが安全ですし、理解しやすいかもしれません。ただし、`src[]` を全体として2回走査するので、少し効率が落ちます。

終端のヌル文字は、ループ後に `dest[len]='\0'` と代入してもよいのですが、ループ回数を文字数よりも1回多めにしても同じ効果があります。このとき、0始まりの良いループ (☞ 5.5 節) になるよう、条件は `i <= len` よりも、イコールのない `i < len + 1` がよいでしょう。そして +1 の理由をコメントに書いておきます (16 行目)。

コラム：良いコメント

コメントには、後から読む人の役に立つことを書きましょう。ちなみに、**明日の自分は他人**なので、他人のために書いたコメントは、後から自分の役に立ちます。

コメントには、表面的な動作を説明するのではなく、このような処理をしている**理由を説明する**のが良いとされています。例えば、ソースコード 9.2 の 16 行目のように、標準から外れた動作をしている理由を書き留めておくとうよいでしょう。

コラム：ヌル文字忘れ

文字列の終端に `'\0'` を付け忘れると、恐ろしいことが起こります。

文字列 `str` を表示するのに `printf("%s", str);` とか、`puts(str);` を使っています。この `str` の終端に `'\0'` がなかったとしたら、表示する内容は、`str` の末尾から続くメモリに突入し、そのまま `'\0'` が出てくるまで進みます。別のプログラムに割り当てられている領域まで侵入すると、OS のメモリ保護機能のおかげで、プログラムが強制停止させられることもあります。

このように、`'\0'` を忘れるだけで**文字列表示さえできなくなる**ことがあります。

強制停止させられると、プログラムの間違いに気づけるので、むしろ幸運かもしれません。また、このような不正アクセスを積極的に検出するツールも存在します。

9.2.3 文字列の連結

`strcat()` は、文字列の後ろに別の文字列を連結する標準ライブラリ関数です。`cat` は、`concatenate` (連結する) の大胆な省略形です。ソースコード 9.3 に自作しなおしました。

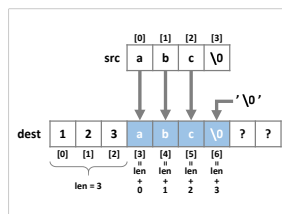
`str_concat(dest, src)` ソースコード 9.2 の `str_copy()` と同じ for ループで、1 文字ずつ `src[]` から `dest[]` にコピーします。ここでは、元々 `dest[]` に格納されていた文字列の後ろにコピーするので、元の文字列の長さを `len` に代入して、`len` だけずらしてコピーします。`dest[]` の末尾に終端のヌル文字を書き込む必要があるのも `str_copy()` と同じです。

`main()` `strcat()` の使い方にはコツがあります。まず大きめの作業領域を用意します。ここでは 100 バイト (99 文字分) の `buff[]` を用意しました。`buff[]` に格納された値は、最初は不定なので、まだ `strcat()` を使うわけにはいきません。`strcpy()` で確定させてから、`strcat()` を繰り返して連結していきます。

ただし、格納先の `buff[]` があふれないことを担保せねばならないので、実際には、`strcat()` の前に `strlen(dest)+strlen(src)+1 <= sizeof(dest)` のようなチェックが必要です。

ソースコード 9.3 文字列を連結

```
1 #include <stdio.h>
2 #include <string.h> // strcpy(), strcat()
3
4 /* strcat()の自作関数：文字列を連結する */
5 void str_concat(char dest[], char src[]) {
6     int i, len = strlen(dest);
7     for (i=0; src[i]!='\0'; i++) {
8         dest[len + i] = src[i];
9     }
10    dest[len + i] = '\0';
11 }
12
13 int main(void) {
14     char buff[100]; // buff : 不定
15     strcpy(buff, "123"); // buff : "123"
16     strcat(buff, "abc"); // buff : "123abc"
17     str_concat(buff, "ABC"); // buff : "123abcABC"
18     puts(buff);
19     return 0;
20 }
```



コラム：空文字列で初期化

`int` の変数を 0 で初期化するように、`char` 配列を長さ 0 の空文字列で初期化したいことはよくあります。方法はいくつか考えられます。

```
/* 初期化 (変数定義と同時に) */
char s[100] = ""; // 直感的
char t[100] = {'\0'};
```

```
/* 変数定義と分離した代入 */
char s[100], t[100];
strcpy(s, "");
t[0] = '\0'; // 簡便
```

ダブルコーテーション (") を 2 個連続すれば、長さ 0 の空文字列リテラルです。これを初期化子ブロックの代わりに使うのが直感的です。後から代入するなら `strcpy()` を使うのがとりあえずの作法ですが、関数呼び出しがだげさだと思えば、先頭要素にヌル文字を代入してもよいでしょう。

ソースコード 9.3 では最初に `strcpy()` を使いましたが、`buff[]` を空文字列で初期化しておけば、始めから `strcat()` でも大丈夫です。

コラム：文字列の実現方法

C 言語の採用した、ヌル文字で文字列の終端を表すという方法は、文字列を実現する唯一の方法というわけではありません。

長さど、文字列そのものを別に管理するという方法もあります。これなら文字列にどんな文字でも含められますが、逆に長さには上限ができます。

C++ の `std::string`、Perl や Python の文字列もこの方法で実現されているようです。長さを 32 ビット整数で管理していれば数 GB 分格納できるので、事実上制限なしといえるでしょう。

9.2.4 書式変換

`printf()` は、文字列用の標準ライブラリ関数というわけではなく、`<stdio.h>` で宣言されています。いつもの `printf()` は画面（コンソール）に出力しますが、`printf()` は画面の代わりに文字列に書き込みます。接頭辞の `s` は `string` の意味です。

`snprintf()` は `printf()` の変形で、書き込まれる文字列のバイト数を指定します。あふれた部分は書き込みません。最後に必ずヌル文字も付くので、安心して使えます。

```
char buff[100];
char s1[] = "123";
char s2[] = "abc";

printf(buff, "%s%s", s1, s2); // buff : "123abc"
snprintf(buff, sizeof(buff), "%s%s", s1, s2); // buff : "123abc"
```

`snprintf()`^{*3}なら、単純な記述で配列サイズを確実に守れますので、大いに活用すべきでしょう。`strcpy()`などのほうが効率はいいでしょうが、危ない橋を渡るのと同じことで、少し油断すると、配列サイズをオーバーしてしまいます^{*4}。

コラム：サイズ上限付き文字列

`<stdio.h>` に新しく加わった `snprintf()` は、書き込む文字列領域のサイズが指定できて、その文字列の終端には必ずヌル文字を書き込んでくれます。

`<string.h>` に古くからある関数の中にも、`strncpy()` や `strncat()` のように、やはり文字列領域のサイズが指定できる関数があります。しかしどうやらこれらは、終端を表すのにヌル文字だけに頼らず、サイズ上限も終端を示すという、少し変わった文字列のデータ構造を前提としているようです。つまり、領域いっぱいの文字列を作ると、終端のヌル文字が抜け落ちる関数が混ざっています。

このデータ構造は、かつてよく使われた、固定長のデータを扱うのには役に立つのですが、画面表示には `puts()` が使えず、`printf()` に文字数の最大値を指定する必要があって、扱いが難しいので、我々は利用しないほうがよいでしょう。

初期化子の文法にもこのデータ構造への用意があって、「`char s[2] = "12";`」のように、要素数がヌル文字分を含めなくても警告されないのが、要注意です。

^{*3} 正式に C 言語の規格に取り込まれたのは C99 からですが、POSIX という Unix 系の規格には存在していたので、よく利用されていました。

^{*4} 確かに `snprintf()` はバッファオーバーフローを防げますが、バッファが足りないまま処理を続けるとおかしな動作になります。したがって、バッファ不足を検出してエラー処理せねばなりません。POSIX 時代と C99 とで `snprintf()` の戻り値が微妙に異なるため、一筋縄では検出できないのが惜しいところです。

頻出ミス

C 言語のプログラムを作っていて、文字列を扱うのに `strcpy()` や `strcat()` を駆使して、`snprintf()` も正しく使えるようになって、やれやれと思っている人がいるかもしれません。

そのような人には申し訳ないですが、そもそも文字列操作を主体とするプログラムは、C 言語ではやめておくのが賢明です。メモリの不正アクセス検出機能がなくて、間違いに気づけないので練習にもなりません。世間を騒がすセキュリティホールを多くを、ヌル文字の付け忘れや、領域サイズの計算間違いによるバッファオーバーフローが占めていることを肝に銘じておきましょう。

文字列操作をするなら、文字列を単なるバイト列ではなく、オブジェクトとして操作する Python や Perl といったスクリプト言語で開発することを強くおすすめします。これは、くれぐれもです。

コラム：複合リテラル

関数の実引数に配列を渡す場合、char 配列なら、変数に代入することなく、文字列リテラルを直接渡せます。それでは、char 以外の配列はどうでしょうか。

```
void func(char x[]);

int main(void) {
    func("string");
    ...
}
```

```
void func2(int x[]);

int main(void) {
    func2( (int[]){1,2,3} );
    ...
}
```

C99 から、このような変数と結びつかない配列（複合リテラル (compound literal)）が使えるようになりました。ただし C++ ではサポートされていないからか、それとも要素数を計算できないからか、それほど使われている場面を見かけません。Java では「無名配列」という機能名でよく使われています。

9.2.5 文字列の比較

`strcmp()` は、文字列を辞書順 (dictionary order) で大小比較する標準ライブラリ関数です。cmp は compare (比較する) の、よくある省略形です。

辞書順というのは、文字通り辞書に現れるような順序です。アルゴリズム的に記述すると、次のようになります。まず先頭の文字を比較します。異なる文字であれば、この文字の文字コードで大小が決まります。同じ文字なら、次の文字を比較します。次の文字も同じなら、さらにその次と、繰り返します。同じ文字が続いてどちらかの文字列の末尾に達したら、文字列の長い方を大きいと判断します。同じ長さなら、等しいと判断します。

`strcmp(s1, s2)` の戻り値は、`s1` が `s2` に比べて、大きければ正の値、小さければ負の値、等しければ 0 です。文字列に限らず、比較関数の戻り値が「プラス、マイナス、0 のどれか」というのは、よくある取り決めです。+1、-1 と決めていないのは、プラスとマイナスをまとめて `return a-b;` のように処理できる場合があるからです。

```
char s1[] = "hoge";
char s2[] = "fuga";
// if (s1 < s2) { ... } // この表記では文字列比較にならない
if (strcmp(s1, s2) < 0) { ... } // 0 と比較する不等号は、上と同じ
```

この取り決めに従うと、呼び出し側では、上の例のように「`s1` と `s2`」の間で使いたい比較演算子を、「関数値と 0」の間で使えばよいことになります。上記は < の例ですが、== でも != でも同じ方法で大丈夫です。

ソースコード 9.4 に `strcmp()` の実装例^{*5}を載せておきます。6 行目で `unsigned char` にキャストしているのは、言語規格でこのように比較するよう指示があるからです。

ソースコード 9.4 文字列の比較

```
1 /* strcmp() の自作関数：文字列を比較する */
2 int str_compare(char s1[], char s2[]) {
3     int i;
4     for (i=0; s1[i]!='\0' && s2[i]!='\0'; i++) {
5         if (s1[i] != s2[i]) {
6             return (unsigned char)s1[i] - (unsigned char)s2[i];
7         }
8     }
9     if (s1[i] != '\0') return +1; // s1が長い
10    if (s2[i] != '\0') return -1; // s2が長い
11    return 0; // 等しい
12 }
```

^{*5} これを見てわかるように、1 バイトずつを文字コードで比較しているので、マルチバイト文字が混じると、同一判定は大丈夫ですが、大小判定はおかしくなる可能性があります。標準ライブラリ関数の `strcmp()` でも事情は同じです。

9.3 ポインタ

通常の変数は、割り当てたメモリを「変数名」で区別しています。一度割り当てた領域は、途中で変わることはありません。

ポインタは、CPUがメモリを管理する上で用いる「番地」を、そのまま使うものです。ポインタ変数の値(番地)を変更すると、ポインタの指し示すメモリの場所が変わります。このおかげで、柔軟な処理ができる反面、指し示すメモリに何が保管されているかを、プログラマが意識して管理する必要があります。

ポインタにも「型」があります。指し示す先のメモリに保管されている型を示します。どの型のポインタであるのかは、しっかりと認識せねばなりません。変数定義では、intへのポインタであれば「int *p;」とします。*がポインタであることを示します。

ポインタは、メモリを「変数名」ではなく「数値(番地)」で区別する手段です。この点で、ポインタと配列が似ていると思った人もいるかもしれません。そのとおり、**ポインタと配列は表裏一体**です。9.2節で char s[] と char *s が同じだと説明したように、互いに交換できる場面まであります。

9.3.1 ポインタを扱う演算子

変数のアドレスを取り出すのが**アドレス演算子 &**です。逆に、アドレスの先にある変数の値を取り出すのが**間接参照演算子 ***です。変数定義の*とは、役割が違う*⁶ので注意してください。「int *p, a;」があったとして、p = &a; (アドレスの代入) と *p = a; (値の代入) が文法的に正しいです。

```
int *p, a; // p は int* と解釈
p = &a;   // アドレスの代入
```

```
int *p, a; // *p は int と解釈
*p = a;   // 値の代入
```

ポインタを用いると、ソースコード 9.5 のように、スワップを関数で実現できます。

⁶ 作用はほぼ反対なのですが、巧妙にも、つじつまが合うように作られています。

ソースコード 9.5 スワップ関数

```

1 #include <stdio.h>
2
3 void swap(int *a, int *b) {           // a と b はintへのポインタ
4     int tmp = *a; *a = *b; *b = tmp; // *a と *b はint
5 }
6
7 int main(void) {
8     int a = 1, b = 2;
9     swap(&a, &b); // & でポインタにする (アドレスを取り出す)
10    printf("main: a=%d, b=%d\n", a, b); // a=2, b=1
11    return 0;
12 }

```

9.3.2 配列とポインタの関係

配列の `a[]` とポインタの `p` が同じアドレスを指しているとしましょう。`a[0]` と `*p` が同じ働きをします。`a[1]` には `*(p+1)` が対応します。`*` の優先順位が高いため、アドレスの加算を先にするよう、カッコが必要です。

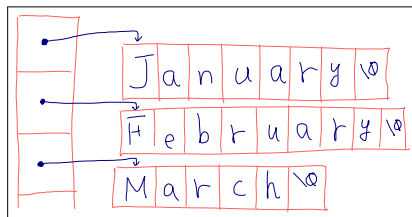
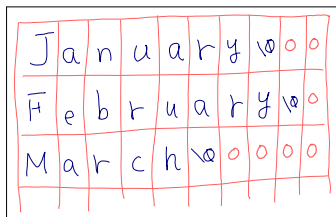
```

int a[10]; int *p = a;
a[0] <=> *p
a[1] <=> *(p+1)
a[2] <=> *(p+2)

```

9.3.3 NULL ポインタ

どのオブジェクトも指さない、特別なアドレスを表す `NULL` というポインタが用意されています。ポインタを返す関数で、エラーが起こったことを示すような場面でよく使われます (☞ 11.2.1 項)。値が `0` であることも、言語規格で保証されています。



9.4 文字列の配列

いくつかの文字列をまとめて扱うには、文字列の配列を利用したくなります。C 言語では 2 通りの実現方法があって、扱う文字列の長さが揃っているかどうかで、メモリの無駄の具合が変わってきます。

ソースコード 9.6 の `month_name` は、**char** の 2 次元配列です。12 ヶ月分、各月名の長さに関係なく、一番長い月名に合わせて 10 バイトずつの領域を割り当てているので、短い名前の後ろには使わないメモリができてしまいます。

`month_name2` は、`char*` の配列、つまり **char** へのポインタの配列です。初期化子のブロックは、先ほどとまったく同じです。ポインタの指す先には、各月名の長さちょうど文字列があります。メモリの無駄は避けられますが、ポインタ配列の領域が余分に必要なので、長さのばらつきが少ないと、どちらが節約できているのかわからなくなります。

16 行目のように、どちらの配列も使い方は同じです。C 言語の巧妙なところです。

ソースコード 9.6 月名の配列

```

1 #include <stdio.h>
2 #define N_MONTH 12
3
4 char month_name[N_MONTH][10] = {
5     "January", "February", "March", "April", "May", "June",
6     "July", "August", "September", "October", "November", "December"
7 };
8
9 char *month_name2[N_MONTH] = {
10    "January", "February", "March", "April", "May", "June",
11    "July", "August", "September", "October", "November", "December"
12 };
13
14 int main(void) {
15     for (int i=0; i<N_MONTH; i++) {
16         printf("%s %s\n", month_name[i], month_name2[i]);
17     }
18 }

```

9.5 練習問題

1. [可搬性]

会社員 A 「strlen() の値を printf() で表示したくて、試行錯誤の末に"%u"の書式文字列なら警告されないことを突き止めた。そして以下のプログラムを作成して、手で正しく動作することも確かめた。」

```
printf("文字列の長さは%u\n", strlen("abc")); // NG
```

しかし会社はこのプログラムを採用しない。その理由を説明せよ。

2. [文字列操作]

ソースコード 9.1 を参考に、次の関数を作れ。4.5.2 項の isdigit() も参照のこと。

- int str_ndigit(char str[]) は、str[] に含まれる数字の個数を返す。

3. [文字列操作]

ソースコード 9.2 を参考に、次の関数を作れ。

- void str_copy_toupper(char dest[], char src[]) は、src[] から dest[] へ、英小文字を大文字に変換しながらコピーする。(☞ 2.5.1 項の toupper())
- void str_copy_swap_case(char dest[], char src[]) は、src[] から dest[] へ、英字の大文字と小文字を入れ替えながらコピーする。

ヒント：1 文字の大文字と小文字を入れ替える関数を作り、繰り返し呼び出せ。

4. [文字列比較]

気象庁の発表する、6 種類の特別警報と 7 種類の警報について、所属機関が休講になるかどうかを表示せよ。

警報の種類や、所属機関の規定は、Web 等を活用して調べよ*8。

【実行例】

- 「暴風雪警報」→ 休講
- 「大雪警報」→ 開講
- 「特別警報 (大雪)」→ 不明*7

5. [文字列操作]

文字列で表された数値を int に変換せよ。

ヒント：文字列の先頭から 1 文字ずつ調べ、ヌル文字が出てくるまで、「これまでの数を 10 倍して今の数を加える」を繰り返せばよい。

"1" ⇒ 0 * 10 + 1 = 1

"12" ⇒ (0 * 10 + 1) * 10 + 2 = 12

"123" ⇒ ((0 * 10 + 1) * 10 + 2) * 10 + 3 = 123

6. [ループ処理への書き換え] ☞ 12.1.1 項

*7 このような警報はありません。気象庁の用語では「大雪特別警報」です。

*8 休講の条件のみが記載されていると、記載のないものが本当に「開講」なのか、あるいは検討されてなくて「不明」なのか、大いに迷うところです。プログラムの仕様書なら、休講と開講の両方の条件を列挙して、それ以外を不明とすべきでしょう。(あるいは、事前に警報の種類をコード化しておきます。)

第 10 章

構造体

今までの変数が箱だとすると、構造体とは、いくつかの箱を詰め合わせたダンボール箱のようなものです。ここでは複数のデータを、ダンボール箱に入れるかのようにして、まとめて扱う手法を学びます。

ここにて、型の概念がますます重要になります。**変数**はプログラムで定義するものでしたが、**型**も作ることができます。変数には実体がありますが、型だけでは実体がありません。変数か型か、どちらの話かよく区別して読み進めてください。

構造体は、オブジェクト指向にもつながる重要な概念です。それでいて、難しくはありませんので、しっかりと身につけていきましょう。

キーワード

- struct
- メンバ変数
- 値渡し
- ユーザ定義型・typedef
- 型と関数名を直交させる

→オブジェクト指向

10.1 構造体

例えば `double` 型の 2 つの変数 `x, y` で平面上の座標を表したとしましょう。変数 2 個一組で 1 点を表すので、常にこのペアで使うことになります。

このような場面では、変数をひとまとめにして新たな型を作る**構造体 (structure)** が役立ちます。構造体を宣言するには `struct` というキーワードを用いて、以下のように書きます。

```
/* 文法 */
struct 構造体名 {
    型名1 メンバ名1;
    型名2 メンバ名2;
    ...
};
```

```
/* 実例 */
struct point {
    double x;
    double y;
}; // ←セミコロン忘れに注意
```

構造体には名前をつけることができます。右の例では `point` としました。ひとまとめにした変数は、構造体の**メンバ (member)** と呼ばれます。右の例では `double` 型の `x` と `y` の 2 つです。(たまたま同じ型ですので、`double x, y;` と続けて書くこともできます。)

この構造体を使って、変数を定義してみましょう。構造体は新しい**型**ですから、「`int`」のような型を書くところに「`struct point`」と書きます。

```
/* 普通の変数 */
double x, y;
x = 1.0; y = 2.0;
```

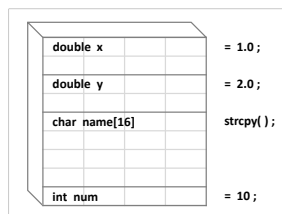
```
/* point構造体の変数 */
struct point p;
p.x = 1.0; p.y = 2.0;
```

右の例では、変数定義の後に、`p` の中のメンバ `x` と `y` に代入してみました。構造体の中のメンバを指すには、構造体の変数名に続けてピリオド (`.`) とメンバ名を書き加えます。このようにすると、左の例と同じく、メンバ変数を今までの変数と同じように扱えます。

構造体にも初期化 (変数定義と同時の代入) にだけ使える、特別な文法があります。とは言っても、1 次元配列の初期化子のブロックと同じで、代入する値をブロックで囲って書き並べます。

```
/* 普通の変数の初期化 */
double x = 1.0, y = 2.0;
```

```
/* point構造体の変数の初期化 */
struct point p = { 1.0, 2.0 };
```



頻出ミス

文字列変数への代入は、strcpy() を呼ぶ必要があったことを思い出してください。文字列リテラルの、イコールによる代入のような書式は、変数定義と同時にのみ行える特別なものでした。構造体のメンバになっても同じです。

```
/* 変数定義と分離 */
char name[100];
// name = "Taro";           // NG
strcpy(name, "Taro");       // OK

/* 変数定義と同時 */
char name2[100] = "Jiro"; // OK
```

```
struct person {
    int id;
    char name[100];
};

/* 変数定義と分離 */
struct person taro;
taro.id = 1;
strcpy(taro.name, "Taro");

/* 変数定義と同時 */
struct person jiro = {
    2, "Jiro"
};
```

10.1.1 構造体の使用例

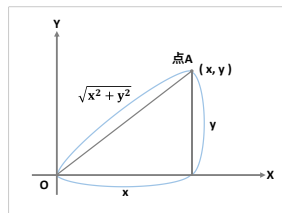
それでは、point 構造体の示す点と原点の距離を求める関数を作ってみましょう。ソースコード 10.1 のようになります。構造体の宣言（4-7 行目）は、関数定義やプロトタイプ宣言よりも前に書きます。（順序を逆にすると不可解なエラーになります。）

11 行目からの point_dist(p) は、原点から点 p までの距離を返します。16 行目からの point_print_dist(p) は、その距離を表示します。「int」と「struct point」はどちらも型ですから、どちらも同じように関数の引数に使えます。24-25 行目の main() 関数での呼び出しでも、構造体変数 p1 と p2 を普通の変数のように引数に渡しています。

ひとまずプログラムは完成しましたが、後になって機能拡張したくるときがあります。例えば点に名前（ラベル）を付けたくなったとしましょう。それには、構造体のメンバを増やせばよいです。ソースコード 10.2 の 7 行目をみてください。文字列を格納するメンバ変数 label を追加しました。16 行目からの point_print_dist() は、引数はそのま

ソースコード 10.1 原点からの距離を表示

```
1 #include <stdio.h>
2 #include <math.h>
3
4 /* 点を表す構造体 */ // 関数定義やプロトタイプ宣言よりも前に必要
5 struct point {
6     double x, y;
7 };
8
9
10 /* 原点からpまでの距離を返す */
11 double point_dist(struct point p) {
12     return sqrt(p.x*p.x + p.y*p.y); // .(ドット)でメンバを取り出す
13 }
14
15 /* 原点からpまでの距離を表示する */
16 void point_print_dist(struct point p) {
17     printf("原点と(%g,%g)の距離は%fです。\\n",
18         p.x, p.y, point_dist(p));
19 }
20
21 int main(void) {
22     struct point p1 = { 1.0, 2.0 },
23         p2 = { 9.0, 8.0 };
24     point_print_dist(p1); // 原点と(1,2)の距離は2.236068です。
25     point_print_dist(p2); // 原点と(9,8)の距離は12.041595です。
26     return 0;
27 }
```



までですが、label を表示するための処理を追加しました。22–23 行目の初期化子にも、label に対応する文字列を追加しました。

メンバが増えても 11 行目からの point_dist() は以前と同じです。24–25 行目の関数呼び出しもそのままです。もし x, y, label を別々の引数にしていたら、関数の引数をあちこち修正せねばなりません。このようにデータセットを柔軟に拡張ができるのが、構造体のありがたいところです。

ソースコード 10.2 原点からの距離を表示（ラベル付き）

```

1  #include <stdio.h>
2  #include <math.h>
3
4  /* 点を表す構造体 */
5  struct point {
6      double x, y;
7      char label[16]; // 追加
8  };
9
10 /* 原点から p までの距離を返す */
11 double point_dist(struct point p) {
12     return sqrt(p.x*p.x + p.y*p.y);
13 }
14
15 /* 原点から p までの距離を表示する */
16 void point_print_dist(struct point p) {
17     printf("原点と点%s(%g,%g)の距離は%fです。\\n", // 点%s 追加
18         p.label, p.x, p.y, point_dist(p)); // p.label 追加
19 }
20
21 int main(void) {
22     struct point p1 = { 1.0, 2.0, "A" }, // "A" 追加
23                     p2 = { 9.0, 8.0, "B" }; // "B" 追加
24     point_print_dist(p1); // 原点と点A(1,2)の距離は2.236068です。
25     point_print_dist(p2); // 原点と点B(9,8)の距離は12.041595です。
26     return 0;
27 }

```

10.1.2 構造体の代入・引数・戻り値

「int」と「struct point」が似たような役目を果たすと説明しました。それならば int の i と j で「i = j」の代入ができたように、struct point の p と q で「p = q」の代入もできることになりそうですが、本当でしょうか。

本当です。イコールひとつで、**すべてのメンバがコピー**され、同じ内容の実体ももう一つできあがります。しかも、この代入相当の動作が、(明示的には代入していない) 以下の場合にも起ります。

引数 構造体を引数に渡すことは、既にソースコード 10.1 や 10.2 で実践しました。値渡しを実現するために、構造体がコピーされます。

戻り値 struct point 型の関数を作れば、return で struct point の変数を返すことができます。このときにも、コピーが行われます。

代入は便利そうに思える機能ですが、いずれの場面でも、構造体が大きくなると非効率性が問題になります。実務的にはポインタを多用するところですが、本書の範囲を越えますので、割愛します。

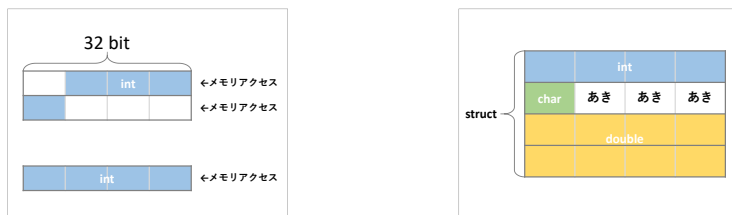
コラム：構造体の大きさの上限

文字列を代入するのに、直接の代入 (=) が使えず、わざわざ strcpy() を呼び出していました。しかしながら、構造体のメンバに固定長の文字列を含めてしまえば = で代入できてしまいます。これを利用して、構造体の中に大きな文字列の領域を作っておけば、あとは = で代入できて便利だ、と思う人がいるかもしれません。

残念ながら、構造体 (に限らずオブジェクト) の大きさに上限があります。C99 で 64KB、C89 で 32KB 以下なら動作保証がありますが、それより大きなものは、環境によって動いたり動かなかったりします。

10.1.3 構造体のスコープ

構造体 (変数ではなく、型の) スコープは、宣言したブロックに制限されます。構造体は、関数の引数などプログラム全体にわたって使いたいので、スコープを制限する意味はあまりありません。トップレベル (関数の外)、しかも関数のプロトタイプ宣言よりも前で構造体を宣言します。ソースコード 10.1 や 10.2 でも、そのようにしています。



10.1.4 構造体のアライメント・パディング（発展的内容）

構造体の大きさは、含まれているメンバ変数の大きさの合計に等しいと思いがちですが、実際にはそれよりも大きくなる場合があります。その仕組みを説明します。

まず、通常の変数を考えてみます。32ビットCPUであれば、1回のメモリアクセス（読み書き）を32ビット単位で行います。32ビットは4バイトですから、メモリの番地が4の倍数からの4バイトなら1回でアクセスできます。ところが、途中からの4バイトには2回のアクセスが必要となり、速度が低下します。これを避けるために、コンパイラは隙間を空けて、次の4の倍数の番地から変数を配置することがあります。このように、倍数に合わせて配置することを境界調整とかアライメント (alignment) といいます。

構造体のメンバも事情は同じです。例えば、1バイトのcharのあとに3バイトの隙間を空けて、次の4の倍数の番地に続きのメンバを割り当てる、といったことがよく行われています。隙間（詰め物）のメモリのことをパディング (padding) といいます。

どのようにアラインするか、いくつパディングを入れるかは、環境（CPUやコンパイラのオプションなど）によって変わります。プログラムから見れば、いくつパディングされていてもあまり影響はありませんが、メモリの使用量を見積もるなど、考慮が必要な場面もあります。実際のパディング量はsizeof演算子で確かめましょう。

```

struct int_and_char {
    int a;
    char b;
};
printf("%zu\n", sizeof(int)); // 例:4 (環境による)
printf("%zu\n", sizeof(char)); // 1
printf("%zu\n", sizeof(struct int_and_char)); // 例:8 (環境による)

```

上の例では、intが4バイト、charが1バイトで、これを合わせた構造体が8バイトですから、この構造体には $8 - (4 + 1) = 3$ バイトのパディングが含まれているとわかります。

10.2 基本データ型・ユーザ定義型

ここでは、型の種類を 2 つに大別してみます。(構造体の話題から一旦離れます。)

基本データ型 (basic data type) `int` とか `double` のような、C 言語に最初から備わっている型です。

ユーザ定義型 (user-defined data type) プログラマが作った型です。構造体はユーザ定義型の 1 種です。

`typedef` というキーワード^{*1}も、ユーザ定義型^{*2}を作ります。type definition (型定義) の意味でしょう。次のような文法で、基本データ型や構造体に別の名前をつけます。

```
/* 文法 */
typedef 既存の型名 新規の型名;
```

```
/* 実例 */
typedef int myint_t;
```

右の例のようにすると、`myint_t` が `int` とまったく同じ働きをするようになります。実質的には何も増えないのですが、ユーザが定義した、新たな型ができたように見えます^{*3}。

コラム：typedef の使い道

C 言語の `int` や `long` の記憶できる値の範囲は、CPU やコンパイラなどの環境によって異なります (☞ 2.3.1 項)。`int` に 16 ビットの領域を割り当てる場合も、32 ビットの場合もあります。

ある数値を扱うのに 32 ビットの領域が必要だとしましょう。直接 `int` を使うと、環境によっては正しく動かなくなります。そこで、例えば `int32_t` という型を使うことにして、環境に応じて以下のどちらか片方を有効にして、`int` か `long` かを選びましょう。

```
typedef long int32_t; // intが16ビットなら
//typedef int int32_t; // intが32ビットなら
```

このように、後から型を変更するのが `typedef` の典型的な使用例でした。

もっともこれは以前の話で、C99 から `int32_t` が言語規格に取り込まれ、プログラマが使い分ける必要はなくなりましたが、今でも裏方で `typedef` が働いています。

*1 マクロの `#define` と働きが似ているためか、末尾の `;` を忘れがちです。

*2 `enum`, `union` もユーザ定義型を作りますが、本書では扱いません。

*3 マクロでも近い効果は得られますが、配列やポインタを含む型では不都合があります。

10.2.1 typedef と struct の組合せ

この typedef を使うと、struct のキーワードを書く場面を減らせます。

```
struct point {
    double x, y;
};

typedef struct point point_t; // point_tが新たな型名
point_t p; // 変数定義
```

typedef の働きで「struct point」と「point_t」がまったく同じ意味になります。(型名には“_t”の接尾辞をつける習慣があります。)

さらに、次のように typedef と struct を続けて書くこともできて、しかも構造体の名前が不要になります。よいことづくめで現実にもよく用いられるので、本書でも、ここからは typedef struct を慣用句として活用^{*4}していきます。

```
typedef struct { // 構造体名(point)はもう省略してよい
    double x, y;
} point_t; // point_tが新たな型名

point_t p; // 変数定義
```

コラム：C++ の struct

struct のキーワードを書く場面を減らすための慣用句が広まったのは、なぜでしょうか。構造体を使う立場のプログラマにしてみれば、構造体であることを意識せずにすむので、ありがたいのは確かですが、理由はそれだけでしょうか。

C++ は、C 言語の上位互換のような文法になっています。C と C++ の両方で通用するようなプログラムを作る機会も多いです。その C++ では、構造体の名前がそのまま型名として使えます。つまり変数定義に、もともと struct を書かなくてよいのです。

C++ でも typedef が使えますから、この慣用句なら C と C++ を区別せずにすみます。これも大きな理由になっていそうです。

^{*4} typedef struct なら末尾の ; を忘れることもありませんので、初めに説明してもよかったかもしれません。

10.3 構造体の配列

int に配列があるように、構造体の配列も作れます。int と同様に、変数定義の変数名に続けて [] と、括弧の中に要素数を書きます。

```
/* 普通の配列 */
double x[3], y[3];
char label[3][16];

x[0] = 1.0; y[0] = 3.0;
strcpy(label[0], "A");
```

```
/* 構造体の配列 */
typedef struct {
    double x, y;
    char label[16];
} point_t;
point_t p[3];

p[0].x = 1.0; p[0].y = 3.0;
strcpy(p[0].label, "A");
```

p[0] のように、添字をつけると構造体の 1 つの変数を表すので、そのメンバを指すには、p[0].x のような表記になります。

初期化子のブロックは、2 次元配列のように 2 重になります。右下の例のように、1 つの構造体のデータが内側のブロックにまとまって好都合です。通常の配列だと、左下の例のように、値が散らばってしまいます。

```
/* 普通の配列 */
double x[3] = { 1.0, 2.0, 3.0 };
double y[3] = { 3.0, 2.0, 1.0 };
char label[3][16] = {"A", "B", "C"};
```

```
/* 構造体の配列 */
point_t p[3] = {
    { 1.0, 3.0, "A" },
    { 2.0, 2.0, "B" },
    { 3.0, 1.0, "C" },
};
```

構造体の配列も、もちろん関数の引数として渡せます。関数に渡すと、(構造体に限らず) 配列は参照渡しですから、関数内で構造体の配列を書き換えることができます^{*5}。

頻出ミス

配列の [] に書く数字は、変数定義のときは要素数で、式に現れるときは添字 (何番目) でした。要素は 0 から数え始めるので、要素数番目まで使うと使い過ぎです。

```
point_t p[10]; // 10個確保
p[0].x = 0; // OK
// p[10].x = 0; // NG
```

^{*5} 実務的には、構造体を参照渡ししたいときにはポインタを使います。本書の範囲を越えますが、構造体のポインタが多用されることは言語規格上も想定されていて、専用の文法まで用意されています。

10.4 型にまつわる命名の習慣

ユーザー定義型の名前に“_t”の接尾辞をつける習慣があることは、既に述べました。

ユーザー定義型を扱う関数の名前には、接頭辞に型の名前（もちろん“_t”を除いて）をつけるとよいでしょう。そして最初の引数にその変数を持ってくるとわかりやすいでしょう。プロトタイプ宣言は、例えば以下のようになります。

```
void point_print(point_t p);
point_t point_move(point_t p, double dx, double dy);
```

ところで、型が違って、似たような動作をする関数があります。例えば point_print() のような画面表示の関数は、どの型でも必ず作ります。同じ動作をする関数名が、型によって hoge_output() とか fuga_dispay() のように違うとややこしいので、（接頭辞の型名を除いて）統一したいところです。つまり、

型と関数名を直交させる

という方針が沸き起こります。残念ながら C 言語には、関数名を統一するような支援機能はありませんので、プログラマが意識的に揃える必要があります。

```

      ×
void point_print(point_t p);
void hoge_output(hoge_t h);
void fuga_dispay(fuga_t f);
```

```

      ○
void point_print(point_t p);
void hoge_print(hoge_t h);
void fuga_print(fuga_t f);
```

コラム：構造体からオブジェクト指向へ

型（構造体）を中心にして関数を作り始めると、関数を型ごとに分類したくなります。その分類のまま「型とそれに関数」をひとまとまりにしたものが、オブジェクト指向言語の“クラス”に相当します。つまり構造体はオブジェクト指向への足がかりになります。

オブジェクト指向言語では、もう関数に型名の接頭辞をつける必要はありません。さらに、クラス（型）が異なっても、メソッド名（≈関数名）を自然と揃えたいくなる仕組み（クラスの継承）もあります。継承のおかげで、クラス群全体を考えながらクラスを設計することになります。

10.5 例題・分数計算

分数の和と差を計算してみましょう。7章の練習問題3.では、和を受け取るのに分子と分母の2つの関数を使いましたが、構造体を用いると、どのように扱いやすくなるでしょうか。ソースコード10.3を見てみましょう。

データ型 (構造体) 5-8行目で、分数 $\frac{b}{a}$ を表す型を作ります。型名は、分数 (fraction) を省略して、frac_t にしてみます。分子、分母とも整数でよくて、メンバ名を簡単にbとaにします。分母を正にすると決めておくと、場合分けが少なくなります。

約分 (戻り値は引数を利用) 19-24行目のfrac_reduce(x)は、まず受け取ったxの分子(の絶対値)と分母の最大公約数を求めます(20行目)。xの分子と分母をそれぞれ求めた公約数で割って(21-22行目)、最後にreturnでxを返します(23行目)。つまり、受け取った引数を、加工して返しています*6。

和 (戻り値はローカル変数) 分数の和 $\frac{b}{a} + \frac{d}{c} = \frac{bc+ad}{ac}$ の計算をします。計算が複雑なので、引数を加工せず、関数の中でローカル変数を確保して代入することにします。26-32行目のfrac_add(x,y)では、ローカル変数retを用意します(28行目)。retの分子と分母に、約分前の計算結果を代入します(29-30行目)。約分するためにfrac_reduce(ret)を呼び出して、これをそのままreturnで返します*7(31行目)。

差 (和を利用) 差の計算は、符号を変えれば和の計算と同じです。34-38行目のfrac_sub(x,y)では、yの分子の符号を変えてから(36行目)、和の関数を呼び出して簡単にすませています(37行目)。

構造体を使わずに、分子と分母の2つの関数に分けていたときは、通分のための最大公約数を、別々に求めていました。31行目や37行目のように、「関数を呼び出して、戻り値をそのままreturnに使う」などという使い方もできませんでした。構造体のおかげで、便利になったことを感じてもらえたでしょうか。

ソースコード 10.3 分数の和と差

```

1 #include <stdio.h>
2 #include <stdlib.h> // abs()
3
4 /* b/a の分数を表す型 (a>0) */
5 typedef struct {
6     int b; // 分子
7     int a; // 分母
8 } frac_t;
9

```

*6 返すときには、構造体はコピーされています。引数やローカル変数は、関数を抜け出した瞬間に消滅するので、戻り値がコピーされることは重要です。

*7 ここでもやはりコピーが行われています。

```

10 /* xとyの最大公約数を返す (x,y>=0 かつ x+y>0 とする) */
11 int gcd(int x, int y) {
12     while (y != 0) { // ユークリッドの互除法
13         int t = x % y; x = y; y = t;
14     }
15     return x;
16 }
17
18 /* 約分 (x を約分して返す) */
19 frac_t frac_reduce(frac_t x) {
20     int d = gcd(abs(x.b), x.a);
21     x.b = x.b / d;
22     x.a = x.a / d;
23     return x;
24 }
25
26 /* 和 (x+y を約分してから返す) */
27 frac_t frac_add(frac_t x, frac_t y) {
28     frac_t ret;
29     ret.b = x.b * y.a + x.a * y.b;
30     ret.a = x.a * y.a;
31     return frac_reduce(ret); // 約分を利用
32 }
33
34 /* 差 (x-y を約分してから返す) */
35 frac_t frac_sub(frac_t x, frac_t y) {
36     y.b = -y.b;
37     return frac_add(x, y); // 和を利用
38 }
39
40 void frac_print(frac_t x) {
41     printf("%d/%d", x.b, x.a);
42 }
43
44 int main(void) {
45     frac_t x = { 7, 12 }; // 7/12
46     frac_t y = { 1, 4 }; // 1/4
47
48     printf("add "); frac_print(frac_add(x,y)); printf("\n"); // 5/6
49     printf("sub "); frac_print(frac_sub(x,y)); printf("\n"); // 1/3
50     printf("0 "); frac_print(frac_sub(x,x)); printf("\n"); // 0/1
51     return 0;
52 }

```

$$\frac{x_b}{x_a} = \frac{x_b/d}{x_a/d}$$

$$\frac{x_b}{x_a} + \frac{y_b}{y_a} = \frac{x_b y_a + x_a y_b}{x_a y_a}$$

もうひとつ重要なテーマがあります。frac_tにはintが2個入っているだけですから、int x[2]の配列を使えば、同じような関数は作れそうですが、それで安全でしょうか。

例えばintで“身長”と“体重”の入ったbody_t型があったとしましょう。body_tとfrac_tは明確に型が違うので、いくら中身が2個のintで同じでも、「身長」と“体重”で

分数の足し算」のような無意味なことは、コンパイルエラーでできません。これがもし、どちらも int 配列であれば、型の区別なく計算できてしまいます。

このように、構造体は変数に意味づけを与えて、無意味な操作を防いでいます。

10.6 練習問題

1. [関数の再利用]

ソースコード 10.3 では、`frac_sub()` は `frac_add()` を呼び出しているが、呼び出す代わりに `frac_add()` と同様の計算を符号違いで実装することも可能である。この 2 通りの方針を比べて、信頼性や保守性についてどんなメリット、デメリットがあるかを論ぜよ。

2. [構造体の関数渡し]

ソースコード 10.3 に、逆数を返す関数 `frac_inv(x)` を追加せよ。x の分子が 0 でないことを前提としてよいが、逆数の分母が負になれば、分子分母ともに符号を反転して、分母を正に保つようにせよ。

そして、掛け算 `frac_mul(x,y)` と、割り算 `frac_div(x,y)` の関数を追加せよ。

3. [構造体の関数渡し]

10.2.1 項の点 (2 次元座標) を表す `point_t` 型を用いて、次の関数を作れ。

- `point_t point_move(point_t p, double dx, double dy)` は、点 p を x 軸方向に dx、y 軸方向に dy だけ移動した点を返す。

4. [メンバ変数の追加]

ソースコード 10.2 に次の 2 つの改造を行なえ。(1) 10.2.1 項の `point_t` 型を用いる。(2) z 座標を追加して、3 次元座標上の点に拡張する。`main()` の p1 と p2 の z 座標は、自由に設定せよ。

5. [構造体の関数群]

2 次方程式 $ax^2 + bx + c = 0$ の係数 double a,b,c を保持する型 (構造体) `quad_t` を作り、`quad_t` 変数で 2 次方程式を表すものとする。以下の関数群を作れ。簡単のため、判別式が非負であるとしてよい。

- `double quad_det(quad_t q)` は方程式 q の判別式 ($b^2 - 4ac$) を返す。
- `double quad_large_sol(quad_t q)` は方程式 q の大きい方の解を返す。
- `double quad_small_sol(quad_t q)` は方程式 q の小さい方の解を返す。
- `quad_t quad_param(double alpha, double beta)` は α と β を解にもつ 2 次方程式 (の係数) を返す。 $(x-\alpha)(x-\beta) = x^2 - (\alpha+\beta)x + \alpha\beta$ を利用してよい。

6. [標準ライブラリの構造体を用いる] [☞ 12.1.1 項](#)

第 11 章

ファイルの読み書き

ファイルは、コンピュータ上の「データを保存する入れ物」です。テキストエディタで作っている C 言語プログラムもファイルです。写真のような画像も、表計算ソフトの文書もファイルです。

一度プログラムを終了しても、ファイルにデータを保存しておけば、引き続いて動作を続けることができます。駐車スペースごとに、駐車開始と終了の時刻をファイルに保存しておけば、後から統計をとって、1 日の売上高を計算することもできます。

我々の作るプログラムからもファイルが扱えます。扱い方には決まった手続きがあります。

そもそも、たいていの作業には、(1) 前処理 (2) 主処理 (繰り返し) (3) 後処理、という流れがあります。ファイルを扱うのにも、それぞれの手続きがありますので、ここで学んでいきましょう。

キーワード

- ファイルポインタ (fp)
- fopen / fclose
- 文字単位・行単位
- NULL, EOF
- 標準入出力

11.1 ファイルとは

ファイルは、データを保存するものです。いくつも作れるので、名前（ファイル名）をつけて区別します。データは 8 ビット (=1 バイト) 単位で保存されますので、ファイルの長さは「何バイト」と数えます。長さの上限は、通常は意識する必要はありません。

保存されるデータの形式は、一般に次の 2 通りに大別されます。

テキスト形式 そのままで画面表示ができて、文字として見ることができます。ASCII 文字のみを含むときは、ASCII 形式ともいいます。人が編集したり、アプリケーション間でデータを交換するのに向いています。(例:ソースコード、HTML、CSV など)

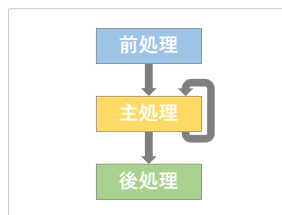
バイナリ形式 そのままでは画面表示できず、何らかの変換をせねば意味のわからないものです。コンピュータ向きです。(例:実行ファイル、画像ファイル、圧縮ファイル、アプリケーションごとの独自形式など)

我々の作るソースプログラムは、もちろんテキスト形式です。テキスト形式には、数字やアルファベットなどの ASCII 文字だけでなく、ひらがなや漢字のようなマルチバイト文字を含めることがあります。マルチバイト文字の表現方法（文字エンコード）は日本語だけでも何通り^{*1}もあって、食い違うといわゆる「文字化け」が起こります。

改行コードも OS によって異なり、`\r\n` (Windows) , `\n` (Unix 系) , `\r` (古い Mac) の 3 通りがよく使われています。ただし、C 言語の標準ライブラリが違いを吸収するので、プログラムからは気にしなくてよいことになっています。

ここでは、テキスト形式を扱いますが、マルチバイト文字にはあまり立ち入らないことにします。

^{*1} 以前は JIS (ISO-2022-JP)、ShiftJIS、EUC-JP の 3 つがよく使われていましたが、ようやく UTF-8 に統一されつつあります。



11.2 プログラムからの入出力

C のプログラムでファイルを扱うには、FILE * という型^{*2}を使います。この型の変数には fp のような名前をつけるのが習慣です。ファイルポインタ (file pointer) の意味です。

ファイルポインタを扱う関数群があります。いずれの関数名の先頭にも f がついています。FILE の意味でしょう。

作業	使う関数
前処理	fopen()
主処理 (繰り返し)	fgets(), fprintf() など
後処理	fclose()

ファイルポインタは、ファイルそのものではありません。オープンしているファイルの状態が保存されます。具体的に何が保存されるのかは、プログラマは知る必要はなく、この型を扱う関数群だけを使って操作すればよいことになっています。ファイルポインタの使い方は次の通りです。

```

FILE *fp; // 構造体へのポインタ (ファイル 1 つにつき 1 個必要)

fp = fopen(ファイル名, "r"); // 前処理
.... // 主処理
fclose(fp); // 後処理
  
```

上のように、fopen() でファイルポインタ fp を手に入れます。これを使って主処理をしてから、fclose(fp) で 1 つのファイルの処理が完了です。

^{*2} この型の実体は構造体で、しかも必ずポインタで使います。今の習慣なら file_t の名前にするところですが、何か理由があって、すべて大文字の、マクロのような名前になっています。

11.2.1 オープン（前処理）

ファイルのオープンに使う `fopen()` のプロトタイプの意味は、次のとおりです。

```
FILE *fopen(const char *path, const char *mode);
```

path どのファイルを開きたいかを指定します。単にファイル名だけならば、実行プログラム（`a.exe` や `a.out`）と同じフォルダ^{*3}のファイルになります。フォルダが異なれば、絶対パスや相対パスを使います^{*4}。

mode ファイルを読み取るのか、書き込むのかの区別をする**オープンモード**を指定します。以下の3通りがよく使われます。

mode	"r"	"w"	"a"
動作	読み取り (read)	書き込み (write)	追記 (append)
ファイルの存在する必要	あり	なし	なし

戻り値 オープンに成功すると、ファイルポインタを返します。失敗すると `NULL` という特別な値を返すので、検出してエラー処理せねばなりません。

頻出ミス

`mode` を `"w"`（書き込み）にすると、既に同名のファイルが存在する場合は、上書きして**元のファイルは消滅**しますので、注意が必要です。

`"a"`（追記）は、ファイルがあれば末尾に追記するので、消滅の心配はありません。

コラム：ファイルオープン失敗

ファイルのオープンに失敗する原因は、デバイスの破損だけではありません。

読み取り (1) 指定したファイルが存在しない (2) 存在しても読み取り権限がない

書き込み, 追記 (1) デバイスの容量不足 (2) 指定のフォルダに書き込み権限がない

(3) 書き込めないファイルが既に存在している (☞ 177 ページの頻出ミス)

^{*3} 正確には、「プログラムを実行したときのカレントディレクトリ」です。

^{*4} Windows のフォルダの区切り文字は「\」と、エスケープ文字と同じですから、プログラム中に記載するのなら、エスケープ文字自身を表すのに「\\」と2個に増やす必要があります。ただし Cygwin は、Unix 系の「/」ですので、気にしなくて大丈夫です。

表 11.1 主処理に使う主な関数とプロトタイプ

対象	読み取り	書き込み・追記
1 文字 (char)	fgetc()	fputc()
1 行 (string)	fgets()	fputs()
書式つき (format)	fscanf()	fprintf()

```

int fgetc(FILE *fp);
char *fgets(const char *buff, int size, FILE *fp);
int fscanf(FILE *fp, const char *format, ...);

int fputc(int c, FILE *fp);
int fputs(const char *s, FILE *fp);
int fprintf(FILE *fp, const char *format, ...);

```

11.2.2 読み書き（主処理）

読み書きのどちらかによって、使う関数が変わります。また、文字単位か、行単位か、どちらで処理するのかによっても変わります。まとめると表 11.1 のようになります。関数名の最後の c は char（文字）、s は string（文字列）の意味でしょう。put ↔ get の対義語も使われて、（激しく省略されていますが）規則正しい名前になっています。

ファイルポインタは、ファイル上の現在の読み取り・書き込み位置も保持しています。これらの関数を呼び出すと、この位置が後ろに進んでいきます。繰り返し呼び出すことで、複数文字あるいは複数行を処理できます。

それぞれの関数の使用例は、後ほど紹介します。

表 11.2 頻出マクロ一覧

マクロ名	記載ヘッダ	典型的な値	意味
NULL	<stdio.h>	必ず 0	ポインタの特別な値
EOF	<stdio.h>	-1	ファイルの終端 (End Of File)
EXIT_SUCCESS	<stdlib.h>	0	正常終了
EXIT_FAILURE	<stdlib.h>	1	異常終了

11.2.3 クローズ (後処理)

主処理が終わったら、`fclose()` でファイルをクローズします。書き込みファイルは、クローズすると、ようやくすべての内容が書き込まれます。プロトタイプは次の通りです。

```
int fclose(FILE *fp);
```

同時にオープンできるファイルの数には上限があるので、こまめにオープン・クローズします*5。1つのプログラムで、通常なら 2~3 個で足りるはずですが、10 個も同時に必要になったら、プログラムの設計から見直したほうがよいでしょう。

*5 OS のプロセス保護の不十分だった時代には、不正終了したプログラムのオープンしていたファイルがクローズされずに残ることがあり、繰り返すうちに OS 全体のファイルの同時オープン数を使い果たすという事態が起きました。今の OS なら保護が手厚くなっていて、少々のことならプログラムの終了と同時にクローズしてくれますが、作法としては、できれば早い段階で、明示的にクローズしましょう。

11.2.4 プログラムの中断（エラー処理）

ファイル処理と直接の関係はないのですが、ファイル処理の途中でエラーが発生した場合など、プログラムを強制的に終了したくなることがあります。そのような場合に、`exit()`を使います。

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

どの関数から `exit()` を呼び出してもプログラムを終了します。`exit()` は関数なのに、呼び出し元には戻りません。

引数の `status` に与える値として、正常終了を表す `EXIT_SUCCESS` と、異常終了を表す `EXIT_FAILURE` というマクロ定数が用意されています。`status` は `main()` の `return` で返す値と同じ働きがあります。表 11.2 のように、`EXIT_SUCCESS` は 0 に定義されているので、これまで `main()` で `return 0;` としてきたのは、正常終了を表しているのです。

コラム：main 関数の return

`main` 関数は `int` 型で、いつも `return 0;` を書いてきましたが、この値は誰が受け取るのでしょうか？

プログラムを実行したコマンドインタプリタが受け取ります。コマンドインタプリタは環境によって違います。`cmd.exe` だったり `bash` だったりいくつもの種類があり、それぞれに受け取った後の扱い方が異なります。大枠では、0-255 の値を受け取って、0 を正常終了、それ以外を異常終了とみなす、というところだけは共通しています。値によって異常の内容を表すのですが、どんな値でどんな異常を表すのか、一般的な決まりはありません。

11.3 ファイルの使用例

11.3.1 書き込み

プログラムでファイルを作ってみます。表 11.1 の書き込み用の関数を使います。

- `fprintf(fp, "...", ...)` は `printf(..., ...)` とほぼ同じ動作で、画面に表示する代わりにファイルに書き込みます。
- `fputc(c, fp)` は 1 文字を書き込みます。 `fprintf(fp, "%c", c)` と同じです。
- `fputs(s, fp)` は文字列を書き込みます。 `fprintf(fp, "%s", s)` と同じです。

これら 3 つを取り混ぜて使っても大丈夫です。 `fprintf()` はさまざまな型で使える万能選手です。特定の型専用の `fputc()` や `fputs()` は、効率的に動作しそうです。

実際のプログラムをみてみましょう。ソースコード 11.1 です。

前処理 5 行目で用意しておいたファイル名を使って、8 行目でファイルをオープンします。書き込みモードの "w" を指定します。9 行目の if でオープンに成功したかどうかを検査します。オープンに失敗したら、もう処理を続けられません。10 行目でオープンに失敗したファイル名を表示してから、11 行目の `exit(EXIT_FAILURE)`; でプログラムを中断します。

ここは `main()` なので、`return EXIT_FAILURE`; でも同じですが、エラーによる中断は呼び出された関数で行うことが多いので、`exit()` を使ってみました。

ファイル名は、8 行目の `fopen()` と 10 行目の `printf()` の 2 回使われています。ファイル名を変更しても、これらが食い違うことのないよう、変数 `filename` に代入して使っています。複数回使うものはまとめる。プログラマの鉄則です。

主処理 ファイルに書き込んでいます。 `fprintf(fp, ...)` は (引数が 1 つ多いですが) 画面表示する `printf()` と使い方が同じです。 `fputs()` は "... " の文字列リテラル、 `fputc()` は '.' の文字定数を出力するのに便利です。

後処理 24 行目でファイルをクローズして、25 行目で正常終了です。これまで `main()` では「`return 0;`」と書いてきましたが、 `EXIT_FAILURE` との対比で `EXIT_SUCCESS` を使ってみました。

プログラムを実行すると、同じフォルダに "output.txt" というファイルができるので、内容を確認しましょう。Cygwin や Unix 系のターミナルなら "cat output.txt"、Windows のコマンドプロンプトなら "TYPE output.txt" というコマンドで画面に表示されます。あるいはテキストエディタで開いても構いませんし、ファイルマネージャでダブルクリックすると、関連付けられたアプリケーションで閲覧できるでしょう。

ソースコード 11.1 九九の表をファイルに出力

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(void) {
5     char filename[] = "output.txt"; // 書き込むファイル名
6
7     /* 前処理 */
8     FILE *fp = fopen(filename, "w");
9     if (fp == NULL) {
10        printf("ファイルオープンに失敗しました '%s'\n", filename);
11        exit(EXIT_FAILURE); // プログラムを中断する (異常終了)
12    }
13
14    /* 主処理 */
15    fputs("九九の表\n", fp);
16    for (int i=1; i<=9; i++) {
17        for (int j=1; j<=9; j++) {
18            fprintf(fp, "%2d ", i*j);
19        }
20        fputc('\n', fp);
21    }
22
23    /* 後処理 */
24    fclose(fp);
25    return EXIT_SUCCESS; // 正常終了
26 }
```

頻出ミス

Windows のテキストエディタで、プログラムの出力したファイルを開覧するときの注意事項です。

テキストエディタの中には、閲覧中のファイルを書き込み禁止にするものがあります。この場合、閲覧中はプログラムがファイル出力に失敗するので、ファイルを閉じてから実行しなおす必要があります。

逆に書き込み禁止にしない場合は、プログラムをいつでも実行しなおせますが、閲覧内容に即座に反映されるわけではないので、ファイルを閉き直す必要があるでしょう。(書き換えを検知して、閉き直しを提案してくれるエディタもあります。)

11.3.2 文字単位の読み取り

ファイルから 1 文字を読み取るには `fgetc()` を使います。関数の戻り値が、`char` ではなく `int` であることに注意してください。

```
int fgetc(FILE *fp);
```

- 正常に 1 文字を読み取ると、その文字コードを返します。つまり 0 以上の値です。
- ファイルの最後まできて（あるいはエラーが起こって）、もう読み取れなくなると EOF という値を返します。EOF はファイルの終端（End Of File）を表すマクロ定数で、文字コードと区別のためにつく値（例えば-1）に定義されています（表 11.2）。

ファイルの内容をそのまま画面表示するプログラムでの使用例を、ソースコード 11.2 で見てみましょう。下線部はソースコード 11.1 と違うところです。前処理と後処理は、`fopen()` のモードが `"r"` になっただけでほぼ同じですが、主処理はすべて変わっています。

15 行 読み取った 1 文字を記憶するための変数 `c` を用意します。

16 行 `while` の条件式は複雑です。「`fgetc()` で読み取った文字を `c` に代入」と「ファイルの最後まできたらループ終了」の 2 つの動作を複合して行っています。分解して、無限ループで書き直すと右下のようになります。これでは長くなりますし、あまりにもよく使う処理なので、慣れた人は左下の書き方を慣用句として覚えています。我々も覚えてしまいましょう。

```
/* 慣用句 */
while ((c = fgetc(fp)) != EOF) {
    ...;
}
```

```
/* 無限ループで分解 */
for (;;) {
    c = fgetc(fp);
    if (c == EOF) break;
    ...;
}
```

17 行 `c` を画面に表示^{*6}します。これを繰り返して、ファイル全体を表示します。

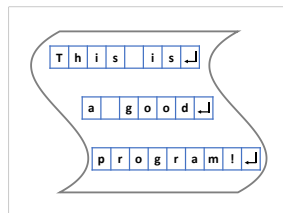
このプログラムを実行するには、先に `input.txt` というファイルを作っておきます。実行すると、このファイルの内容がそのまま表示されるはずですが。

プログラムを少し改造してみましょう。先頭に `#include <ctype.h>` を書き加え、17 行目を以下のように変更すると、大文字変換して表示するようになります^{*7}。

```
printf("%c", toupper(c)); // 17行目の差し替え
```

^{*6} (f)printf() は %c で表示する文字を int で受け取るので、変数 `c` に (unsigned char) のキャストは不要です。

^{*7} Shift_JIS などではマルチバイト文字が化けるかもしれません。UTF-8 は大丈夫です。



ソースコード 11.2 ファイルの読み取り（文字単位）

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(void) {
5     char filename[] = "input.txt"; // 読み取るファイル名
6
7     /* 前処理 */
8     FILE *fp = fopen(filename, "r");
9     if (fp == NULL) {
10        printf("ファイルオープンに失敗しました '%s'\n", filename);
11        exit(EXIT_FAILURE); // プログラムを中断する（異常終了）
12    }
13
14    /* 主処理 */
15    int c; // 読み取る1文字を保存する（char ではない）
16    while ((c = fgetc(fp)) != EOF) { // EOF はファイルの終端の目印
17        printf("%c", c); // cの1文字を画面表示
18    }
19
20    /* 後処理 */
21    fclose(fp);
22    return EXIT_SUCCESS; // 正常終了
23 }

```

頻出ミス

fgetc() は、1文字読み取ると、ファイルの読み取り位置を進めてしまいます。下の例のように、fgetc() を、EOF かどうかの比較と、1文字の読み取りとの、2回に分けて呼び出すとおかしなことになるので注意してください。

ループ中の fgetc() は 1 回だけにします。そのために、代入と同時に比較する、あの複雑な条件式の慣用句があるのです。(feof() も使えません。📖11.3.5 項)

```

while (fgetc(fp) != EOF) { // (NG) 終端チェックで読み捨てる
    c = fgetc(fp); // 1文字おきに読み取る
    ...;
}

```

11.3.3 行単位の読み取り

改行を区切としてデータを並べることがよくあります。このようなファイルは行単位で作業をしたほうが便利ですので、1 行を読み取る `fgets()` を使います。

```
char *fgets(char *buff, int size, FILE *fp);
```

buff ファイルから読み取った 1 行を保存します。1 行というのは、改行で区切られたところまでという意味で、`buff` の末尾にはたいてい `'\n'` が付きます*8。

size `buff` の大きさをバイト数で指定します。1 行がこれより長いと、いっぱいになったところで*9読み取りを打ち切ります。(残りは次の `fgets()` で読み取ります。)

戻り値 正常に読み取ったら `buff`*10を、ファイルの終端やエラーなどで読み取れなかったら `NULL` を返します。`NULL` かどうかは、必ず検査しましょう。

使用例はソースコード 11.3 です。下線部はソースコード 11.2 と違うところです。

読み取る文字列を格納するために、**行バッファ (line buffer)** と呼ばれる作業領域が必要になります。バッファの長さは事前（できたらコンパイル時）に決めておきたいので、1 行の最大の長さのものを想定しておきます。とは言っても、事前には知りようがないので、たいていは 100 文字とか 1024 文字のような適当な値にしておき、後からでも変更できるようにマクロで定義します。

4 行目で行バッファのサイズをマクロ定義し、8 行目で行バッファに使う文字列 `buff` を確保します。18 行目の `while` ループの条件式は、以前よりも簡単になりました。`fgets()` が `NULL` でなければ `buff` に読み取った文字列が入っているので、19 行目の `printf()` の `%s` で `buff` を表示します。これで 1 文字読み取りのときの動作と同等です。

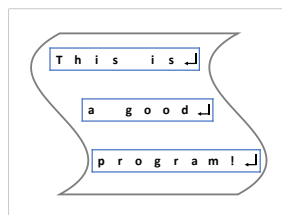
行単位で読み取るようになったので、各行に整数値の書いてあるファイルを読み取って、合計と平均を計算してみましょう。文字列から `int` への変換は `<stdlib.h>` の `atoi()` という関数*11が便利です。ソースコード 11.3 の主処理 (17-20 行) を以下で差し替えます。

*8 ファイルの末尾に `'\n'` がいないときや、1 行が `size` バイトに収まりきらなかったときを除きます。また `putc()` と同様、OS による改行コードの違いを吸収して、`'\n'` に揃えてくれることになっています。

*9 `buff` に、文字列の終端の `'\0'` は必ず付きます。`size` はこれも含んだ大きさなので、文字列の最大の長さは `(size-1)` です。

*10 戻り値の `buff` の値を利用する例は、見たことがありません。`NULL` ではない値の代表値なのでしょう。

*11 `long` への変換なら `atol()`、`double` なら `atof()` です。☞ A.6 節



ソースコード 11.3 ファイルの読み取り (行単位)

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 #define BUFF_SIZE 1024
5
6 int main(void) {
7     char filename[] = "input.txt"; // 読み取るファイル名
8     char buff[BUFF_SIZE]; // 読み取る1行を保存する行バッファ
9
10    /* 前処理 */
11    FILE *fp = fopen(filename, "r");
12    if (fp == NULL) {
13        printf("ファイルオープンに失敗しました '%s'\n", filename);
14        exit(EXIT_FAILURE); // プログラムを中断する (異常終了)
15    }
16
17    /* 主処理 */
18    while (fgets(buff, BUFF_SIZE, fp) != NULL) {
19        printf("%s", buff); // NULL はファイルの終端の目印
20    }
21
22    /* 後処理 */
23    fclose(fp);
24    return EXIT_SUCCESS; // 正常終了
25 }

```

```

/* 主処理 (差し替え) */
int count = 0, sum = 0;
while (fgets(buff, BUFF_SIZE, fp) != NULL) {
    printf("'%s' を読み取りました\n", buff); // 読み取った1行を表示
    int val = atoi(buff);
    sum += val;
    count++;
}
printf("個数=%d, 合計=%d, 平均=%f\n",
       count, sum, (double)sum/(double)count);

```

平均を知るためには、データの個数も数える必要があります。ループ 1 回につき、count に 1 を加え、sum にデータの値を加えています。ファイルを最後まで読み取ったらループが終了するので、結果を出力します。

頻出ミス

ファイルオープン 1 回につき、読み取りループは 1 つです。ファイルの終端に達すると、それ以上の読み取りはできません。上のように、データの「個数」と「合計」を同時に求めれば大丈夫です。

「個数」と「合計」を別のループで求めるなら、2 つめのループの前に、ファイルを先頭から読み直すためにクローズして再びオープンする必要があります。配列とは状況が違うわけです。世の中には、再びオープンできないファイルもあるので、ひとまずはループ 1 つですませる努力をしてみましょう。

入力ファイルの例

```
1
3
5
7
9
2
4
6
8
10
```

出力結果の例

```
'1
'を読み取りました
'3
'を読み取りました
~(中略)~
'を読み取りました
'10
'を読み取りました
個数=10, 合計=55, 平均=5.500000
```

読み取った 1 行をシングルクオート (') で囲って表示したので、1 行の末尾に改行文字 '\n' のついていることがわかります。int 変換には影響はありませんが、状況によっては除去する必要があります。

頻出ミス

データ 1 個につき改行は 1 個です。input.txt のファイルの末尾で改行を繰り返すと、改行だけでも 1 つのデータ (数値なら 0) があるとして扱われます。

テキストエディタには、たいいてい改行文字を視覚的に表示する機能があります。ファイルの末尾に「EOF」のようなマークを表示する場合があります。設定をよく見直して、活用しましょう。

11.3.4 書式つきの読み取り

表 11.1 を見ると、読み取りには書式つきの関数 `fscanf()` がまだ紹介せずに残っていますが、この関数を現実に使う場面は、そう多くはありません。なぜなら、`scanf()` 系の関数は、書式の整ったものを読み取るために作られていて、想定してない文字列がやってくると、エラー処理ができないからです。実際のプログラミングの現場では、`fgets()` と `sscanf()` を組み合わせて使いますので、`sscanf()` を紹介します。

```
int sscanf(const char *str, const char *format, ...);
```

`sscanf()` は、与えられた文字列 `str` から `format` に従って値を読み取り、... に列挙された変数にその値を保存します。`fgets()` で読み取った 1 行の中から、整数や小数や文字列が混在していても、1 回の `sscanf()` で複数の値を読み取れます。`scanf()` はキーボードから値を読み取りますが、その文字列版といったところです。13.8.2 項も参照してください。

なぜ直接 `fscanf()` で読み取ると不都合があるかを説明します。例えば、`fscanf(fp, "%d,%d", &a, &b);` で 2 つの値を読み取るとします。すると `fscanf()` は、ファイルから 1 つめの値を読み取った後に、"," が出てくるまで、それ以外の文字を読み飛ばします。もし"," のないファイルを読み込んでいると、スペースと改行を同一視して、ファイルの末尾までどんどん読み飛ばします。これが 1 回の `fscanf()` で起こるので、プログラムでは途中で止められません。つまり、"," がないというエラーを報告することができません。

`fgets()` と `sscanf()` を組合せると、少なくとも 1 行で止まるので、ファイルの末尾までは影響を受けなくて済みます。

11.3.5 終端検査、エラー検査

あまり使わない関数ですが、使い方を間違えそうなので紹介しておきます。

ファイルの読み取りに失敗した場合、ファイルの終端なのか、エラーが起こったのかを見分ける手段が用意されています。

```
int feof(FILE *stream);
int ferror(FILE *stream);
```

`feof()` はファイルの終端で 0 以外の値を返します。`ferror()` はエラー発生で 0 以外の値を返します。いずれも、`fgetc()` など読み取りに失敗した後に有効な値を返します。つまり、どちらも 0 であったとしても、**次の読み取りが成功する保証はありません**。あくまでも読み取りに失敗した後に、その理由を調べるためのものです。

頻出ミス

`feof()` と `ferror()` を `while` の条件にするには無理があります。`fgetc()` などでの終端検出は依然として必要なので、これなら無限ループにしても同じです。

```
while (!feof(fp) && !ferror(fp)) {
    int c = fgetc(fp);
    if (c == EOF) break; // この検査は省略できません
    ...
}
```


11.3.6 2つのファイルを同時に読み書き

読み取りと書き込みを同時に行ってみましょう。ファイル1つにつきファイルポインタ1つが必要なので、変数を2つ用意するところに注意してください。変数の名前はfpに限らず何でもよいので、2つの変数名を考えます。

ソースコード 11.4 ではファイルポインタを `fpin` と `fpout` としてみました。前処理で2つのファイルをオープンして、後処理では2つとも閉じるのを忘れないようにします。

ソースコード 11.4 ファイルのコピー

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 #define BUFF_SIZE 1024
5
6 int main(void) {
7     char infile[] = "data.txt";
8     char outfile[] = "output.txt";
9     char buff[BUFF_SIZE];
10
11     /* 前処理 */
12     FILE *fpin = fopen(infile, "r");
13     if (fpin == NULL) {
14         printf("ファイルオープンに失敗しました '%s'\n", infile);
15         exit(EXIT_FAILURE);
16     }
17     FILE *fpout = fopen(outfile, "w");
18     if (fpout == NULL) {
19         printf("ファイルオープンに失敗しました '%s'\n", outfile);
20         exit(EXIT_FAILURE);
21     }
22
23     /* 主処理 */
24     while (fgets(buff, BUFF_SIZE, fpin) != NULL) {
25         int val = atoi(buff);
26         fprintf(fpout, "%dのデータがありました\n", val);
27     }
28
29     /* 後処理 */
30     fclose(fpin);
31     fclose(fpout);
32     return EXIT_SUCCESS; // 正常終了
33 }
```

11.4 オープンするファイルを実行時に決める

これまでのプログラムでは、取り扱うファイル名をプログラム中に埋め込んでいました。(このような手法を**ハードコーディング (hard coding)** といいます。) これでは対象のファイルを変更するのに再コンパイルが必要なので、とても不便です。もちろん、キーボードからファイル名を受け取るように改造することも簡単です。しかし、それでも使い勝手はよくありません。なぜなら、プログラムの作成中には、繰り返し実行して動作を確かめたいので、そのたびにファイル名を手入力する必要があるからです。

取り扱うファイルをプログラムに指示することは、多くのプログラムに共通する操作なので、いろいろ工夫されています。

まずコマンド (a.exe や a.out) の後ろに書いた文字 (**コマンドライン引数** といいます) をプログラムで受け取りましょう。main() の引数を void ではなく、以下のように書きます。

```
int main(int argc, char *argv[]) {
    if (argc != 2) {
        printf("使い方: %s filename\n", argv[0]);
        exit(EXIT_FAILURE);
    }
    char *file = argv[1];
    ...
}
```

このプログラムを `./a.exe input.txt` として実行すると、main() 関数が `argc=2`, `argv[0]="./a.exe"`, `argv[1]="input.txt"` となった状態で呼び出されて、上のような処理で `file` にファイル名が手に入ります。input.txt をつけずに実行すると `argc=1` となって、最初の if が成立して、使い方のメッセージを表示します。argv[0] には常にコマンド名が入り、argc がコマンドライン引数の個数 +1 になるというわけです。

ここまで来れば、あとはコマンド実行時にファイル名をうまく入力できれば楽になります。同じファイル名でよければ、シェルの履歴 (history) 機能で十分です。カーソルキーの **(↑)** を押すと、1 つ前に入力したコマンドが、引数を含めて表示されるはずですが、初めて入力するときでも、補完入力 (ファイル名の途中まで入力して **(Tab)** を何度か押す) を使えば、存在するファイル名を間違いなく入力できます。

ファイルマネージャとターミナルの連携機能で、ファイルのアイコンをターミナルにドロップすると、そのファイルのフルパス名が入力されるようになっていることが多いです。これは強力な機能です。

eclipse や Visual Studio のような統合開発環境でも、コマンドライン引数を指定することはできるのですが、残念ながらあまり使い勝手はよくありません。

11.5 現実的なファイルオープン

これまでオープン時の煩雑なエラー処理を何度も行ってきました。世の中のプログラマが、みんなこの処理を書いているかという点、現実は少し違います。

エラーチェック付きオープン 単純にオープン失敗時にプログラムを停止させてよいなら、以下のようなエラーチェック付きの関数がよく使われます。これを `fopen()` の代わりに呼び出すだけで、エラー処理が省けます。

```
/* ファイルオープン (NULL は 返さず exit() する) */
FILE *xfopen(char *filename, char *mode) {
    FILE *fp = fopen(filename, mode);
    if (fp == NULL) {
        fprintf(stderr, "ファイルオープン失敗 '%s'\n", filename);
        exit(EXIT_FAILURE);
    }
    return fp;
}

int main(void) {
    FILE *fpin = xfopen("input.txt", "r"); // エラー処理不要
    FILE *fpout = xfopen("output.txt", "w"); // エラー処理不要
    ...
}
```

標準入出力 標準入出力 (`stdin`, `stdout`) を使えば、プログラム開始時点でオープンされているので、エラー処理は不要です。(☞ 13.8.3 項)

出荷される製品 本格的なプログラムなら、利用者に正しいファイル名の再入力を促すなど、もっと手間のかかる処理を書くことになるでしょう*12。

*12 銀行のオンラインシステムや原子力プラントの制御プログラムのように、クリティカルな場面で使われるものだと、プログラムの大半がエラー（例外）処理になってしまうことも珍しくありません。

11.6 練習問題

1. [文字単位の読み取り]

ソースコード 11.2 を参考に、読み取ったファイルに含まれている文字数を数えてみよ。また、数字 (0 から 9 の文字) がいくつあったかを数えてみよ。同様にアルファベットの大文字や小文字も数えてみよ。(☞ 4.5.2 項)

2. [文字単位の読み取り]

ソースコード 11.2 を参考に、(‘\n’ も含めて) すべての文字についての出現頻度を数えてみよ。

3. [行単位の読み取り]

ソースコード 11.3 を参考に、読み取った各行について、以下のように、行末から逆順に表示せよ。fgets() で読み取ると、末尾に改行文字 ‘\n’ がついているので、除去してから表示せよ。

入力の例 (3行)	出力の例
12345	54321
6789 →	9876
10	01

コラム：廃止された gets()

標準入力から 1 行を読み取る gets() は、C11 で廃止されました。fgets(..., stdin) と似た動作をしていたのですが、以下の点で動作が異なります。

- 読み取る文字列の長さの上限を設定できないため、バッファオーバーフローを防げません。(これが廃止の理由です。)
- 行末の ‘\n’ を削除します。

gets() の代わりを探しても、行末の改行文字を削除するのに都合のよい標準関数が見当たらず、完全に置き換えるには手間をかけるしかなさそうです。(新設された gets_s() はオプション扱いで、gcc では使えません。)

第 12 章

最終目標

時間貸し駐車場（コインパーキング）の自動清算機の動作を模倣（エミュレート）しましょう。モデルとする駐車場は、次のようなものです。

- 駐車場の入り口を通過すると、時刻を印刷したカードが発行されます。
- 駐車場から出るときには、このカードを出口の自動清算機に通します。
- 駐車した時間の長さによって駐車料金が決定され、硬貨や千円札などで支払うと、（バーが上がるなどして）場外に出られます。

時刻は 24 時間制で扱うことにします。簡単のため、24 時を越えて駐車することは考えず、1 日のうちに必ず清算することになります。駐車料金は「10 分まで 100 円」を繰り返して適用します。（つまり 11 分は 200 円です。）支払いに使えるお金は、1000 円、500 円、100 円の 3 通りにします。

1 日あたりの料金の上限などはひとまず考えませんが、自由課題を設けるので、自由に条件を考えてみてください。駐車場によっては、出入り口にバーがなく、駐車位置のセンサーと車止めで管理しているタイプもあります。そのエミュレートをするのもよいでしょう。

12.1 規定課題

規定課題で作るプログラムの動作は、以下のように決めておきます。

1. 駐車開始時刻を尋ねます。カードの読み取りに相当する動作です。(利用者は時、分、秒を入力します。)
2. 現在時刻を取得して、経過時間を元に駐車料金 (fee) を決定します。
3. 残額 (最初は fee) を表示して、お札や硬貨を 1 枚ずつ受け付けます。(利用者は入金する金額を入力します。)
4. 入金額だけ残額を減らします。まだ残額があれば 3. に戻ります。
5. fee を越えて入金された金額は、お釣り (change) として返却します。「x 円硬貨 y 枚」のように表示します。

実行例を載せておきますので、動作を頭に入れておいてください。赤の斜体の文字は、キーボードから入力した文字です。

ソースコード 12.1 の実行例

```
駐車開始時刻を入力してください
時 => 10
分 => 0
秒 => 0
現在時刻は 10:53:31 です
経過時間は 3211 秒です。
料金は 600 円です。

残額は 600 円です。
お金を投入してください(1000, 500, 100円のみ) => 10
お金を投入してください(1000, 500, 100円のみ) => 100
残額は 500 円です。
お金を投入してください(1000, 500, 100円のみ) => 100
残額は 400 円です。
お金を投入してください(1000, 500, 100円のみ) => 1000
おつりは 600 円です。

500 円硬貨 1 枚
100 円硬貨 1 枚
```

12.1.1 部品の作成

各章の学習が終わった時点で、以下で指示する部品となる関数を作ってください。これらの関数は、最後に結合してプログラムを完成させるので、今の段階では単独での動作を確実にしておきましょう。

3章

- `int sec2fee(int sec)`

`sec` 秒駐車したときの駐車料金を返します。料金は 10 分までごとに 100 円とします。

10 分 (600 秒) の割り算を切り上げで行なう必要があります。しかし、C 言語の正の整数の割り算は切り捨てです。double で計算して int にキャストしても、小数以下は切り捨てられます。

小数から整数への切り上げには、数学関数の `ceil()` があります。あるいは、整数演算で a/n の切り上げの割り算を行うには $(a+n-1)/n$ の慣用句を用います。n-1 の意味は、2 章の練習問題の 7. を思い出して考えてください。

6章

- `int is_coin_note(int value)`

`value` が受け取り可能なお札や硬貨の金額であれば TRUE を、そうでなければ FALSE を返します。

`value` が 1000, 500, 100 のいずれかの場合に TRUE を返します。

- `int get_coin_note(void)`

キーボードから金額を受け取り、受け取り可能なお札や硬貨の金額であれば、その値を返します。そうでなければ、再びキーボードから金額を受け取り直します。

キーボードから数値を受け取るには、`scanf()` あるいはソースコード 13.4 の `input_int()` を使えばいいでしょう。受け取り可能かどうかの判定には、`is_coin_note()` を呼び出します。

7 章

- `int sec_diff(int h1, int m1, int s1, int h2, int m2, int s2)`
h1 時 m1 分 s1 秒から h2 時 m2 分 s2 秒までの、経過秒数を返します。

時分秒のように、60 倍ずつの位取りのある数値を扱うと、

- 加算では繰り上がり
- 減算では繰り下がり

が起り、処理が複雑になります。

そこで、「h 時 m 分 s 秒」を、「0 時 0 分 0 秒」を起点とする経過秒数に変換し、その秒数で加減算を行う、という手法がよく用いられます^a。こうすれば、繰り上がりや繰り下がりが起らず、処理が単純になります。

つまり、`sec_diff()` を実現するために、次の関数を補助的に作るとよいでしょう。

- `int hms2sec(int h, int m, int s)`
0 時 0 分 0 秒を起点とする、h 時 m 分 s 秒までの経過秒数を返します。
- `void print_sec2hms(int sec)`
0 時 0 分 0 秒から sec 秒経過した時刻を「h 時 m 分 s 秒」の形式で表示します。

^a UNIX の時刻は、1970 年 1 月 1 日 0 時 0 分 0 秒を起点とする、経過秒数で管理されています。

9 章

- `void put_change(int x)`
x 円のお釣りを返すのに、500 円硬貨と 100 円硬貨を組み合わせ、「a 円硬貨 b 枚」の形式で表示します。硬貨の合計枚数が最小になる払い方にしてください。

高額のものから枚数を決めていくことを含めて、4 章の練習問題 7. とほぼ同じです。今は配列の知識があるので、以前のプログラムを書き換えてみましょう。金額違いで同じ処理を繰り替えているところを、同じ部分をループ処理に、異なる部分を配列変数にしましょう。これなら、硬貨の種類が変わっても、配列を修正するだけですみます。

10 章

• `int decide_fee(void)`

キーボードから対話的に駐車開始時刻を受け取り、以下の `get_current_time()` で現在時刻を取得し、経過時間から駐車料金を決定して返します。

`struct tm` は `<time.h>` で定義されていて、メンバーの `tm_hour` に時 (0-23), `tm_min` に分 (0-59), `tm_sec` に秒 (0-59) が入ります。使い方はソースコード 12.1 も参考にしてください。

```
#include <time.h>
struct tm get_current_time(void) {
    time_t t = time(NULL);
    return *localtime(&t);
}
```

駐車料金の算出には、以前に作った `sec2fee()` などを活用します。

12 章

• `int receive_fee(int fee)`

残額 (`fee`) を表示し、入金する金額をキーボードから対話的に受け取り、その金額だけ `fee` を減らします。これを `fee` が 0 より大きい間繰り返します。`fee` が 0 以下になったら、お釣りの金額を返します。

お金を受け取るために、6 章で作った `get_coin_note()` を呼び出すと、自動的に金額が限定されます。返すお釣りの値は、正の値になるように気をつけましょう。

12.1.2 部品の結合

最後に、これまで作ってきた部品の関数と、次の `main()` 関数を 1 つのファイルに収めて、1 本のプログラムを完成させましょう。

```
int main(void) {
    int fee    = decide_fee();    // 駐車料金を決定する
    int change = receive_fee(fee); // 料金を受け取る
    put_change(change);         // お釣りを返す
    return 0;
}
```

全体の概形をソースコード 12.1 に載せておきます。必要なヘッダファイルなども参考にしてください。

ソースコード 12.1 駐車場の自動清算機 (概形)

```
1 #include <stdio.h> // BUFSIZ
2 #include <stdlib.h> // atoi(), exit()
3 #include <time.h> // time(), struct tm
4
5 #define FALSE 0
6 #define TRUE 1
7
8 /* 13章 さらなる成長に向けて */
9 int input_int(void) {
10     char buff[BUFSIZ];
11     if (fgets(buff, BUFSIZ, stdin) == NULL) exit(EXIT_FAILURE);
12     return atoi(buff); // 文字列→int変換
13 }
14
15 /* 3章 関数 (1) */
16 int sec2fee(int sec) { // 10分ごとに100円
17     /* ここを作る */
18 }
19
20 /* 6章 繰り返し処理 (2) */
21 int is_coin_note(int value) {
22     /* ここを作る */
23 }
24
25 /* 6章 繰り返し処理 (2) */
26 int get_coin_note(void) {
27     /* ここを作る */
28 }
29
30 /* 7章 関数 (2) */
31 int hms2sec(int h, int m, int s) {
32     /* ここを作る */
33 }
34
35 /* 7章 関数 (2) */
36 int sec_diff(int h1, int m1, int s1, int h2, int m2, int s2) {
37     /* ここを作る */
38 }
39
40 /* 9章 文字列とポインタ */
41 void put_change(int x) {
42     printf("おつりは %d 円です。 \n\n", x);
43     /* ここを作る */
44 }
```

```
45
46 /* 10章 構造体 */
47 struct tm get_current_time(void) {
48     time_t t = time(NULL);
49     return *localtime(&t);
50 }
51
52 /* 10章 構造体 */
53 int decide_fee(void) {
54     printf("駐車開始時刻を入力してください\n");
55     printf("時 => "); int h = input_int();
56     printf("分 => "); int m = input_int();
57     printf("秒 => "); int s = input_int();
58
59     struct tm now = get_current_time();
60     printf("現在時刻は %d:%d:%d です\n",
61         now.tm_hour, now.tm_min, now.tm_sec);
62
63     int sec = 0;
64     /* ここを作る */
65     printf("経過時間は %d 秒です。 \n", sec);
66
67     int fee = 0;
68     /* ここを作る */
69     printf("駐車料金は %d 円です。 \n\n", fee);
70     return fee;
71 }
72
73 /* 12章 最終目標 */
74 int receive_fee(int fee) {
75     /* ここを作る */
76 }
77
78 int main(void) {
79     int fee = decide_fee(); // 駐車料金を決定する
80     int change = receive_fee(fee); // 料金を受け取る
81     put_change(change); // お釣りを返す
82     return 0;
83 }
```

12.2 自由課題

自分で条件を自由に設定し、機能拡張してみてください。以下に、機能拡張のヒントを挙げておきます。

- 扱うお札や硬貨の種類を変更する
- `is_coin_note()` や `get_coin_note()`, `put_change()` の金額の種類を、内部で統合する（グローバル変数の配列を作る）
- 現在時刻をコマンドライン引数で指定できるようにする（デバッグに有用）
- 駐車料金を工夫する
 - 単価を変更する（マクロで定義するとよい）
 - 駐車開始直後に無料の時間帯を設ける
 - 1日あたりの上限金額を設ける
 - 時間帯によって（夜間料金のように）単価を変更する
 - 日付をまたぐことを考慮する^{*1}
- 駐車場所ごとに入庫時刻を管理し、出庫時には駐車場所の番号を指定する
 - 駐車場所ごとの入庫時刻をファイルから読み込む

コラム：実装の意図

時刻を秒単位で管理したのは、過剰スペックかもしれませんが、システムの時刻も秒単位ですから、（自作の）`sec_diff()` を（標準ライブラリの）`difftime()` で置き換えるのに好都合です。

`put_change()` という関数名を `print_change()` にしなかったのは、単に画面表示しているのではなく、機器に払い出すよう指示している気分を出すためです。

^{*1} `struct tm` で表した時刻を `mktime()` で `time_t` 型の通算秒数に戻し、経過秒数は `difftime()` で求めるとよいでしょう。いずれも `<time.h>` の関数です。

第 13 章

さらなる成長に向けて

この章では、C 言語に限らず、コンピュータ言語に精通するための手法や技術、コンピュータ上の道具の使い方のヒントなどを列挙してみます。

章の途中からは、少しテクニカルな C 言語の性質を説明します。理解できると有用なのですが、幅広い知識が必要なため、必要に応じて参照してもらえば十分です。

コラム：クマさんに相談

思うようにプログラムが動かず、原因がわからなくて頭を抱えているときには、誰かに相談したくなります。ある研究室では、部屋の片隅にクマのぬいぐるみが置いてあって、まずクマさんに相談することになっています。それでも解決しなければ、ようやく研究室のメンバーに相談します。



変なルールのようなのですが、クマさんに(?)説明を始めると、おかしなところに気づいて自己解決することが頻繁にあるのです。身に覚えのある人もいるでしょう、状況を最初から順序立てて説明すると、それだけで物事が見えてきます。

13.1 プログラミング環境の上手な操作

以下のような操作は、作業効率を向上させます。少し意識して使い始めると、自然と使えるようになるので、試してみてください。

- ウィンドウ操作
 - ウィンドウを整然と並べる（見たいところが同時に見えるように）
 - フォーカスウィンドウの切り替え：**Alt+Tab**
- シェルやターミナルの操作
 - コマンド履歴の呼び出し：**↑ ↓**
 - 入力を補完（入力の途中で）：**Tab**
 - ファイル名のフルパス入力：アイコンのドラッグ
- テキストエディタの基本操作（[108](#) ページのコラム）
 - カット：**Ctrl+X**
 - コピー：**Ctrl+C** または **Ctrl+Insert**
 - ペースト：**Ctrl+V** または **Shift+Insert**
 - 行頭・行末ジャンプ：**Home End**
 - キーによる選択：**Shift+↓** **Shift+→** など
 - マウスによる選択：ダブルクリック（単語単位）、トリプルクリック（行単位）
 - 文字列検索、置換
- テキストエディタの効率的な使い方
 - 手入力を避ける（動作実績のあるコードをコピーする、補完入力する）
 - 対にして先に入力する
 - (i) **" ... "** (ii) **{ ... }** (iii) **(...)** (iv) **/* ... */**
 - 整形（インデント自動調整など）や予約語のハイライトなどの支援機能

13.2 効率のよいデバッグ術

- コンパイル時のエラーや警告は、最初を見る (☞5 ページのコラム)
- コンパイル時の警告を増やしておく (☞219 ページの A.10 節)
- デバッグライトを活用する (☞81 ページのコラム)
- ソースコードのバックアップを作って再現性を確保する
- 挙動の違いの原因を確かめるためには、ソースコードの修正を一ヶ所に留める
- `<assert.h>` の `assert()` 診断マクロを利用する

コラム：動作不良の再現性

時々受ける質問に、「自分の作っていたプログラムが動作不良を起こしていたが、何とか解消できた。しかし、なぜだったのか気になる。」というのがあります。

様子を聞くと「ここを直したらエラーが解消した」というのですが、状況的にはそれでは解消するはずがないので、さらに聞くと「別のところも修正した」と。恐らく、そちらで解消したと思えるので、実演しようとする、「もう再現できない」。このような、歯がゆい思いをする状況が、ままあります。

原因を探ろうというのは、向上心があって良いことです。もう一歩進めて、バックアップをとるなりして、エラーを再現できると、さらに上達するでしょう。

13.3 コーディング上の良い習慣

折に触れて言及してきたことですが、あらためて列挙してみます。

- スペースング (☞6 ページの 1.3.5 項、79 ページのコラム)
 - (i) インデント (ii) `for` の; のあとのスペース (iii) コンマの後のスペースや改行
- スコープは狭く (☞42 ページのコラム、89 ページの 6.1.1 項)
- 変数の初期化は使う直前 (☞79 ページの頻出ミス)
- キャストの使用は最小限に (☞31 ページのコラム)
- 意味のある変数名・関数名 (☞116 ページの 7.3 節)
- マジックナンバーを排除 (☞122 ページの 8.2 節)
- 見慣れた処理を組み合わせる (☞77 ページの頻出ミス、85 ページの 5.5 節)
- コメントでは見慣れない動作の理由を説明する (☞145 ページのコラム)

13.4 プログラムの上達へむけて

同じ処理は 2 度書かない 同じ処理を 2 度書く（コピーする）と、修正のあった場合に、もう片方の修正を忘れがちです。

「車輪の再発明」を避ける 標準ライブラリの機能で事足りるなら、なるべくそれを使いましょう。例えば日付計算など、誰もが必要になりそうなものは、既に作られているものを探すべきです。自作して信頼性で追いつくのは至難の技です。

関数設計のヒント：モデルとビューは分離する 3.7 節のように、計算（モデル）だけの関数と、表示（ビュー）だけの関数に分離しましょう。関数が再利用しやすくなります。一体にした関数だと、表示形式を変えるのに計算部分をコピーした別の関数を作るようになって、上記の「同じ処理は 2 度書かない」を破ることもなります。

コード・リーディング 身近な人とプログラムを読み合ひましょう。技術の共有に役立ちます。とても一人では思いつかないコードを身につける、あるいは避けるべきパターンを覚える、良い機会になります。

処理系依存と可搬性（ポータビリティ） 言語規格で保証された動作であるか、あるいは処理系依存でたまたまその動作になっているかは、区別せねばなりません。保証がないと、異なる環境での動作がいつおかしくなるかわかりません。目の前のコンパイラ環境の動作から言語規格は推し量れませんので、最終的には仕様書で確かめるしかありません。経験の積み重ねも重要です。

13.5 値の変化の頻度に応じた扱い

状況に応じて変化する値と、本当に固定化されて変化しない値があります。どんな値でも変化するという前提にすると、プログラムが複雑になりすぎます。逆に変化しないと決めてしまうと、ちょっとした状況変化でプログラムが役立たなくなります。

右には、変化の度合いに応じたプログラム上の実現方法を挙げました。再コンパイルすれば変えられる、実行しなおせば変えられる、実行中にも変化するなど、いくつもの手段があります。扱う値の性質を見抜いて、適切な手段を選びましょう。

↑ 変化の頻度少ない

- ハードコーディング
- マクロ定数, const グローバル変数
- コマンドライン引数
- キー入力（標準入力）
- 関数の const 引数, const ローカル変数
- グローバル変数
- 関数の引数, ローカル変数
- ループ変数

↓ 変化の頻度多い

13.6 開発のための技術・ツール

プログラムの規模が大きくなると、以下のような手法やツールが重要になってきます。Unix 系でよく使われるコマンド名も挙げておきます。

テスト 単体テストは 113 ページのコラムで紹介しました。本来はテストを自動化します。境界値分析 (☞ 70 ページのコラム) など、テストの方針も提唱されています。

分割コンパイル ソースコードを複数のファイルに分割して開発する手法です。

ビルドツール 複数のソースファイルを、並列にコンパイルしたり、再コンパイルに不要なものを見分けたりして、短時間で実行ファイルを生成するツールです。例: `make`

デバッガ プログラムの実行時に、変数の値を確認しながら、逐次実行するツールです。他のツールと組み合わせることもある、開発基盤です。例: `gdb`

メモリーク検出 プログラムの実行時に、配列あふれなどのメモリー破壊を検出するツールがあります。方式がいくつかあり、万能のものは存在しません。例: `valgrind`

プロファイラ プログラムの実行時に、関数の呼び出された回数などを記録しておいて、後から集計するツールです。プログラムの高速化に役立ちます。例: `gprof`

静的解析 文法チェッカなど、ソースコードを検査するツールがあります。コンパイラに詳細な警告を出力させるのも、その第一歩です。例: `lint`

テキスト差分 2つのテキストファイルの違う部分のみを抽出するツールがあります。ソースコードはもちろん、実行結果の違いを検出するのも有用です。例: `diff`

バージョン管理 ソースコード（に限らず、テキストファイル）の世代管理を行なうツールがあります。差分を表示したり、複数人で共同開発する際の集中管理的な役目もあります。プログラマに必須です。例: `git`, `subversion`, `cvs`

ドキュメント生成 ソースコード中に、特定形式で書いておいたコメントから、HTMLなどの形式でプログラム全体のドキュメントを生成するツールがあります。Java 言語なら標準の `javadoc`、C 言語にはオープンソースの `doxygen` が有名です。

13.7 浮動小数点型の性質

浮動小数点型は複雑な性質を持っています。内部表現の指数部と仮数部が固定長です。典型的な `double` は、図 13.1 のように、符号に 1 ビット、指数部に 11 ビット、仮数部に 52 ビットを割り当てた 2 進数表記です。

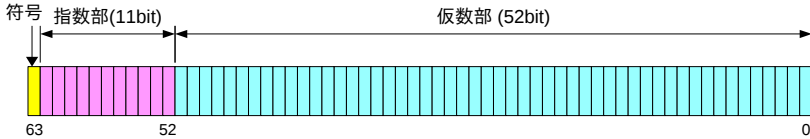


図 13.1 IEEE 754 による `double` のビット割り当て

このように精度に限られるため、数値誤差が常に付きまといまいます。誤差のない数学モデルからは想像もつかない動きの一端を紹介します。

13.7.1 丸め誤差

浮動小数点型が 2 進数で表記されていると、 $1/2 (=0.5)$ や $1/8 (=0.125)$ は正確に表せても、我々になじみのある **0.1** には誤差が入り込みます。例えばソースコード 13.1 は 0.1 を 10 回足して表示しているだけですが、想像に反して 1.0 になりません。

ソースコード 13.1 0.1 を 10 回足した結果を表示する

```

1 #include <stdio.h>
2
3 int main(void) {
4     double x = 0.1;
5     x = x + 0.1; x = x + 0.1; x = x + 0.1;
6     x = x + 0.1; x = x + 0.1; x = x + 0.1;
7     x = x + 0.1; x = x + 0.1; x = x + 0.1;
8     printf("%.17f =(?) %.17f\n", 1.0, x);
9     return 0;
10 }
```

ソースコード 13.1 の実行例 (GCC Intel 64 ビットの場合)

```
1.00000000000000000 =(?) 0.99999999999999989
```

このため、比較の際には「微小な誤差を等しいとみなす」処理を追加する必要があります。どの程度の誤差を許容すればよいかは、計算内容によって大きく左右されるため、統一的手段は確立されていません。

ソースコード 13.2 では、等しいとみなす相対誤差をマクロで定義して「if (x <= y)」の代わりに「if (fcmp(x,y) <= 0)」で大小比較できるようにしてみました。

ソースコード 13.2 浮動小数点の誤差を見込んだ比較

```

1 #include <math.h> // fabs(), fmax()
2 #include <float.h> // DBL_EPSILON
3
4 /* 等しいとみなす相対誤差 */
5 #define RELATIVE_ERROR (DBL_EPSILON * 8)
6
7 /* aとbがほぼ等しければ0, a>bなら正, a<bなら負を返す */
8 int fcmp(double x, double y) {
9     double diff = x - y;
10    double max = fmax(fabs(x), fabs(y)); // 絶対値の大きい方
11    if (fabs(diff) < max * RELATIVE_ERROR) return 0; // 許容誤差以下
12    if (diff > 0) { return +1; } else { return -1; }
13 }

```

13.7.2 情報落ち・桁落ち

有効精度が 10 進数 3 桁の状況を考えてみましょう。精度の悪くなる演算があります。

情報落ち 絶対値が極端に違う値の和や差からは、小さい方の値の情報が抜け落ちます。

例: $1.00 + 0.0123 \rightarrow 1.01$

桁落ち 値の近い数の差は、有効精度が少なくなります。

例: $1.00 - 0.999 \rightarrow 0.001$ (有効精度 1 桁)

演算の順序を工夫することで、精度の悪化を防げる場合もあります。

13.7.3 無限大・非数

浮動小数点数には、特別な値があります。

非数 (NaN, Not a Number) $0.0 / 0.0$ (数学的に不定) や、負数の平方根 (実数では計算できない) などが非数になります。printf() で「nan」のように表示されます。この値が紛れ込むと通常の四則演算ができなくなり、演算結果にも非数が伝搬します。

無限大 (infinity) 0 の割り算などで無限大になります。符号があって、 $5.0 / 0.0$ だと「inf」、 $-5.0 / 0.0$ だと「-inf」のように表示されます。

マイナスゼロ ゼロにも符号があります。 $1.0 / inf$ だと「0.00」、 $1.0 / -inf$ だと「-0.00」のように使い分けます。

13.7.4 定数

浮動小数点型には、表 2.3 以外にも多数の定数があります。その一部を紹介します。

機械エプシロン (machine epsilon) 「1 より大きい最小の数」と 1 との差です。仮数部の精度が反映されます。double の定数は DBL_EPSILON です。

正規化数 (normalized number) 仮数部を 1 以上かつ基数未満に正規化した値です。指数部で調節しますが、限界もあります。DBL_MIN は double で表せる正の最小の正規化数です。これより絶対値が小さいと正規化できず、仮数部が 1 未満の**非正規化数** (subnormal number) になって、精度が落ちます。

丸め方向 浮動小数点型に共通の FLT_ROUNDS*¹ が現在の丸め方向を、次の値で示します。

(0) 0 方向 (1) 最も近い値*² (2) +∞ (切り上げ) (3) -∞ (切り捨て) (-1) 不確定

コラム：浮動小数点演算の速度

浮動小数点演算の速度は、CPU の持つ演算回路によって大きく左右されます。

1990 年代より前、浮動小数点演算は CPU に専用の回路がなく、整数演算よりも数十～数百倍以上も遅いものでした。しかも double は float の何倍も時間がかかりました。整数演算でソフトウェアエミュレーションしていたためです。

2000 年より少し前から double の演算回路が当たり前になりました。整数演算との速度差が数倍にまで縮まり、しかも float より double のほうが少し早いという**逆転現象**まで起こりました。まだ float 用の回路はなく、double に変換して計算していたので、変換のオーバーヘッドがあったのでした。

マルチコアが当たりの今では float 用の回路もできていて、逆転現象こそ解消しているものの、float と double の速度差はそれほどありません。ちなみに整数の short と int も同じような関係で、short に速度メリットはほとんどありません。

*¹ C89 までは定数でした。C99 からは fesetround() で変更される変数に変わったはずですが、以前の実装のままになっている環境も見受けられます。

*² 等距離の値がある場合には、偶数を採用するのが一般的です。これは「偶数丸め」と呼ばれる手法です。0 から遠い方を採用すると、四捨五入になります。

13.8 入出力インタフェース

プログラムが利用者からデータを受け取る手段は、いくつかあります。もちろんファイルをオープンすれば、いくらでもデータを受け取れますが、そこまでしなくても、簡単にすませる方法があります。ファイルの出力にも、簡便な方法があります。

13.8.1 コマンドライン引数

プログラムを実行するときに、ターミナルでコマンド名 (a.exe など) に続けて書いた文字列をコマンドライン引数 (command line argument) といいます。これをプログラムで受け取るには、main() 関数の引数を void とせず、次のようにします。

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

プログラム開始時には、コマンドライン引数 (に加えてコマンド名^{*3}) が argv[] に格納され、その個数が argc でわかります。

コマンドライン引数は文字列で得られるので、int に変換するには <stdlib.h> の関数を利用して、atoi(argv[1]) のようにします。double なら atof(argv[1]) です。

```
./a.exe 123 4.56
```

↓

```
argc = 3;
argv[0] = "./a.exe"; // コマンド名
argv[1] = "123";     // 第1引数
argv[2] = "4.56";   // 第2引数
```

コマンドライン引数が必須であれば、付け忘れたときの対策として、argc が少ない時に使用法のメッセージを表示するのが習慣です。

ソースコード 13.3 コマンドライン引数を受け取る

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(int argc, char *argv[]) {
5     if (argc < 3) { // 引数2個に足りないとき
6         printf("用法: %s 整数 小数\n", argv[0]);
7         exit(EXIT_FAILURE); // 中断
8     }
9     int a = atoi(argv[1]); // 第1引数
10    double b = atof(argv[2]); // 第2引数
11    printf("%d * %f = %f\n", a, b, a*b);
12    return EXIT_SUCCESS;
13 }
```

^{*3} argv[0] を利用すると、コマンド名前で動作を切り替えるプログラムも作れます。

13.8.2 キーボード

キーボードから入力を受け付けるなら、`scanf()` が簡便ですが、問題点もあります。右の例では、変数の `a` と `b` にキーボードから入力した数値が入ります。

`scanf()` は、`printf()` と似た形式の書式文字列に従って、キーボードから受け取った値を、引数で指定されたアドレスに書き込みます。

```
printf("整数を入力してください> ");
int a;
scanf("%d", &a);

printf("小数を入力してください> ");
double b;
scanf("%lf", &b);

printf("a=%d, b=%f\n", a, b);
```

- 例えば `%d` に対応する引数には、`int` の値を書き込みます。書式の詳細は [A.8 節](#)
- `%` ではない、普通の文字列に対しては、その文字がキーボードから入力されるまで待ちます。(つまり、それ以外の文字を読み飛ばします。)
- スペース文字や改行文字は、どれも同じ (空白文字) と扱われます。

`printf()` との混同 引数をポインタにするのに `&` が必要です。さらに `double` には `%lf` と `l` ^{エル} が必要です。`printf()` とは、似ているようで違います。

プロンプト表示 プログラムを実行すると、`scanf()` の部分で入力待ちになって、停止したようにも見えるので、右上の例のように、プロンプト (入力を促すメッセージ) を表示するのがよいでしょう*4。

書式違い `printf()` と `scanf()` の `f` は `formatted`、つまり **書式の定まった** という意味です。`printf()` で出力する書式はプログラムで定めるのでいいのですが、`scanf()` は **書式の整った文字列を受け取る** ように作られています。それを人間が手入力する場面に流用しているので、書式が少し違うだけで簡単に処理できなくなります。例えば数値を受け取るはずのところアルファベットが入力されると、プログラムがそこで停止してしまいます。

書式違いの対策には、まず `fgets()` でキーボードからの入力を文字列として受け取って、その後 `atoi()` や `sscanf()` で解析します。これなら値の受け取りに失敗しても、プログラムが停止することはありません。ソースコード [13.4](#) のように、数値を受け取る処理を関数に独立させておくと、後からの差し替えにも好都合です。

*4 対話的なのでわかり易い動きと思えるのですが、何度も繰り返すと、メッセージが煩わしくもなります。ファイルにリダイレクトすることが前提なら、プロンプトは標準エラー出力にして、ファイルに残らない工夫をします。定型処理を繰り返すのであれば、コマンドライン引数のほうが向いています。

ソースコード 13.4 キーボードから数値を受け取る

```
1 #include <stdio.h>
2 #include <stdlib.h> // atoi(), atof(), BUFSIZ
3
4 int input_int(void) {
5     char buff[BUFSIZ];
6     if (fgets(buff, BUFSIZ, stdin) == NULL) exit(EXIT_FAILURE);
7     return atoi(buff); // 文字列→int変換
8 }
9
10 double input_double(void) {
11     char buff[BUFSIZ];
12     if (fgets(buff, BUFSIZ, stdin) == NULL) exit(EXIT_FAILURE);
13     return atof(buff); // 文字列→double変換
14 }
15
16 int main(void) {
17     printf("整数を入力してください> ");
18     int a = input_int();
19
20     printf("小数を入力してください> ");
21     double b = input_double();
22
23     printf("a=%d, b=%f\n", a, b);
24     return EXIT_SUCCESS;
25 }
```

13.8.3 標準入出力・リダイレクト・フィルタ

UNIXには、名前にuni-（単一の）の接頭辞がついている通り、物事を統一的に扱おうという思想があります。この影響を受けてか、C言語プログラムはキーボードもコンソール画面もファイルとして統一的に扱います。

シェル（コマンドインタプリタ）は、プログラム起動時に3つのファイルを開きます。標準入力、標準出力、標準エラー出力です。通常はキーボードやコンソール画面につながっています。C言語からは、それぞれstdin, stdout, stderrというファイルポインタが割り当てられるので、printf("123");とfprintf(stdout, "123");が同じ意味になります。

プログラムを起動するときに、普段はキーボードにつながっている標準入力を、ファイルに置き換えることができます。この操作をリダイレクト（redirect）といいます。キーボードから入力する文字列を、あらかじめ"input.txt"というファイルに記録しておけば、次のように、不等号とファイル名を指定すると、その内容を読み取ってくれます。

表 13.1 標準入出力の種類

名称	既定の接続先	FILE*	リダイレクト操作
標準入力	キーボード	stdin	command < file
標準出力	コンソール画面	stdout	command > file (上書き) command >> file (追記)
標準エラー出力	コンソール画面	stderr	command 2> file など

```
$ ./a.exe < input.txt
```

画面に表示されるメッセージをファイルに保存したければ、逆向きの不等号です。

```
$ ./a.exe > output.txt
```

この操作では、"output.txt"というファイルに記録しますが、同名のファイルが存在していると、上書きして元の内容は消えてしまいます。追記するなら>>と2個にします。また、入力と出力の両方を同時にリダイレクトしても構いません。

標準エラー出力は、エラーメッセージや、時間のかかる処理の途中経過など、ファイルにはリダイレクトされないことを前提として使われます。

2つのプログラムの標準入力と標準出力をつなげることもできます。パイプ (pipeline) でつなげる、といいます。中間ファイルを生成する必要がなくなります。

```
# 中間ファイルを生成
$ ./a.exe > tmpfile.txt
$ ./b.exe < tmpfile.txt
```

```
# パイプで連結して一括処理
$ ./a.exe | ./b.exe
```

標準入力から受け取ったデータを加工して、標準出力に結果を流すだけの単純なプログラムを**フィルタ (filter)**といいます。パイプはフィルタを組み合わせるのに役立ちます。このような、単機能のプログラムを組み合わせ、複雑な動作をすることは、UNIXの思想に合致します。

コラム：キーボードの EOF

キーボードがファイルである以上、終端を示ことができるはずですが。ターミナルエミュレータによって、この操作が違います。

Cygwin や Unix 系では、**Ctrl+D** です。状況によっては **Enter** の直後に入力する必要があります。

Windows のコマンドプロンプトでは **Ctrl+Z** です。

13.9 疑似乱数

疑似乱数 (pseudo-random number) を生成する関数が `<stdlib.h>` にあります。再現可能な疑似乱数ですが、(☞ 14 ページのコラム)、以下では簡単に「乱数」と表記します。

```
#include <stdlib.h>
定数
    RAND_MAX // 32767以上
関数
    int rand(void);
    void srand(unsigned int seed);
```

`rand()` は、0 以上 `RAND_MAX` 以下の一様乱数を返します。関数を呼び出すたびに新しい値が手に入ります。`RAND_MAX ≥ 215 - 1 = 32767` であることが言語規格で保証されています。欲しい乱数の種類に合わせて、以下のように加工して使います。

0 以上 1 未満 (double)

```
double frand(void) {
    return 1.0 / ((double)RAND_MAX + 1.0) * rand();
}
```

1 未満になるよう、`rand()` を `RAND_MAX + 1` で割りたいのですが、整数演算ではオーバーフローの危険があるので、`RAND_MAX` を `double` に変換してから 1 加えています。この範囲の乱数は、さらに加工するのに都合のよいものです^{*5}。

0 以上 6 未満 (int)

```
int a1 = rand() % 6;           // 下位ビット重視
int a2 = (int)(frand() * 6);   // 上位ビット重視
```

`a1` のように、6 の剰余をとると、目的の範囲に入ります。ただし、かつての乱数は、特に下位ビットのランダム性が悪く、偶数と奇数が交互に発生するものもあったため、上位ビットを生かす方法が良いとされていました。そのために `a2` では、まず `frand()` を 6 倍して、0 以上 6 未満の浮動小数点を得ます。そして `int` にキャストして小数を切り捨てます。これで目的の範囲の整数が得られます^{*6}。

1 以上 6 以下 (int)

```
int b1 = 1 + (rand() % 6);
int b2 = 1 + (int)(frand() * 6);
```

`a1` や `a2` にを 1 加えます。これでサイコロの目に相当するものが作れました。範囲を x 以上 y 以下にするなら、1 を「 x 」、6 を「 $y - x + 1$ 」と読み替えます。

^{*5} Java なら `Math.random()` が同じ範囲の乱数を生成します。

^{*6} もし `frand()` の範囲に 1 が含まれていれば、非常に低い確率で `a2` が 6 になったことに注意してください。

乱数を生成する際に、内部的に用いる値を**乱数の種 (seed)**と呼びます。次の乱数のための種を指定するのが `srand()` です。同じ種を `srand()` で指定すれば、その後の `rand()` で得られる乱数系列は同じです。右の例は GCC 7.5 (glibc-2.27) の場合です。

```
srand(314159); // 314159を種にする
int a = rand(); // → 414777680
int b = rand(); // → 2009630532
int c = rand(); // → 102799611
srand(314159); // 同じ種
int d = rand(); // → 414777680
int e = rand(); // → 2009630532
int f = rand(); // → 102799611
```

- `srand()` による種の指定は、通常は `main()` 関数で 1 度だけ行なえば十分です。
- `<time.h>` の `time(NULL)` を種にすると、典型的には 1 秒ごと^{*7}に異なる乱数系列になります。(☞ソースコード 1.5)
- 乱数の生成方法は言語規格では規定されていないので、発生する値は処理系に依存します。実行環境によらず同じ値が必要なら、自前で生成ルーチン^{*8}を用意します。

13.10 関数の引数の可変長

関数の中には、`printf()` や `scanf()` のように、引数がいつも同じではなく、状況によって個数や型の変わる**可変長引数 (variable length arguments)**をとるものがあります。本書の範囲を越えますが、自作プログラムでも `<stdarg.h>` の `va_arg()` といったマクロで実現できます。

このような関数では、プロトタイプ宣言による引数の型チェックは望めませんので、呼び出しには細心の注意が必要です。ただし、標準ライブラリ関数の `printf()` や `scanf()` については、コンパイラが特別扱いをして、書式文字列を解析して、続く引数の型チェックを行なう場合があります^{*9}。

なお、(単純化のためか) 可変長引数に現れる `float` は、`double` に格上げされることに決まっているので、`printf()` では `float` と `double` の区別がありません。(`scanf()` ではポインタで扱うため、`float*` と `double*` には大きな違いがあります。)

^{*7} `time()` で得られる現在時刻が、1 秒単位であることが多いためです。(言語規格上は処理系依存です。)

^{*8} 松本 眞氏の *Mersenne Twister* がおすすめです。

^{*9} 伝統的な (標準ライブラリ関数を特別扱いしない) 慣習からするとイレギュラー、しかし現実的な対処といえるでしょう。

付録 A

一覧表

A.1 記号の読み

+	プラス	.	ピリオド, ドット
-	マイナス, ハイフン	,	コンマ, カンマ
*	アスタリスク	:	コロソ
/	スラッシュ	;	セミコロン
\	バックスラッシュ, 逆スラッシュ	@	アットマーク
%	パーセント記号	?	クエスチョン, 疑問符
\$	ダラー, ドル記号	!	エクスクラメーション, 感嘆符
#	ナンバーサイン, シャープ	&	アンパサンド, アンド記号
~	チルダ		バーティカルライン, 縦棒, パイプ
_	アンダーバー, アンダースコア, 下線	^	ハット
=	等号, イコール	'	シングルコーテーション, シングルクオート, 一重引用符, アポストロフィ
< >	不等号, 小なり・大なり	"	ダブルコーテーション, ダブルクオート, 二重引用符
()	丸括弧, パーレン	'	バッククオート, 逆クオート
[]	角括弧, ブラケット		
{ }	波括弧, ブレース, カーリーブラケット		

A.2 ASCII コード

ASCII (American standard code for information interchange) コードは、文字コードの一種で、以下のような 7 ビットの値をとります。

	0x00	0x10	0x20	0x30	0x40	0x50	0x60	0x70
+0x00	NUL (ヌル文字)	DLE	␣	0	@	P	‘	p
+0x01	SOH	DC1	!	1	A	Q	a	q
+0x02	STX	DC2	"	2	B	R	b	r
+0x03	ETX	DC3	#	3	C	S	c	s
+0x04	EOT	DC4	\$	4	D	T	d	t
+0x05	ENQ	NAK	%	5	E	U	e	u
+0x06	ACK	SYN	&	6	F	V	f	v
+0x07	BEL (警報音)	ETB	'	7	G	W	g	w
+0x08	BS (後退)	CAN	(8	H	X	h	x
+0x09	HT (水平タブ)	EM)	9	I	Y	i	y
+0x0a	LF (改行)	SUB	*	:	J	Z	j	z
+0x0b	VT (垂直タブ)	ESC	+	;	K	[k	{
+0x0c	FF (改頁)	FS	,	<	L	\	l	
+0x0d	CR (復帰)	GS	-	=	M]	m	}
+0x0e	SO	RS	.	>	N	^	n	~
+0x0f	SI	US	/	?	O	_	o	DEL

- 行と列に振られた数値の和が ASCII コードです。例えば A は $0x40 + 0x01 = 0x41$ と読み取ります。
- 0x00 の NUL のように、2 文字以上のものは制御文字であり、画面表示できません。C 言語のエスケープシーケンスにあるものは意味を書いております。

A.3 エスケープシーケンス

文字定数（リテラル）や文字列リテラルでのエスケープシーケンスです。

表記	ASCII コード	意味
<code>\a</code>	0x07	警報音 (BEL)
<code>\b</code>	0x08	後退 (BS)
<code>\t</code>	0x09	水平タブ (HT)
<code>\n</code>	0x0a	改行 (LF)
<code>\v</code>	0x0b	垂直タブ (VT)
<code>\f</code>	0x0c	改頁 (FF)
<code>\r</code>	0x0d	復帰 (CR)
<code>\\</code>	0x5c	文字としての \
<code>\'</code>	0x27	文字としての '
<code>\"</code>	0x22	文字としての "
<code>\?</code>	0x3f	文字としての ?
<code>\0</code>	0	ヌル文字 (NUL)
<code>\ooo</code>	0ooo	8 進数 (ooo) の文字コードの文字
<code>\xhh</code>	0xhh	16 進数 (hh) の文字コードの文字

「文字としての」というのは、特殊文字の機能をキャンセルした、その文字自体です。

- `\` には「エスケープシーケンス開始」の機能があります。`\` 自体を表すには、`\\` と、2 度繰り返す必要があります。

例 (`\\`) : `printf("改行を示すには\\nと書きます。\\n");`

- `'` は、文字定数の開始と終了の両方を示すので、文字定数に `'` 自体を含めるには、エスケープして終了の機能をキャンセルする必要があります。

`"` は、文字列リテラルについて、同じことがあてはまります。

例 (`\'`) : `printf("' の文字コードは %d です\\n", '\');`

例 (`\"`) : `printf("\" の文字コードは %d です\\n", '\"');`

- `??` には、次の 1 文字と合わせた 3 文字で、別の 1 文字を表す **トライグラフ (trigraph)** という機能^{*1}があります。コンパイラによっては独自に無効にしている場合がありますが、エスケープすると可搬性が高まります。

例 (`\?`) : `printf("答えはいくつでしょう?\\??\\n");`

^{*1} ASCII コードよりも文字集合の小さな環境のために考案されましたが、今ではほぼ不要な機能です。なお C++ では、2017 年の規格で廃止されました。

A.4 EBCDIC コード

C 言語は、特定の文字コードに依存しません。ASCII コード以外の例として、EBCDIC (エビシディック、`extended binary coded decimal interchange code`) を挙げておきます。

	00	10	20	30	40	50	60	70	80	90	A0	B0	C0	D0	E0	F0
+0					␣	&	-						{	}	\	0
+1							/		a	j	~		A	J		1
+2									b	k	s		B	K	S	2
+3									c	l	t		C	L	T	3
+4									d	m	u		D	M	U	4
+5									e	n	v		E	N	V	5
+6									f	o	w		F	O	W	6
+7									g	p	x		G	P	X	7
+8									h	q	y		H	Q	Y	8
+9								'	i	r	z		I	R	Z	9
+A					[]		:								
+B					.	\$,	#								
+C					<	*	%	@								
+D					()	_	'								
+E					+	;	>	=								
+F					!	^	?	"								

- 上記は、EBCDIC のコード表の中でも、ASCII コードと同じ文字集合を使うものです。行と列に振られた数値は 16 進数で表記しています。
- EBCDIC は、企業内のメインフレーム (大型汎用機) やオフィスコンピュータなど、限られたところで用いられており、一般に目にすることはほとんどありません。

コラム：文字コード依存処理

'a'-'A' は、ASCII コードの環境なら 32、EBCDIC なら -64 です。

文字 `c` の小文字判定を `'a' <= c && c <= 'z'` で行うと、EBCDIC では `~ (0xa1)` を誤ります。

`<ctype.h>` にある `islower()` などの関数なら、文字コードの差異を吸収します。

(☞ 4.5.2 項)

A.5 演算子の種類

すべての演算子を、優先順位の高いものから列挙しました。(グループ内の横線のないところは、同じ優先順位です。)ただし、微妙な違いをこの表で調べて使い分けるのではなく、迷ったらカッコをつけて優先順位を明示すれば大丈夫です。

後置演算 →		ビットシフト →	
x [y]	配列参照	x << y	左シフト
x (y)	関数呼び出し	x >> y	右シフト
x . y	構造体メンバ参照	比較演算 →	
x -> y	構造体メンバ間接参照	x < y	小なり
x ++	後置インクリメント	x > y	大なり
x --	後置デクリメント	x <= y	以下
(T){ x } ^{C99}	複合リテラル	x >= y	以上
単項演算とキャスト ←		x == y	等しい
++ x	前置インクリメント	x != y	異なる
-- x	前置デクリメント	ビット演算 →	
& x	アドレス	x & y	ビット AND
* x	間接参照	x ^ y	ビット XOR
+ x	単項プラス	x y	ビット OR
- x	単項マイナス	論理演算 →	
~ x	1の補数 (ビット反転)	x && y	論理積
! x	論理否定	x y	論理和
(T) x	キャスト	条件演算 ←	
sizeof x	サイズ	x ? y : z	条件
sizeof (T)	サイズ	代入演算 ←	
_Alignof (T) ^{C11}	境界調整サイズ	x = y	代入
四則演算 →		x ⊗ = y	複合代入
x * y	乗算	⊗ は右のいずれか * / % + -	
x / y	除算	<< >> & ^	
x % y	剰余	順次評価演算 →	
x + y	加算	x , y	順次評価
x - y	減算		

(↗ 右上に続く)

- の部分が演算子、x, y, z, T はオペランドです。T は型名です。
- 矢印は、そのグループの演算子が、**右結合** (←) か、**左結合** (→) かを表します。同じ優先順位の演算が並んだときに、左右のどちらを優先するかを示すものです。
- ^{C99} 印は C99 で、^{C11} は C11 で導入されました。印なしは C89 にあったものです。

A.6 ユーティリティ関数 <stdlib.h>

標準ライブラリにある利用価値の高いユーティリティ関数を抜粋しました。これらを使うには、ソースコードに `#include <stdlib.h>` が必要です。

- 引数の `x` の型は、関数の型と同じです。
- 引数の `s` の型は文字列 (`const char *`) です。
- ^{C99} 印は C99 で導入されました。

絶対値		絶対値 (正確には <math.h> の関数)	
int	<code>abs(x)</code>	float	<code>fabsf(x)</code> ^{C99}
long	<code>labs(x)</code>	double	<code>fabs(x)</code>
long long	<code>llabs(x)</code> ^{C99}	long double	<code>fabsl(x)</code> ^{C99}
文字列→整数 (簡易)		文字列→浮動小数点数 (簡易)	
int	<code>atoi(s)</code>	—	—
long	<code>atol(s)</code>	double	<code>atof(s)</code>
long long	<code>atoll(s)</code> ^{C99}	—	—
文字列→整数		文字列→浮動小数点数	
—	—	float	<code>strtof(s,p)</code> ^{C99}
long	<code>strtol(s,p,b)</code>	double	<code>strtod(s,p)</code>
long long	<code>strtoll(s,p,b)</code> ^{C99}	long double	<code>strtold(s,p)</code> ^{C99}

- `ato-`で始まる簡易版の関数では、文字列をどこまで読み取ったかがわかりません。
- `strto-`で始まる関数の引数の `p` は文字列へのポインタ (`char **`) です。 `s` から解析を始めて、読み取りをやめた文字へのポインタが `*p` に書き込まれます。ただし `p` が `NULL` なら書き込まれません。
- 引数の `b` は `int` 型で、 $2 \leq b \leq 36$ の値で `b` 進数として解釈することを指示します。 `b = 0` の場合は、文字列の先頭が `0x` なら 16 進数、 `0` なら 8 進数、それ以外を 10 進数として解釈する指示になります。
- `strtol()` と `strtoll()` には、それぞれ `unsigned` 版の `strtolu()` と `strtollu()` もあります。

以下は本文を参照してください。

- プログラムの中断 [☞ 11.2.4 項](#)
- 擬似乱数 [☞ 13.9 節](#)

A.7 数学関数 <math.h>

標準ライブラリにある主要な数学関数^{*2}を抜粋しました。ソースコードに `#include <math.h>` と、環境によってはコンパイルオプションが必要です。(☞ A.10 節)

- 以下の関数は、戻り値、引数の x, y はいずれも `double` です^{*3}。
- ^{C99} 印は C99 で導入されました。

絶対値・剰余・最大・最小		三角関数 (角度はラジアン)	
<code>fabs(x)</code>	絶対値 $ x $	<code>sin(x)</code>	x [rad] の正弦
<code>fmod(x, y)</code>	$x \div y$ の剰余 (x と同符号)	<code>cos(x)</code>	x [rad] の余弦
<code>fmax(x, y)^{C99}</code>	x と y の最大値	<code>tan(x)</code>	x [rad] の正接
<code>fmin(x, y)^{C99}</code>	x と y の最小値	<code>asin(x)</code>	x の逆正弦 ($[-\pi/2, +\pi/2]$)
整数への丸め		<code>acos(x)</code>	x の逆余弦 ($[0, \pi]$)
<code>ceil(x)</code>	切り上げ $\lceil x \rceil$	<code>atan(x)</code>	x の逆正接 ($[-\pi/2, +\pi/2]$)
<code>floor(x)</code>	切り捨て $\lfloor x \rfloor$	<code>atan2(x, y)</code>	y/x の逆正接 ($[-\pi, +\pi]$)
<code>trunc(x)^{C99}</code>	0 方向への切り捨て	指数関数・対数関数	
<code>round(x)^{C99}</code>	絶対値の四捨五入 ^{*4}	<code>exp(x)</code>	指数関数 e^x
べき乗 (累乗)		<code>exp2(x)^{C99}</code>	2^x
<code>pow(x, y)</code>	べき乗 x^y	<code>log(x)</code>	自然対数 $\ln(x)$
<code>sqrt(x)</code>	平方根 \sqrt{x}	<code>log10(x)</code>	常用対数 $\log_{10}(x)$
<code>cbirt(x)^{C99}</code>	立方根 $\sqrt[3]{x}$	<code>log2(x)^{C99}</code>	二進対数 $\log_2(x)$
<code>hypot(x, y)^{C99}</code>	直角三角形の斜辺 $\sqrt{x^2 + y^2}$		

- 以下の関数型マクロは、浮動小数点数の種類を見分けます。
- 引数の x は `float`, `double`, `long double` のいずれの型でもよくて、戻り値は `int` です。

浮動小数点数の分類マクロ ^{C99}		浮動小数点数の種別を表す定数 ^{C99}	
<code>fpclassify(x)</code>	FP_NAN などの定数	FP_NAN	非数 (Not a Number)
<code>isnan(x)</code>	非数なら真 (0 以外)	FP_INFINITE	$+\infty$ または $-\infty$
<code>isinf(x)</code>	$\pm\infty$ なら真 (0 以外)	FP_ZERO	$+0$ または -0
<code>isfinite(x)</code>	非数, $\pm\infty$ 以外なら真	FP_SUBNORMAL	非正規化数 (絶対値が正規化数の最小値以下)
<code>isnormal(x)</code>	正規化数なら真	FP_NORMAL	上記以外 (正規化数)
<code>signbit(x)</code>	負 ($-\infty, -0$ 含む) なら真		

^{*2} 複素数の `<complex.h>` は C99 で導入されたものの、C11 でオプションに変更されたので、省きました。

^{*3} C99 からの機能で、関数名に接尾辞の `f` をつけると、`double` の戻り値と引数が、`float` に切り換わります。(つまり `double abs(double x)`; には `float absf(float x)`; が用意されています。) 接尾辞の `l` なら `long double` になります。さらに `<math.h>` の代わりに `<tgmath.h>`^{C99} を読み込むと、この切り替えを、引数の型に応じて自動的に行うようになります。tg は type-generic (総称型) の意味です。

^{*4} `round(x)` と同じ値を、`lround(x)`^{C99} は `long` 型で、`llround(x)`^{C99} は `long long` 型で返します。

A.8 printf()/scanf() の書式文字列

書式文字列に使える、主な機能を紹介します。^{C99}印は C99 で導入されました。注意の必要な **変則的** な部分に色をつけました。

変数の型	整数 ^{*5}				浮動小数点数 ^{*6}			文字 (列)	
	short	int	long	long long	float	double	long double	char	char*
printf 系	%hd	%d	%ld	%lld ^{C99}	%f (%lf ^{C99*7})		%Lf ^{C99}	%c	%s
scanf 系	%hd	%d	%ld	%lld ^{C99}	%f	%lf	%Lf ^{C99}	%c	%s ^{*8}

- printf() で % そのものを表示するには、%% と 2 回重ねます。
- scanf() に渡す変数には、アドレス演算子 & をつけて、ポインタにします。
- コンパイラによっては、可変長引数 (☞ 13.10 節) としては例外的に、型チェックの可能な場合があります。(☞ A.10 節の -Wall や -Wformat)

^{*5} printf 系の整数の d には、右のような形式のバリエーションがあります。また % と d などの間には 2 つの数値が指定できて、全体の最小文字数 x と、数値の最小桁数 y を %x.yd の形式で書きます。 x も .y も省略可能です。

d, i	符号付き 10 進数
u	符号なし 10 進数
o	符号なし 8 進数
x, X	符号なし 16 進数

例: ("%5d", 1) → "_____1"

例: ("%5.3d", 1) → "___001"

例: ("%1x", 65535) → "ffff"

^{*6} printf 系の浮動小数点数の f には、右のような形式のバリエーションがあります。また % と f などの間には 2 つの数値が指定できて、全体の最小文字数 x と、小数点以下の桁数 y を %x.yf の形式で書きます。 x も .y も省略可能で、.y を省略すると 6 とみなされます。g の小数部の末尾の 0 は省かれます。

f, F	小数表記
e, E	指数表記
g, G	f と e の自動切り替え

例: ("%6.3f", 1.2) → "___1.200"

例: ("%6.3e", 1.2) → "1.200e+00"

例: ("%6.3g", 1.2) → "___1.2"

- ^{*7} printf 系は可変長引数 (☞ 13.10 節) であるため、float と double に区別がなく、どちらも %f ですが、scanf 系の double は %lf です。これを混同する人が多くいたためか、C99 で printf 系にも %lf が追認され、%f と同じ動作をします。
- ^{*8} 変数に & が不要です。また、文字列長を制限するために必ず %99s のように、領域のバイト数よりも 1 小さい値を指定します。(終端のヌル文字の領域のためです。)

A.9 予約語

auto	else	long	switch	_Atomic ^{C11}
break	enum	register	typedef	_Bool ^{C99}
case	extern	restrict ^{C99}	union	_Complex ^{C99}
char	float	return	unsigned	_Generic ^{C11}
const	for	short	void	_Imaginary ^{C99}
continue	goto	signed	volatile	_Noreturn ^{C11}
default	if	sizeof	while	_Static_assert ^{C11}
do	inline ^{C99}	static	_Alignas ^{C11}	_Thread_local ^{C11}
double	int	struct	_Alignof ^{C11}	

- ^{C99} 印は C99 で、^{C11} は C11 で導入されました。印なしは C89 にあったものです。
- C99 で導入された `bool`, `true`, `false` は予約語ではなく、`<stdbool.h>` で定義されたマクロです。ヘッダを読み込んだときのみ有効で、`bool` は `_Bool` の別名になります。これらのシンボルを既に自前で定義しているプログラムへの配慮です。

A.10 コンパイルオプション

gcc のよく使われるコマンドライン引数です。コマンド名の `gcc` に続けて、スペースで区切って書き並べます。順序はあまり影響しませんが、コンパイルオプションはソースファイル名よりも前に指定したほうが安全です。例：`gcc -Wall -O2 -g source.c`

`-Wall` 警告メッセージを増やします。

`-Wextra` 警告メッセージをさらに増やします。

`-Wformat` `printf()` などの書式に関する警告を行います。`-Wall` に含まれています。

`-O0` 最適化を行いません。

`-O2` 最適化を行います。`-Wall` と組み合わせると、未初期化変数などの警告が増えます。

`-g` デバッグ情報を埋め込みます。実行時エラーの表示が詳細になります。

`-std=c99` ソースコードを C99 に準拠して解釈します。`M_PI` が無効になります。

`-std=gnu99` ソースコードを、C99 に GNU 拡張を加えて解釈します。

`-std=gnu11` ソースコードを、C11 に GNU 拡張を加えて解釈します。

`-lm` 数学ライブラリをリンクします。環境によっては不要です。

`-o filename` 出力ファイル名 (通常は `a.out` や `a.exe`) を `filename` に変更します。

`-c` オブジェクトファイルを出力します。分割コンパイルするときに使います。

`-Dmacro=def` マクロを定義します。ソース上の `#define macro def` と同じ働きです。

参考文献

- [1] B. W. Kernighan and D. M. Ritchie. *The C Programming Language, 2nd edition*. Prentice-Hall, 1988.
- [2] B. W. Kernighan (原著), D. M. Ritchie (原著), 石田 晴久 (翻訳). *プログラミング言語 C ANSI 規格準拠 第 2 版 (訳書訂正)*. 共立出版, 1994.
- [3] B. W. Kernighan (原著), R. Pike (原著), 福崎 俊博 (翻訳). *プログラミング作法*. KADOKAWA, 2017.
- [4] Samuel P. Harbison, 3 (原著), Jr. Steele, Guy L. (原著), 玉井 浩 (翻訳). *S・P・ハービソン 3 世と G・L・スティール・ジュニアの C リファレンスマニュアル*. エスアイビーアクセス, 2008.
- [5] WG14/N1256. Programming languages — C. <http://www.open-std.org/JTC1/SC22/WG14/www/docs/n1256.pdf>, September 2007.
- [6] WG14/N1570. Programming languages — C. <http://www.open-std.org/JTC1/SC22/WG14/www/docs/n1570.pdf>, September 2011.
- [7] WG14/N2731. Programming languages — C. <http://www.open-std.org/JTC1/SC22/WG14/www/docs/n2731.pdf>, October 2021.
- [8] 株式会社アंक. *C の絵本 第 2 版 C 言語が好きになる新しい 9 つの扉*. 翔泳社, 2016.
- [9] 村山 公保. *C プログラミング入門以前 [第 2 版]*. マイナビ出版, 2019.
- [10] 情報処理推進機構ソフトウェア高信頼化センター. *組込みソフトウェア開発向けコーディング作法ガイド: C 言語版: ESCR ver. 3.0*. SEC books. 情報処理推進機構 <https://www.ipa.go.jp/sec/publish/tn18-004.html>, 2018.
- [11] 奥村 晴彦. */改訂新版/C 言語による標準アルゴリズム事典*. 技術評論社, 2018.
- [12] 柴田 望洋. *新・明解 C 言語 入門編 第 2 版*. SB クリエイティブ, 2021.
- [13] 高橋 麻奈. *やさしい C 第 5 版*. SB クリエイティブ, 2017.

ソースコード一覧

1.1	初めてのプログラム	4
1.2	簡単な計算	10
1.3	簡単な変数	10
1.4	簡単な行列計算	11
1.5	丁半プログラム	13
2.1	型ごとの上下限の値を表示	32
3.1	正方形の面積を求める関数	37
3.2	三角形の面積を求める関数	40
3.3	三角形の面積を表示する関数	44
3.4	変数の値を入れ替える (?)	47
4.1	文字の種類を見分ける	64
4.2	条件を表す関数の例	65
4.3	if の入れ子と論理積の短絡評価	67
5.1	2 の累乗を列挙	72
5.2	x を越える最小の自乗数 (while 版)	73
5.3	偶数を表示する	76
5.4	約数の個数	78
5.5	約数の和	80
5.6	3 乗根を近似的に求める	83
6.1	x を越える最小の自乗数 (for 版)	92
6.2	コラッツ予想	94
6.3	九九の表	95
6.4	1 から n までの数をコンマ区切りで表示	96
6.5	素数判定	100
7.1	プロトタイプ宣言なし	104
7.2	プロトタイプ宣言あり	105
7.3	呼び出された回数を数える関数	113
8.1	配列の最大値	122

8.2	配列の最大値 (関数版)	127
8.3	月ごとの日数 (0 始まり)	128
8.4	月ごとの日数 (1 始まり)	128
8.5	配列要素のコピー	129
8.6	エラトステネスのふるい	131
8.7	ヒストグラム	133
8.8	覆面算	134
8.9	2 次元配列の縦横合計	137
9.1	文字列の長さ	142
9.2	文字列をコピー	144
9.3	文字列を連結	146
9.4	文字列の比較	150
9.5	スワップ関数	152
9.6	月名の配列	153
10.1	原点からの距離を表示	158
10.2	原点からの距離を表示 (ラベル付き)	159
10.3	分数の和と差	166
11.1	九九の表をファイルに出力	177
11.2	ファイルの読み取り (文字単位)	179
11.3	ファイルの読み取り (行単位)	181
11.4	ファイルのコピー	185
12.1	駐車場の自動清算機 (概形)	194
13.1	0.1 を 10 回足す	202
13.2	浮動小数点の誤差を見込んだ比較	203
13.3	コマンドライン引数を受け取る	205
13.4	キーボードから数値を受け取る	207

索引

■ 記号・数字

! (論理否定) 54
!= 52
& (アドレス演算子) 151
&& (論理積) 54
|| (論理和) 54
* (間接参照演算子) 151
< 52
<= 52
== 52
> 52
>= 52
\
(バックスラッシュ) viii, 31, 213
\
(ヌル文字) 140
\
(改行文字) 8
/* */ (コメント) 7
// (コメント) 7
0 オリジン 84, 85
1 オリジン 84, 85, 128
2 重ループ 88

■ A

ANSI C vi
ASCII コード 24, 212
ASCII 文字 4, 170
<assert.h> 199
atoi() 180, 205

■ B

bool 63, 219
break 90

■ C

-c 219
C11 vi
C89 vi
C99 vi
cat 2
cbirt() 84
cd 2
ceil() 191
char 24

<complex.h> 217
const 141
continue 90
Ctrl 73, 108, 198
<ctype.h> 62, 64, 214
C 言語 1

■ D

-D 219
DBL_EPSILON 204
DBL_MAX 23
DBL_MIN 204
#define 31, 63
do-while 92
double 22

■ E

EBCDIC コード 214
else 52
else if 53, 60
else 節 52
Enter 4
EOF 178
EUC-JP 170
EXIT_FAILURE 175
EXIT_SUCCESS 175

■ F

fabs() 82
FALSE 63
false 63, 219
fclose() 174
fg 73
fgetc() 178
fgets() 180
FILE 171
float 22
<float.h> 23, 32
FLT_ROUNDS 204
fopen() 172
for 74
fprintf() 176
fputc() 176

fputs() 176
_func 81

■ G

-g 219
gets() 188
getter 118

■ H

history 2
Home 108

■ I

if 52
#include 6, 41, 107, 216, 217
inf 203
int 20
int32_t 162
isalpha() 64
islower() 214

■ K

K&R (書籍) ix
K&R (言語規格) vi
kill 73

■ L

<limits.h> 21, 25, 32
-lm 41, 219
long double 22
long int 20
long long int 20
ls 2

■ M

M_PI 50, 219
main() 6, 186, 205
<math.h> 41, 82, 217
mkdir 2

■ N

n 79, 122
NaN 203, 217

NULL.....152, 172

■ O

-o filename.....219

-O0.....219

-O2.....219

■ P

printf().....6, 218

pwd.....2

■ R

rand().....209

return.....36

■ S

scanf().....206, 218

setter.....118

Shift_JIS.....6, 170, 178

short int.....20

signed.....20

signed char.....24

size_t.....141, 143

sizeof.....32, 124, 141

snprintf().....148

sprintf().....148

sqrt().....41, 82

srand().....13, 210

sscanf().....183

static.....110, 111, 136

-std.....vi, 50, 219

<stdarg.h>.....210

<stdbool.h>.....63, 219

stderr.....207

stdin.....207

<stdio.h>.....6, 107, 141

<stdlib.h>.....50, 175, 180, 205, 209, 216

stdout.....207

strcat().....146

strcmp().....150

strcpy().....144

<string.h>.....141

strlen().....142

struct.....156

■ T

Tab.....4, 108, 186

<tgmath.h>.....217

then 節.....52

time().....210

<time.h>.....13, 210

tmp.....29

tolower().....25

top.....73

toupper().....25, 154

TRUE.....63, 90, 117

true.....63, 219

typedef.....162

■ U

Unicode.....viii

unsigned.....20

unsigned char.....24

UTF-8.....170, 178

■ V

void.....44, 45, 49

■ W

-Wall.....55, 218, 219

-Wextra.....219

-Wformat.....218, 219

while.....72

■ あ

値渡し.....48, 160

アドレス演算子.....151, 218

アライメント.....161

暗黙的な宣言.....106

■ い

入れ替え.....→スワップ

入れ子.....54, 88

インクリメント.....30, 75

インデックス.....120

インデント.....4, 6, 45, 79, 91, 99, 134

隠蔽.....→シャドーイング

■ え

エスケープシーケンス.....8, 26, 140, 213

エスケープ文字.....172

エラトステネスのふるい.....130

円記号.....viii

演算子.....27, 55, 215

円周率.....50, 86

エンターキー.....ix

エンディアン.....20

■ お

オーバーフロー.....27

オープンモード.....172

オペランド.....27

■ か

改行文字.....4, 6, 8, 95

開発環境.....1

返り値.....36

拡張子.....3

可視範囲.....→スコープ

仮想ターミナル.....8

型.....16, 151

可搬性.....21, 154, 200, 213

可変長配列.....123

可変長引数.....210, 218

空文字列.....→くもじれつ

仮引数.....46, 109

関係演算子.....52

関数.....35, 111

関数スコープ.....111

関数プロトタイプ.....104

間接参照演算子.....151

慣用句29, 33, 37, 118, 124, 126, 127, 163, 178, 191

■ き

偽.....52

機械エプシロン.....204

擬似乱数.....14, 209

基本データ型.....162

キャスト.....30, 45, 46, 141, 143, 199

キャメルケース.....116

境界値分析.....70, 143, 201

境界調整.....→アライメント

行バッファ.....180

局所変数.....109

■ く

偶数丸め.....204

空文字列.....143, 147

繰り返し処理.....72

グローバルスコープ.....111

グローバル変数.....109, 114, 136

■ け

結合テスト.....113

言語規格.....vi

■ こ

合計.....80, 86, 121, 137, 138, 180

構造体.....156

個数.....78, 101, 182

コマンドインタプリタ.....2, 175

コマンドライン引数.....186, 205

コメント.....7, 81, 92, 145

コンソール.....8, 148

コンパイル.....3

コンパイル.....3

コンパイルエラー.....125

コンパイルオプション.....219

■ さ

再帰呼び出し.....48, 115

最大値.....121

左辺値.....17

参照渡し.....48, 126, 129, 164
参照.....17

■し

シェル.....1, 2
識別子.....19, 31, 36, 114, 116
字下げ.....→インデント
四捨五入.....33, 204, 217
辞書順.....150
指数関数.....→べき乗関数
指数表記.....22, 218
四則演算.....27
実行形式.....3
実行時エラー.....125
実行ファイル.....3
実数.....22
実引数.....46
シャドーイング.....114
寿命.....110
条件式.....52
条件分岐.....52
小数表記.....218
剰余演算子.....27
初期化.....18, 110, 120, 147
初期化子.....120, 124, 135, 140,
156, 164
書式文字列.....143, 154, 218
処理系依存.....21, 28, 75, 141, 200
真.....52
シンボル.....19

■す

数学関数.....217
スコープ.....41, 109, 111
スタックメモリ.....136
スネークケース.....116
スペーシング.....6, 199
スペース.....4, 6
スワップ.....29, 47, 126, 151

■せ

正規化数.....204
整数型.....20
全角スペース.....4
漸化式.....82

■そ

総称型.....217
添字.....120
ソース.....3
ソースコード.....3
ソースファイル.....3

■た

ターミナルエミュレータ.....8
大域変数.....109

代入演算子.....17
多次元配列.....136
多重ループ.....89
タブ文字.....4, 6
単項演算子.....28
単体テスト.....113, 201
単文.....61
短絡評価.....66, 138

■ち

長方形配列.....→配列の配列

■つ

通用範囲.....→スコープ

■て

定義.....16
定数.....16, 116, 122
ディレクトリ.....2
テキストエディタ.....vi, 4, 108,
176, 198
テキスト形式.....170
デクリメント.....30
デバッグ.....39
デバッグライト.....81

■と

ド・モルガンの法則.....57
統合開発環境.....1
統合テスト.....→結合テスト
度数分布.....132
トップレベル.....109, 160
トライグラフ.....213

■ぬ

ヌル文字.....140

■ね

ネイピア数.....102
ネスト.....→入れ子

■は

ハードコーディング.....186
バイナリ.....3
バイナリ形式.....170
パイプ.....208
配列.....120
配列の配列.....136
バグ.....39
バックスラッシュ.....viii
バッファ.....180
バッファオーバーフロー.....148,
149, 188
パディング.....161
番地.....151

■ひ

比較演算子.....52, 150
比較関数.....150
引数.....36
非数.....203, 217
ヒストグラム.....132
非正規化数.....204
左結合.....28, 215
標準入出力.....6, 187, 207
標準ライブラリ関数.....107, 139,
141

■ふ

ファイル.....2
ファイルスコープ.....111
ファイルポインタ.....171
フィルタ.....208
ブーリアン型.....→論理型
フォルダ.....2
複合代入演算子.....30
複合リテラル.....149
覆面算.....134
符号付き整数型.....20
符号なし整数型.....20, 141
不定.....18, 58
浮動小数点型.....22
浮動小数点数.....22
フラグ.....99, 117
プリプロセッサ命令.....31, 107
プリントデバッグ.....81
プログラムスコープ.....111
ブロック.....6, 36, 41
ブロックスコープ.....111
プロトタイプ宣言.....105
文.....6
分割コンパイル.....106, 109, 201,
219

■へ

平均.....180
平方根.....41, 82
べき乗.....37
べき乗関数.....37
ヘッダファイル.....106, 107
変数.....16

■ほ

ポインタ.....151
ポインタの配列.....136, 153
ポータビリティ.....→可搬性
補充入力.....186

■ま

マクロ.....31, 111, 116, 180
マクロ定数.....50, 122, 132, 175,
178
マジックナンバー.....122, 199

マルチバイト文字...6, 26, 143,
150, 170

■み

右結合...28, 215
未初期化...18

■む

無限大...203
無限ループ...73, 90

■め

メンバ...156

■も

文字エンコード...6, 143, 170
文字コード...24, 212, 214
文字集合...214
文字定数...24
文字列...26, 140
文字列定数...6

文字列リテラル...6, 8, 26, 140,
143

戻り値...36

■ゆ

ユークリッドの互除法...167
有効範囲...→スコープ
ユーザ定義型...162
優先順位...215
ユニットテスト...→単体テスト

■よ

要素...120, 127-129
要素数...120, 122, 124, 125, 135,
136, 149, 164
予約語...4, 19, 111, 219

■ら

ラインバッファ...→行バッファ
乱数...→擬似乱数
乱数の種...210

■り

リターンキー...ix
リダイレクト...207
履歴...186

■る

累乗...72
累乗関数...→べき乗関数
ルート...→平方根
ループ...72
ループ変数...74

■ろ

ローカルスコープ...111
ローカル変数...109, 114, 123,
136, 166
論理型...52, 93, 99, 100, 134
論理積...54
論理否定...54
論理和...54

著者

土村 展之（つちむらのぶゆき）

1997 年 京都大学工学部 数理工学科卒業

2002 年 東京大学大学院 情報理工学系研究科 科学技術振興特任教員

2005 年 東京大学大学院 情報理工学系研究科 助手

2008 年 関西学院大学 理工学部 情報科学科 教育技術職員，現在にいたる。博士（工学）

主要著書 『Java によるアルゴリズム事典』（共著、技術評論社）

森口 草介（もりぐち そうすけ）

2013 年 東京工業大学大学院 情報理工学系研究科計算工学専攻博士後期課程修了

2013 年 神奈川大学 外部資金雇用研究者

2014 年 関西学院大学 博士研究員

2015 年 関西学院大学 理工学部研究員

2016 年 関西学院大学 理工学部 契約助手

2019 年 東京工業大学 情報理工学院 助教，現在にいたる。博士（工学）

