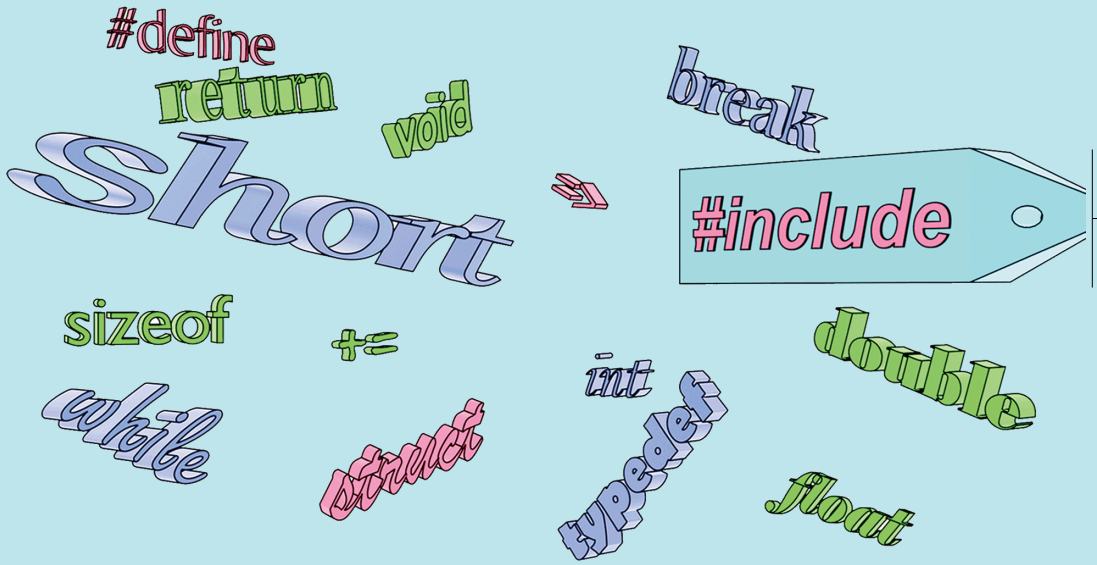
maintsume (2024-03-14 14:13)

キックオフ C 言語



関西学院大学 情報・機械系 実験準備室

2024 年度用

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。

授業でつまず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の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、同研究室の犬童梨子さん、矢野皓己さん、大島由裕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）のティーチ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していただき、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。

2024 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。大学の初等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伝えるために題材を絞ります。例えば以下のものは取り上げません。

- 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
- メモリの動的確保，再帰呼び出し，分割コンパイル

習得目標は、駐車場の自動精算機を模倣することです。そのために必要な機能を述べます。特に**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**代表格ですので、使用例を例題や練習問題にも取り入れました。ほかにも各場面で、初心者にありがちな間違いを指摘しました。

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 12 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（2024 年出版予定 [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[*2]。

[*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] C99 の文法のためにオプション指定の必要なコンパイラも存在します。例えば GCC の Ver. 4 までは "`gcc␣-std=gnu99␣source.c`" のように `-std=gnu99` が必要です。（☞B.11 節）

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)：
  Linux を含む Unix 系と、Windows の Cygwin，MSYS2，MinGW-w64，WSL2
- Clang：Linux を含む Unix 系と、Mac の Xcode
- Microsoft Visual C++：Windows

仮想ターミナルは、Windows 10 までの「Windows コンソールホスト」はコピー＆ペーストの操作が煩雑なので、MinGW では mintty を、WSL2 では wsltty を追加導入するのがお薦めでした。Windows 11 からの「Windows ターミナル」は良くなりました。

シェルは、Windows では PowerShell は癖が強いので、コマンドプロンプト（cmd.exe）か、できれば bash がいいでしょう。その他の環境では、標準のもので十分です。

テキストエディタは、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、予約語強調表示や自動整形くらいの機能は必要です。Microsoft 社の Visual Studio Code (VSCode) は、OS に依存せず、支援機能が充実しているので、よい選択肢です。

[*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。

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。

- Windows の初回セットアップ時にユーザ名の入力が求められ、**漢字氏名**にする人がいます。このユーザ名が、個人ファイル置き場の「フォルダ名」に流用される[*4]ことがあります。フォルダ名に**漢字**や**スペース文字**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るので、避けましょう。ただし、このフォルダ名やユーザ名を**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）**ウイ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 はディ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、ウィルス検査の対象、あるいはクラウドファイル共有の同期対象にすると、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ったり、故障時のデータ救出が困難になる[*5]副作用もあります。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。

[*4] ローカルアカウントの場合です。Microsoft アカウントならメールアドレスから生成されるようです。
[*5] 回復キーを保存していなければ、データ救出は原理的に不可能です。

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**」[13] 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。
- 「**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。
- 「**やさしい C**」[11] は簡潔な説明で、網羅的に内容を取り上げています。
- 「**C 言語 [完全] 入門**」[12] は、ページ数は多いですが、説明が丁寧で、紙面も読みやすいです。内容も新しく、C99 はもちろん、C23 まで言及があります。

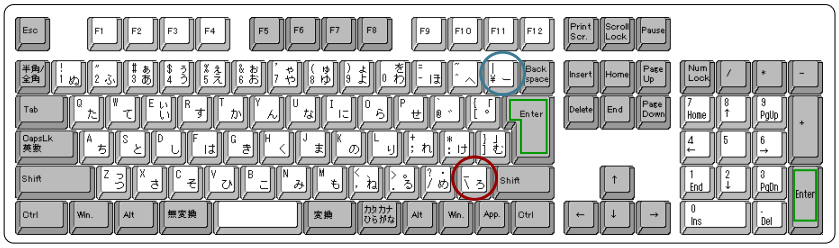
また、本書を一通り読み終えたら、次のような書物も参考になるでしょう。

- 「**C 言語による標準アルゴリズム事典**」[10] は、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**」[9] 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。

図 1　日本語キーボード（ODGA109A）のキー配列



バックスラッシュ・円記号

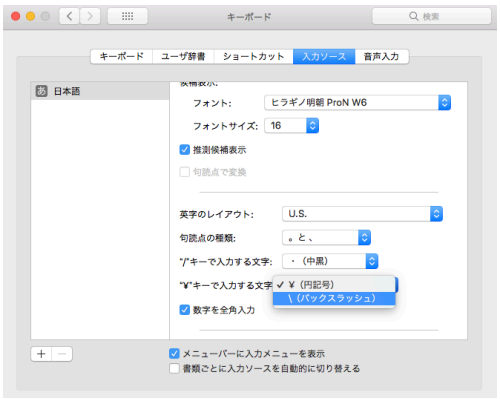
本書では、頻繁に \（バックスラッシュ）を記載しています。この文字は、日本ではこれまで ¥（円記号）に置き換えて使われてきました。そのため図 1 のように、日本語キーボードには \ と ¥ の両方のキーがあるのに、日本語 Windows 環境では区別されず、どちらを入力しても円記号が表示されてきました[*6]。

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　バックスラッシュ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、バックスラッシュ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、バックスラッシュ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、¥ で常にバックスラッシュ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」をバックスラッシュにしてください。



[*6] 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スラッシュ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のでしょう。

キー入力する文字の表記

キーボードから入力する文字は、細かな違いが問題になりがちです。本文中では、下記のように、黄色地のタイプライター書体にします。

```
gcc␣-Wall␣-O2␣source.c
```

- 通常のスペース文字は（前後の文字の間隔が広がるだけで）表記されませんが、上記のように、入力することが重要な場面では、「␣」と下線がついたように表記します。
- 数字の 0（ゼロ）は、タイプライター書体では「0」と、斜線が入ります。アルファベット大文字の O（オー）の「O」と区別するためです。上の例の -O2 は「マイナス・オー・に」です。

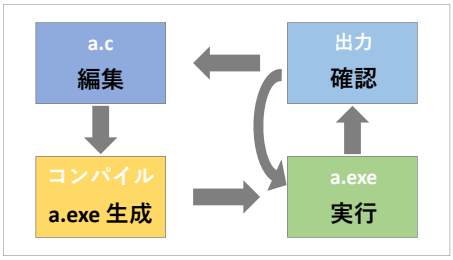
目次

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** (development environment) 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

1

ターミナル（例えば Cygwin Terminal）のウィンドウを開いて、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`（エル） 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。

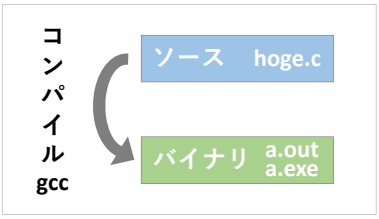
`history` : (history) `history` Enter は最近実行したコマンドを表示します。

`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 C++ 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

1

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>Enter
2 Enter
3 int main(void) {Enter
4 Tab printf("Hello, World!\n");Enter
5 Tab return 0;Enter
6 }Enter
```

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**（☞B.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。バックスラッシュ（ \ ）とエンターキーの注意は ix 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** (line feed 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」する必要もあります。

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と、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。

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。

文法エラー (syntax error) がなければ、実行ファイルが生成されます。ls Enter (Windows のコマンドプロンプトなら dir Enter) で確かめます。

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 C++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： コンパイルメッセージの読み方 (1)

コンパイル時に文法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**」このような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、実体は 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 C++ 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 B.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
:	コロン	\	バックスラッシュ，逆スラッシュ
;	セミコロン	'	シングルコーテーション，一重引用符
&	アンパサンド，アンド記号	"	ダブルコーテーション，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。役目は 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** (string constant) あるいは**文字列リテラル** (string literal) 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] 文字列リテラルに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、コンパイラ側に対策が必要です。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。

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領域のことです。コメントには、このようなメモを書き残せます。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：通常 /* **コメント** */ 通常）
1 行コメント　　// から行末まで（例：通常 // **コメント**）

注意すべき点として、複数行コメントは、さらに複数行コメントで囲おうとしても、入れ子にはできません。コメント開始の /* が何回あっても、1 回の */ でコメント終了です。（例：通常 /* **コメント** /* **コメント** */ 通常）

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活用します。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、コメントが役立ちます。

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領域を「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、日本語では「アンコメント」という人は少ないです。

1.5　複数行表示

1

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、画面上で**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　コンソールと改行

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。プログラムの出力する文字は、コンソールのモニタ上に表示されていました。今ではこの動作を、ソフトウェアで模倣しています。**仮想ターミナル** (virtual terminal) とも**ターミナルエミュレータ** (terminal emulator) とも、単に**ターミナル**とも呼ばれる、ウィンドウシステム上のソフトです。

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、我々はソフトウェアの**仮想ターミナル**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。

コンソールでの文字表示には癖があって、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、次に表示する位置は右にずれます。画面の**右端まで到達**すると、次の行（つまり下に）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。**改行文字**を表示しようとすると、何も表示されずに、**行の途中**でこの改行動作をします。

1.5.2　文字列リテラル中の改行文字

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。つまり、（Enter で入力する）本物の改行文字を含めるわけにはいきません。文字列リテラルでは**エスケープシーケンス** (escape sequence) を用いて \n と表記します。（☞B.3 節）

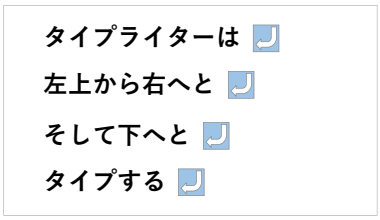
```
/* エラーになる例 */
printf("Hello, World!
"); // 途中で改行してはいけない
```

```
/* 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");
```



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意味です。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、1 つの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**して解釈されるので、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。これが実際の表示イメージにもっとも近いでしょう。

```
/* (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。

プログラムは、最初に main() 関数が実行されると決まっているので、**必ず main() 関数を作って**ください。実行すると「6」の数値が表示されるはずです。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);
5      return 0;
6  }
```

ソースコード 1.2 の実行結果

```
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。2 つの変数 a, b を作って、その値を表示してみました。

ソースコード 1.3　変数を用いた計算

```
1  #include <stdio.h>
2
3  int main(void) {
4      int a = 1;
5      int b = a + 3;  // 1 + 3 = 4
6      printf("a = %d, b = %d\n", a, b);              // a = 1, b = 4
7
8      printf("aとbの和は %d, 差は %d\n", a+b, a-b); // 和は 5, 差は -3
9      return 0;
10 }
```

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 2 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。そして、和（a+b）と差（a-b）の値も計算しました。

ソースコード 1.3 の実行結果

```
a = 1, b = 4
aとbの和は 5, 差は -3
```

このプログラムを改造して、変数を増やしたり、演算の種類も変えてみてください。掛け算の×はキーボードにないので、* で代用します。割り算の÷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　逆行列の計算

```
#include <stdio.h>

int main(void) {
    double a = 1.0;
    double b = 2.0;
    double c = 3.0;
    double d = 4.0;
    double t = a * d - b * c; // 行列式
    printf("%f %f -> %f %f\n", a, b,  d/t, -b/t);
    printf("%f %f    %f %f\n", c, d, -c/t,  a/t);
    return 0;
}
```

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 $\left(\begin{smallmatrix} 1 & 2 \\ 2 & 4 \end{smallmatrix}\right)$ 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) お釣りを計算して、3) 品物とお釣りを出す、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ないので、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 (warning) の場合は、実行ファイルは生成されるので、とりあえずの実行は可能ですが、潜在的な問題が隠れている場合もあります。正常に実行できたとしても、簡単に修正できるので、警告もなくすように心がけましょう。

コラム：コンパイルメッセージの読み方 (2)

☞5 ページより続く

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 'prinft' 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。(☞75 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。全体の動作例は右のようになるでしょう。キーボードから入力した文字は***赤色の斜体***にしています。

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？いくつ試してみましたか？どんなものを試してみましたか？

丁半プログラムの実行例（最初の二回の出力結果はこう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
```

[*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 $1/2^{10} \fallingdotseq 0.1\%$ ですから、まあ大丈夫でしょう。

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 は問題ないと考えるかもしれませんし、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。

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 かどちらかの擬似乱数を生成しています。詳細は A.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）規則で作り出しているので、再現可能です。この点で、本物ではない「擬似」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。

1.9 練習問題

1. **[初めてのプログラム（☞1.3.3 項、1.3.4 項）]**
   (i) 「:」「;」「'」「"」「_」の各記号の読み方を述べよ。
   (ii) `#include <stdio.h>` の「stdio」の読み方（あるいは意味）を述べよ。「スタジオ」ではない。
2. **[変数・式（☞1.6 節）]**
   10 ページのソースコード 1.3 を参考に、整数 y, m に関する、次の式の値を表示してみよ。（式の最後の「`% 7`」は 7 で割った余りを求めるので、0 から 6 の値になる。）

   ```
   (y + y/4 - y/100 + y/400 + 13*(m+1)/5) % 7
   ```

   そして、y 年 m 月 1 日（$3 \leq m \leq 12$）の曜日との関係で、気づいたことを述べよ。

第 2 章

型・値・式・変数

駐車場の自動精算機は、どのように車を管理しているでしょうか。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。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。

デジタルの計算機は、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て、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」の代わりに、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。分割コンパイルをすると区別が必要ですが、（本書の守備範囲の）単一ソースファイルでは、実質的に同じです。

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。変数には何度でも代入できます。最後に代入した値一つだけが保持されて、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が、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。格納先である左辺は、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」は左辺値ではなく、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** (refer) する」とも「変数に**アクセス** (access) する」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（☞3.4 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、代入忘れ防止に役立ちます。

```
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（☞43 ページのコラム）、コンパイラの警告（☞B.11 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。

- アルファベットの大文字と小文字、数字、アンダースコア（_）を組み合わせます。
- アルファベットの大文字と小文字は、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**予約語** (reserved word)（☞B.10 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、システムに予約されているので、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
automation
power_2
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になります。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。通常はもちろん **10 進数**です。

先頭が 0 なら **8 進数**で解釈します。
先頭が 0x なら **16 進数**になり、数字に加えて a~f の英字も使います。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領域サイズの違いで 4 種類の型があります[*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」

不等号はメモリ領域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、特に効果はありません。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。符号なしの表記にするなら、10 進の %u、8 進の %o、16 進の %x があります。%X なら大文字です。(☞B.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("%u", 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決めてよいものを**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、仮数(かすう)部と指数部に分けて格納されます。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、小数点も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しょうが、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。（☞B.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、指数の値に応じて `%f` と `%e` のどちらかが自動的に選ばれます。そして小数

[*12] 実数型 (real data type) 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。

[*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 ☞A.7.4 項。

の末尾の 0 が省かれます。

- %8f は、8 文字以上で表示されます。8 文字より少なければ、先頭にスペース文字が補われます。8 文字より多ければ、必要な文字数まで使われます。
- %.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定できます。省略時は 6 と同じです。%.0f のように 0 を指定すると整数で表示され、小数点も表示されません。

```
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（☞A.7 節）、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。

IEEE 754 という規格書で規定されたフォーマットが C 言語規格で推奨されていて、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**を囲います。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、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。どちらかというと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**（☞B.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ので、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。

文字の操作（例えば大文字変換）は、文字コード体系によって演算が違います。このような環境依存の動作は、(toupper() のように）システム（コンパイラ）が用意します。

```
#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すか？

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（"）で囲みます。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。


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

```
/* 文字列 */
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（☞B.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った気がしませんが、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() で出力するメッセージは文字列リテラルにしてきましたが、今後とも定数にしてください。これを変数にすると、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、l（エル）は 1（いち）と見間違えるので、大文字がよいでしょう。

整数（10 進数）は、基本的には int で、大きな数は値に応じて long や long long に自動的に切り替わります。明示的に指定するときに、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。

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

浮動小数点の場合は、基本的に double で、自動的には切り替わりません。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。

- **2 項演算子**なので、「2+3」のように、演算子の前後に 2 つのオペランドをとります。
- 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。
- 2 つのオペランドの型が**整数なら**、演算結果も**整数**です。「`5/2`」は 2 になります。
- 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）自動的に格上げして揃えられます。

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、**実行時エラー** (run-time error) で強制終了されることが多く、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**桁あふれ**（表現できる範囲を越えた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

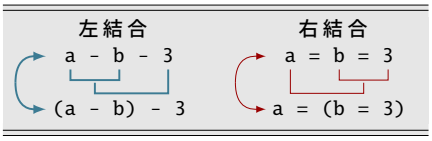
例えば式の先頭に「−3」が出てきた場合、このマイナスは 2 項演算子ではなく、オペランドが 1 つの**単項演算子** (unary operator) として働き、「3」の符号を反転します。表 2.6 のように、マイナスとの対称で、「+3」のような、プラスの単項演算子もあります。

2.6.2　優先順位

数学の式では、カッコ（）で**優先順位** (priority) を上げる（先に計算することを指示）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す。

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 B.5 節を参照してもらうと、単項演算子のほうが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 = 演算の値は、代入した値そのものですから、同じ値を連続して代入するのに使えます。



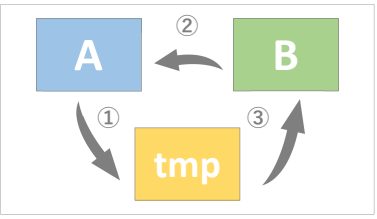
コラム：剰余はプラスに揃えてから

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。

この計算に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 (X) a と同符号　(Y) b と同符号　(Z) 常に 0 以上　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 (X) に定められました。(X) 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 (Y) を採用していますし、数学の有力な定義では (Z) になっていたりと、まちまちです。

		(X)		(Y)		(Z)	
a	b	/	%	/	%	/	%
+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

ここでは、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うと、x を移項して「0=1」の “解なし” になる、おかしな式ですが、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) との呼び名があって、「`x++;`」の専用の単項演算子（表 2.6）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、専用の呼び名はありませんが、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;`」と少し短く記述できます。

```
x = x + 5;    ⇔    x += 5;
x = x - 5;    ⇔    x -= 5;
x = x * 5;    ⇔    x *= 5;
x = x / 5;    ⇔    x /= 5;
x = x % 5;    ⇔    x %= 5;
```

- 四則演算すべてに、この複合代入があります。変数部分が長いときに有用です。
- + 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 と整数で得られますが、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。（☞77 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。

```
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（☞B.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、下位の 1 バイト分だけが取り出されます。

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

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

変化しない値である**定数**は、プログラムではどう扱うのでしょうか。変数に代入してもよいのですが、マクロという機能による**マクロ定数**のほうが安全です。間違って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 (macro) は、#define 命令[*19]で「文字の置き換え」を指示したものです[*20]。置き換え文字に定数を指定したものが**マクロ定数** (macro constant) です。マクロ名にも識別子を用いますが、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 EPS      0.000001
#define POW3_3   (3*3*3)
```

#define 文はインデントせずに、**プログラムの冒頭**（#include 文のすぐ下）に記述するのが習慣です。（☞39 ページのソースコード 3.3，135 ページのソースコード 8.8 など）

[*19] # 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。セミコロンは関係ありません。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。

[*20] 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ので、定義する数値全体をカッコで囲う必要もあります。（☞125 ページのコラム）

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);          // 最大値
7      printf("INT_MIN=%+d\n", INT_MIN);          // 最小値
8      printf("sizeof(int)=%zu\n", sizeof(int)); // バイト数
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);          // 有効精度（桁数）
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 `(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 `(int)(x-0.5)` と、0.5 を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には、数学関数（☞B.7 節）を用いて `floor(x+0.5)` あるいは C99 で新設された `round(x)` が簡便です。さらに long long 型で得るなら `llround(x)` もあります。なお、`printf("%.0f", x)` で表示される値の丸め方向は、C99 では `FLT_ROUNDS`（☞A.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

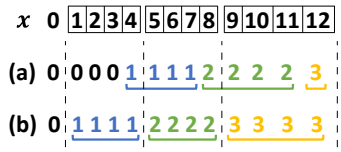
1. **[代入・左辺値・初期化・複合代入（☞2.2.2 項、2.2.3 項、2.7.2 項）]**
   プログラム中の `int` 型変数 `x` について、以下を説明せよ。
   （a）「`x = 10;`」と「`10 = x;`」の違いは何か。（コンパイルエラーに着目せよ。）
   （b）「`int x;`」の定義直後の `x` が 0 である保証はあるか。
   （c）「`x = x + 5;`」は、どのような作用があるか。また += 演算子を用いて同じ動作を実現せよ。

2. **[変数への代入（☞2.2.2 項）]**
   右のプログラムから、無駄を省け。

```c
#include <stdio.h>

int main(void) {
    int a = 10;
    a = 20;
    a = 30;
    a = 40;
    a = 50;
    printf("%d\n", a);
    return 0;
}
```

3. **[左辺値（☞2.2.2 項）]**
   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 型の変数として定義せよ。
4. **[スワップ（☞2.7.1 項）]**
   変数 a, b, c がある。$a \leftarrow b \leftarrow c \leftarrow a$ とプログラムで 3 つの値を入れ替えてみよ。
5. **[剰余と周期性（☞2.6 節）]**
   度数法の 2 つの角度 a, b $(0 \le a, b < 360)$ の和 $(a+b)$ と差 $(a-b)$ の値を、それぞれ 0 以上 360 未満で求めよ。（つまり、例えば 360 なら 0、−1 なら 359 にせよ。）a, b とも int 型とし、剰余演算子 % のオペランドは負にならない式にせよ。
6. **[剰余と十進数表記]**
   正の整数 x を 123 とするとき、以下の値を表示して確かめよ。（商は整数とする。）
   - 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。
   - y を 10 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 の式で表して、プログラムで表示せよ。（**ヒント**：(b) は、x の 3 ずれた (a) と同じ。）
   (a) 満席になった長椅子の台数（高々 1 台の満席ではない椅子を含めない）
   (b) x 人が座るのに必要な長椅子の台数（満席ではない椅子を含める）



8. **[切り上げ・マクロ定数（☞2.8 節）]**
   整数 x (>0) について、西暦 x 年が何世紀であるかをプログラムで表示せよ。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。定数 100 は、マクロ `YEAR_PER_CENTURY` と定義してから用いよ。西暦 0 年は存在しないが、誤って計算しても「1 世紀」にならない式にせよ。

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。駐車場の自動精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。

なお、本文では関数の呼び方に次のようなバリエーションがありますが、すべて同じ意味です。

- f 関数
- f() 関数
- 関数 f
- 関数 f()
- 関数 f(x)
- f()
- f(x)
- double f(double x)

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** (function) も、やはり 3 つの要素があります。

（数学の文法）

関数名 (入力となる変数) = 関数の返す値

```
/* C言語の文法 */
型名 関数名（引数）{
    return 戻り値;
}
```

関数の範囲

（数学の実例）

$f(x) = x^2$

```
/* C言語の実例 */
double f(double x) {
    return x * x;
}
```

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 ( ) を書いて、このカッコの中に型名と変数名のペアを書き並べます。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**(もどりち)とか**返り値**(かえりち)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、その後に現れる `return` のすぐ後です。関数の手続きの範囲は { } で囲って示します。その中で、戻り値を計算する手続きが先にあって、最後に return 文と戻り値を書きます。

頻出ミス

戻り値の、型名は**関数名の前**、値は **return の後**と、離れたところに書くので注意してください。そして、関数名の前の型名と、return の後の値の型は、一致させておくべきです。（☞46 ページのコラム）

「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。

ソースコード 3.1　2 乗を求める関数

```
1 /* x の2乗を返す */
2 double f(double x) {
3     return x * x;
4 }
```

$f(x) = x^2$ を C 言語で実現したものがソースコード 3.1 です。関数の**名前**は、数学の関数と同じ f にしました。**引数**は double 型の変数 x で受け取り、**戻り値**は double 型で返します。計算手順はあまりにも短いので、return 文に直接 `x*x` と[*2]書きました。

なお、この **return 文の**（`x*x` の）**式の型**は double ですから、**戻り値の型名**の double と一致して、首尾一貫しています。

3.1.1　ブロック

ブロック (block) とは、{ } で囲んだ、制御構造のひとかたまりです。これまで関数の範囲を示すのに使ってきましたが、関数の内部にも、新たなブロックをつくれます。

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

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

上記ではブロックを 2 個ずつ作りました。左のように**直列**に並べたり、右のように**入れ子**にもできます。このような単独のブロックを作る機会はそれほどありませんが[*3]、今後は条件分岐やループ処理などの構文と組み合わせて、頻繁に用いるようになります。

[*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。数学関数にはべき乗関数 pow(x,y) がありますが、計算速度は掛け算の 5〜20 倍遅いのが通例で、誤差が混入する可能性もあるため、多用されません。

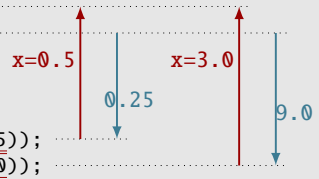
[*3] 変数のスコープ（☞3.4 節）を、わざと狭くするときに使います。

3.2　関数呼び出しと実行順序

では、作った関数 f() を呼び出してみましょう。数学では $f(0.5)$ の表記で $f(x)$ の $x = 0.5$ のときの値がわかります。プログラムでも同じような表記をします。ソースコード 3.2 では main() 関数から f(0.5) を呼び出して、printf() でその値「0.25」を表示してみました。f(3.0) なら「9」と、引数に応じて計算しなおされます。printf() 書式を"%g"としたので、小数部分の 0 が表示されていませんが、内部ではちゃんと double 型の値になっています。

ソースコード 3.2　2 乗を求める関数を呼び出す

```
1  #include <stdio.h>
2
3  /* x の2乗を返す */
4  double f(double x) {
5      return x * x;
6  }
7
8  int main(void) {
9      printf("f(0.5) は「%g」です\n", f(0.5));
10     printf("f(3.0) は「%g」です\n", f(3.0));
11     return 0;
12 }
```



ソースコード 3.2 の実行結果

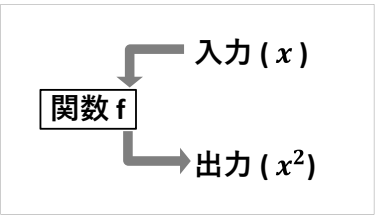
```
f(0.5) は「0.25」です
f(3.0) は「9」です
```

このように、main() から、何度でも f() を呼び出せて、引数の値は、その時ごとに違った値にできます。

ところで、ソースコードには f() のほうが先に現れていますが、**最初に実行されるのは main()** です。つまり関数の**「ソースコード上の記述順序」と「実行の順序」は無関係**ということです。ちなみに、ソースコード上の記述順序を入れ替えると、このままではコンパイルエラー（あるいは警告）になります。ひとまず main()（呼び出し側）を後に書くことにしてください。この記述順序の制約から逃れる方法は 7.1 節で説明します。

コラム：関数の内側・外側

すべての**文**は、どれかの関数に属するよう、関数ブロックの内側に書く必要があります。関数の外側に書くものは、#include や特別な変数の定義など、ごく限られたものです。



次は、関数 f() を様々な引数で呼び出してみましょう。**引数には計算式**を渡せるので、その例です。ソースコード 3.3 の f() は、これまでと同じですが、改行を省略しました。

ソースコード 3.3　$f(x)$ の呼び出しの組合せ

```
1  #include <stdio.h>
2  #define EPS 0.0001
3
4  /* x の2乗を返す */
5  double f(double x) { return x * x; }
6
7  /* f(x) の数値微分を返す */
8  double g(double x) { return (f(x + EPS) - f(x)) / EPS; }
9
10 /* x の4乗を返す */
11 double h(double x) { return f(f(x)); }
12
13 int main(void) {
14     printf("f(3.0) = %.8f\n", f(3.0)); // f(3.0) = 9.00000000
15     printf("g(3.0) = %.8f\n", g(3.0)); // g(3.0) = 6.00010000
16     printf("h(3.0) = %.8f\n", h(3.0)); // h(3.0) = 81.00000000
17     return 0;
18 }
```

$f(x)$ の数値微分、つまり微小な ϵ に対して

$$g(x) = \frac{f(x+\epsilon) - f(x)}{\epsilon}$$

を求める関数 g(x) を作りました。2 行目で ϵ をマクロ定数の EPS として定義しています（☞2.8 節）。8 行目で、数学と同じような記法で f() を 2 回呼び出しました。

2 乗を計算する f() を 2 回連続で呼び出すと、4 乗になります。数学では

$$h(x) = f(f(x))$$

と表記しますから、11 行目で関数 h(x) を、ほぼ同じ表記で実現しました。

最後に main() から f(3.0), g(3.0), h(3.0) を呼び出しました。printf() の `%.8f` で表示したので、コメントに書いたように、小数部分が 8 桁になりました。

3.3　複数の引数・関数の役割分担

三角形の面積を求める関数を作りましょう。ソースコード 3.4 では、いくつかある公式の中で、**頂点座標**から求めるものを採用してみました。関数名には vertex（頂点）の意味で 'v' をつけてみました。

3 頂点が**原点と 2 点** $(x_1, y_1), (x_2, y_2)$ であれば、面積 S は次の簡単な式になります。

$$S = \frac{|x_1 y_2 - x_2 y_1|}{2}$$

4 行 関数の名前は `triangle_v2`、型は 5 行目の return に合わせて double にします。引数として xy 座標を 2 組受け取りたいので、4 つの変数を書き並べています。**複数の引数**は、このようにコ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、「double x1,y1, x2,y2」とまとめることはできません。

5 行 return で S を返します。`fabs()` は、標準ライブラリの**数学関数** (mathematical function) で、引数を double 型で受け取り、その**絶対値** (absolute value) を double 型で返します。S の値が double 型なので、4 行目の関数の型も double にします。

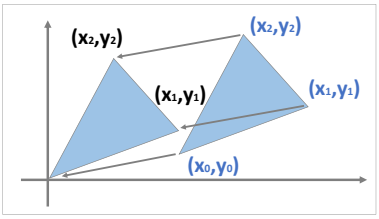
3 頂点が**原点を含まない** $(x_0, y_0), (x_1, y_1), (x_2, y_2)$ だとしましょう。この場合は、

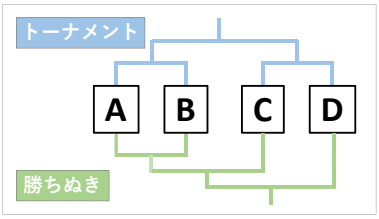
$$\begin{cases} x_0' = x_0 - x_0 = 0 \\ y_0' = y_0 - y_0 = 0 \end{cases} \qquad \begin{cases} x_1' = x_1 - x_0 \\ y_1' = y_1 - y_0 \end{cases} \qquad \begin{cases} x_2' = x_2 - x_0 \\ y_2' = y_2 - y_0 \end{cases}$$

のように、全体を $(-x_0, -y_0)$ 方向に平行移動すると、原点を含むようになるので、S の公式が適用できます。そこで関数 `triangle_v3` では、11–12 行目で**平行移動の計算だけ**を自力で行いました。面積の計算はせずに、最初の関数を呼び出して、得られた値をそのまま return で返すという**役割分担**にしたのがうまいところです。関数を呼び出す際の**複数の引数**も、このようにコンマで区切って並べます。

ソースコード 3.4　三角形の面積を求める関数（座標版）

```
1  #include <math.h> // fabs() に必要
2
3  /* 三角形の面積（頂点が原点，(x1,y1)，(x2,y2)）*/
4  double triangle_v2(double x1, double y1, double x2, double y2) {
5      return fabs(x1 * y2 - x2 * y1) / 2.0;
6  }
7
8  /* 三角形の面積（頂点が(x0,y0)，(x1,y1)，(x2,y2)）*/
9  double triangle_v3(double x0, double y0,
10                     double x1, double y1, double x2, double y2) {
11     return triangle_v2(x1-x0, y1-y0,
12                        x2-x0, y2-y0);
13 }                    //  (-x0,   -y0) 方向に平行移動して、下請けに出す
```





1 行目の「#include <math.h> 」という記述は、標準ライブラリの数学関数を使うときに必要です[*4]。#include の役目は 7.1.2 項で説明します。数学関数の一覧は B.7 節にあります。数学関数は、引数も戻り値も double のものが基本ですが、標準ライブラリ関数なら**型を確かめてから呼び出す**のが本来の姿です。確かめ方も 7.1.2 項で説明します。

3.3.1　関数の組合せ

関数は、**式の一部**でありながら、**引数には式**をとるという、**再帰的な性質**を持っています。そのため、f(f(x)) のような呼び出しができるのですが、このような使い方を思いつくのは意外に難しいので、もう少し例を挙げてみます。

数学関数の double fmax(double a, double b) は a と b の大きい方を返します。これを利用して、ソースコード 3.5 では a, b, c, d の最大値を求める関数 fmax4(a,b,c,d) を作ってみました。6 行目の **fmax() の引数を fmax() にする**のが思いつきにくいでしょう。fmax(a,b) と fmax(c,d) を先に計算して、これらの値の大きい方を、外側の fmax() が求めます。言ってみれば、4 チームのトーナメント形式で最大値を求めます。

ソースコード 3.5　4 つの値の最大値を求める関数

```
1  #include <stdio.h>
2  #include <math.h> // fmax() に必要
3
4  /* a,b,c,d の最大値を返す */
5  double fmax4(double a, double b, double c, double d) {
6      return fmax(fmax(a,b), fmax(c,d));  // トーナメント
7  //  return fmax(fmax(fmax(a,b), c), d); // 勝ち抜き
8  }
9  int main(void) {
10     printf("fmax4(1, 3, 5, 7) = %g\n", fmax4(1, 3, 5, 7)); // 7
11 }
```

引数の組合せを変えれば、勝ち抜き形式にもできます。（もちろん結果は同じです。）

このように**小さな機能を積み重ねて**、徐々に大きな、役立つ作業を実現していくのは、コンピュータ言語全般に通用する重要な手法です。

[*4] Unix 系の gcc のように、さらにコンパイラに -lm のオプションが必要な場合もあります。（☞B.11 節）

3.4　変数の有効範囲（スコープ）

三角形の面積の、別の公式を使ってみましょう。ヘロンの公式は少し複雑です。3 辺の長さを a,b,c としたときに、$s = \dfrac{a+b+c}{2}$ という変数を用意して、面積 T は

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

となります。これをソースコード 3.6 の関数 `triangle_side(a,b,c)` で実現しました。

6 行目では s、7 行目では area2 という変数を定義して、公式の計算を段階的に行います。area2 は面積の 2 乗（つまりルートの中身）です。

8 行目の return 文の `sqrt()` は、やはり数学関数で、**平方根** (square root) を double 型で返します。

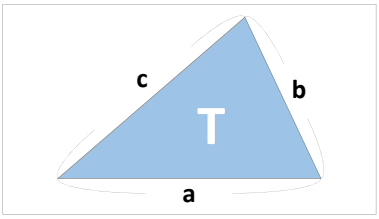
最後に main() から triangle_side() を呼び出します。**コンマの後にスペース文字**を入れるのは、そもそもの**基本的な習慣**ですが、引数に小数ばかりが並ぶ時には、切れ目がすぐにわかるので**特に有用**です。（ピリオドの後だとコンパイルエラーになるので、スペース文字の場所を間違えたらすぐにわかります。）

ソースコード 3.6　三角形の面積を求める関数（辺長版）

```
1  #include <stdio.h>
2  #include <math.h> // sqrt() に必要
3
4  /* 三角形の面積を返す（３辺の長さをa,b,cとする） */
5  double triangle_side(double a, double b, double c) { a,b,c のスコープ
6      double s = (a + b + c) / 2.0;                        s のスコープ
7      double area2 = s * (s-a) * (s-b) * (s-c); area2 のスコープ
8      return sqrt(area2);
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_side(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_side(3.0, 4.0, 5.0));
16     return 0;   スペース文字で切れ目を目立たせる
17 }
```

ソースコード 3.6 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。
```



関数で定義した変数は、プログラムのどこから使えるのでしょうか。変数の使える範囲を**有効範囲**[*5]あるいは**スコープ** (scope) といい、**定義したブロックの最後まで**と決まっています。ブロックを抜け出すと、変数は代入も参照もできなくなります。

つまり、ソースコード 3.6 の 6 行目の変数 s は、ブロックの終わりである 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() のような場所では使えないということです。たとえ main() で同じ名前の s という変数を用意したとしても、triangle_side() 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。

引数のスコープは、（引数名が最初に現れるのは関数のブロックの外側ですが）**関数のブロックの内側の先頭**で定義された場合と同じです。つまり引数は、関数ごとに別のものになります。実際に、39 ページのソースコード 3.3 の関数は f(x), g(x), h(x) と、引数の名前がどれも同じですが、実体は違っていて、数学の関数のような計算ができました。

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。引数の名前を変更した場合でも、影響範囲はその関数内だけで、呼び出す側には何の関係もありません。

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と、良いことがたくさんあります。

[*5] 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。プログラミング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。

3.5　プログラムの信頼性

さて、42 ページのソースコード 3.6 で三角形の面積を計算しましたが、もっと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。もしも異なる数値を書いてしまったら、実行結果はおかしくなります。


```
    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_side(3.0, 4.0, 5.0));
//           表示用                    計算用
```

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 3.7 の新しく作った print_triangle_side()（12 行〜15 行）を見てみましょう。引数で受け取った a,b,c を、printf() の表示と triangle_side() の面積計算の**両方**に使います。この関数を呼び出す限り、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_side(a, b, c));
//         表示用                 計算用
```

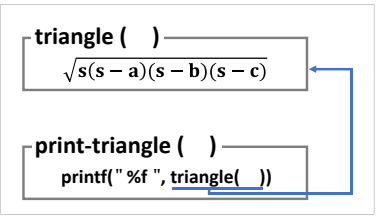
ソースコード 3.6 とソースコード 3.7 を比べると、まったく同じ実行結果が得られるのに、関数を余分に書いたほうがプログラムが長くなったように思えるかもしれません。しかし、表示したい三角形を増やしていくと、長さが逆転します。

3.6　void 型の関数

ソースコード 3.7 の 11 行目の print_triangle_side() 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[*6]、この関数では printf() で表示するという動作をしていますので、C 言語で考えると意味のある作業をしています。

[*6] 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。



ソースコード 3.7　三角形の面積を表示する関数（辺長版）

```
1  #include <stdio.h>
2  #include <math.h> // sqrt() に必要
3
4  /* 三角形の面積を返す（３辺の長さをa,b,cとする） */
5  double triangle_side(double a, double b, double c) {
6      double s = (a + b + c) / 2;
7      double area2 = s * (s-a) * (s-b) * (s-c);
8      return sqrt(area2);
9  }
10
11 /* 三角形の面積を表示する（３辺の長さをa,b,cとする） */
12 void print_triangle_side(double a, double b, double c) {
13     printf("3辺の長さが%g,%g,%gの三角形の面積は%gです。\n",
14            a, b, c, triangle_side(a, b, c));
15 }      //  表示用                  計算用
16
17 int main(void) {
18     print_triangle_side(1.0, 1.0, 1.0);  // void 型の関数なので、
19     print_triangle_side(3.0, 4.0, 5.0);  // 関数呼び出しのみを書く
20     return 0;
21 }
```

この関数は返す値がないので、return 文がありません（15 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（18–19 行目）。

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。

3.7　出力＝ return

return 文は、関数の処理の最後に書いて、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正統です。（しかし現実は甘くはありません。☞下のコラム）

void 型の関数には、return 文を書く必要はありませんが、書きた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。もし関数の途中に書くと、それ以降の処理を行わずに、関数を抜け出します。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にすみます。

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 の式の型が一致しなければ、黙ってキャストされて、警告すら出ないこともあります。例えば int 型の関数で「`return 3.14;`」とすれば、**int の 3** が返されます[c]。

```
double f(double x) { // × 不定
    x = x * x;   // return なし
}

int pi(void) {   // × 3
    return 3.14; // intになる
}
```

このように、コンパイルに成功してもまったく安心できません。せめて警告レベルを上げる（☞B.11 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暗黙のキャストは、文法エラーになります。C 言語では、char と int を（わざと）混同してきている場面があるため、網羅的に検査することは難しいのですが、限られた場面で警告を出すコンパイラ（例えば Clang）もあります。

表 3.1　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	`int f(int x) {...}`	`y = f(2);`
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

3.8　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。

3.8.1　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、呼び出された側とでは役割が違うので、表 3.1 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。

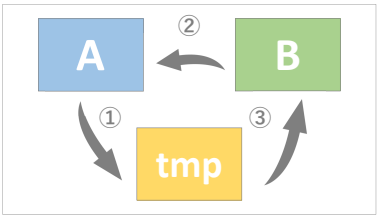
実引数 (actual argument)　関数を呼び出す側が与える値です。定数を与えている場合など、具体的な値が事前にわかることがあります。プログラムに何度現れてもよくて、その度毎に値が異なっても構いません。

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[*7]が、使い分けると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。

[*7] 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、両方とも parameter にしたり、逆に両方とも argument にする例もみられます。



3

3.8.2　値渡し・参照渡し（発展的内容）

2.7.1 項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 3.8 では、main() の変数の入れ替えが起きません。確かに swap() 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 3.8　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) {                                    a,b のスコープ
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;                                        a,b のスコープ
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の値は（当然）入れ替わらず、エラーにもならない
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。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 行目までがスコープです。つまり、2 つの a 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() は、仮引数の a を 1 で、b を 2 で**初期化**して、5–7 行目の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 3.2　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() 関数）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。コピーを作ってから渡して、呼び出し元への影響をなくしています。

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。これが正常に実行できる（つまりコンパイルエラーにならない）なら、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() 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、手間をかけて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ありますが、理由はそれだけでしょうか。

C 言語が登場した 1970~1980 年代、新しい言語の特徴として求められた機能には、(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) 再帰呼び出し可能、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のでしょう。

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（引数を変えながら）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。

3.8.3　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：(void) vs. ()

引数なしを示すのに、C++ や 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。どちらかというと苦肉の策です。

3.9　関数を作る意義

ある手続きを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。手続きが単純か複雑かは別次元の話です。呼び出し側は、名前を頼りに関数を呼び出して、実際に行われる**具体的な処理内容を知らず**にすむのです。

手続きが独立すると**「入力」と「出力」が明確**になります。例えば、42 ページのソースコード 3.6 の triangle_side() 関数であれば、引数が 3 辺の長さですから、「三角形の面積は 3 辺の長さで計算できて、三角形の場所には影響されない」ことがわかります。さらに「辺の長さが double ならば面積も double で得られる」こともわかります。40 ページのソースコード 3.4 の triangle_v3() 関数であれば、「3 頂点の座標がわかればよくて、辺の長さは不要」ということです。手続きの内容以外からも、これだけのことがわかります。

39 ページのソースコード 3.3 では $f(x) = x^2$ という単純な関数を作りました。5 行目の g(x) は、この数値微分を求めていますが、`return ((x+EPS)*(x+EPS) - x*x) / EPS;` などと、$f(x)$ を展開した形で書いてしまうと、関数 f() を書き換えるたびに、関数 g() まで修正せねばなりません。「f() がどんな関数でも、数値微分を求める」ために、**f() の処理内容がどんなに単純であっても**、関数 g() からは f() を呼び出す形にしておきましょう[*8]。

関数は他にも使い道があります。同じ結果になる違う手続きを、別の関数として作っておいて、必要に応じて差し替えることもあります。対になる処理を規則的に作る場面でも役立ちます。本書でも折りに触れて例を示しますので、徐々に身につけていきましょう。

[*8] C99 では、関数のインライン展開を指示する予約語 inline が新設されました。コンパイラが、単純な関数の展開を肩代わりすることも期待できます。

コラム： main も関数

我々がいつも書いている「`int main(void) ...`」も関数ですが、引数なし、戻り値が int 型と決められています。return ではいつも 0 ばかり返す、変わったものです。そのためか、C99 からは main() の「`return 0;`」は**省略可能**になりました。

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。

3.10　練習問題

1. **[関数の入出力（☞3.1 節、3.7 節）]**
   右の関数 cubic(x) は x^3 を返すが、不自然なところがある[*9]。アイウの型のうち直接的に一致させるべきものを述べ、修正方法を 2 通り提案せよ。

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

2. **[単語の読み（☞3.1 節、3.8 節）]**
   「引数」と「仮引数」の、専門用語としての読みを述べよ。「いんすう」ではない。
3. **[関数を作る意義（☞3.9 節）]**
   下の rectangle() 関数は、長方形の面積を返す。長方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、どんな条件でどのような値をとるのか。）

```
/* 長方形の面積を返す（幅をw, 高さをhとする）*/
int rectangle(int w, int h) { /* 省略 */ }
```

4. **[計算式による実引数・浮動小数点の誤差（☞3.2 節、A.7.2 項）]**
   a, b を適当に定めて $f(x) = ax^2 + bx$ を求める関数を作れ。微小な定数 ϵ をマクロ EPS で定め、$g(x) = \dfrac{f(x+\epsilon) - f(x)}{\epsilon}$ と $h(x) = \dfrac{g(x+\epsilon) - g(x)}{\epsilon}$ を求める関数を作れ。main() 関数で、p を適当に定めて $g(p), h(p)$ を表示し、手計算で求めた微係数の理論値 $f'(p), f''(p)$ と比較せよ。また、EPS を調整すると何が起こるか観察せよ[*10]。
5. **[関数の組合せ（☞3.3.1 項）]**
   標準ライブラリの数学関数 <math.h> に、2 つの double の値 a, b の大きい方を返す関数 fmax(a, b) がある。これを用いて、3 つの double 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。
6. **[関数の役割分担・座標の平行移動（☞3.3 節）]**

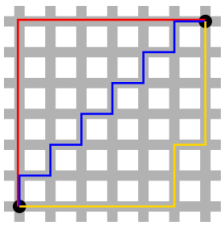
[*9] Java ならコンパイルエラーになる。（しかし C 言語では、例えば gcc では警告にもならない。）
[*10] EPS を小さくしすぎると、有効精度におさまりきらず、かえって精度が落ちることに注意せよ。

次の関数を作れ。引数はすべて double 型とする。呼び出してよい数学関数は、sqrt(x) か hypot(x,y) のどちらか片方だけ、3 つの関数を通じて 1 ヶ所とする。

- double norm(x, y) は、原点と点 (x, y) の間の距離 $\sqrt{x^2+y^2}$ 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**三角形の周囲の長さ**（3 辺の長さの合計）を返す。

7. **[スコープ（☞3.4 節）]**
ソースコード 3.6 の「printf("s=%g\n", s);」を挿入できる行の範囲を述べよ。

8. **[関数の役割分担・void 関数（☞3.3 節、3.5 節、3.6 節）]**
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 $|x_2 - x_1| + |y_2 - y_1|$ と表される。（道順は、右図のように何通りもある。）以下の関数を作れ。



- int mdist(int x1, int y1, int x2, int y2) は $(x_1, y_1) - (x_2, y_2)$ のマンハッタン距離を返す。
- int sum_mdist(int x, int y) は $(0, 0) - (x, y)$ と $(x, y) - (10, 10)$ の、それぞれのマンハッタン距離の合計を返す。
- void print_sum_mdist(int x, int y) は sum_mdist(x,y) の値を表示する。

いくつかの x, y について sum_mdist(x,y) の値を調査し、気のついた顕著な性質を 1 つ述べよ。 <stdlib.h> の絶対値関数 int abs(int j) を用いてよい。

9. **[関数の役割分担・逆関数の作成（☞3.3 節、3.9 節）]**
次の関数を作れ。（標準ライブラリの三角関数については B.7 節を参照し、角度をラジアンで扱うことに注意せよ。180（度）が π（ラジアン）に対応する。）

- 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。

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、典型的には <math.h> でマクロ定数 M_PI が定義される[*11]。

- double sin_deg(double deg) は deg（度）の正弦を返す。
- double asin_deg(double x) は x の逆正弦を度数法（度）で返す。

そして、逆の動作をする関数を、片方ずつ別々に作るのに比べ、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。

10. **[機能を独立させる]** ☞12.1.1 項

[*11] Visual C++ 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良いでしょうか。

駐車場の自動精算機は、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");
}
```

条件式は、2 つの値を比較します。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、いずれも **2 文字めが =** です。
- 2 文字の演算子は、間に空白を入れて 1 文字ずつに分割してはいけません。

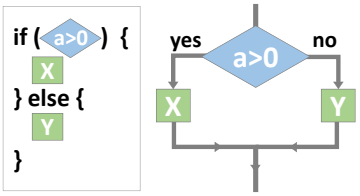
条件式の結果は、**真** (true) と**偽** (false) の 2 通りの値で表されます。比較演算子の大小関係が成立していれば真、そうでなければ偽です。このようなデータ型を、一般に**論理型** (logical datatype) とか**ブーリアン型** (boolean datatype) と呼びます[*2]。

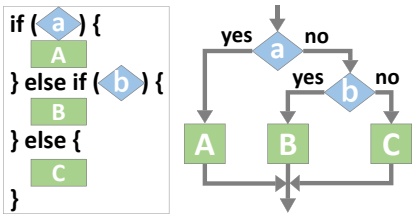
`0 == 0` → 真	`0 != 0` → 偽	`0 =! 0` → 代入（???）
`1 < 5` → 真	`1 >= 5` → 偽	`1 => 5` → コンパイルエラー

数学の表記では、不等号で変数の取りうる範囲を限定したり、場合分けの条件を示すことがあります。しかし C 言語（をはじめとする多くの手続き型言語）の不等号は、その時点の変数の値を用いて、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**するばかりです。「`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ないことに注意してください。

[*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、本物の論理型が導入されたのは C99 のことで、それまでは長らく int を流用してきました。（☞65 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。（最後の 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

2 次元平面上の (x, y) 座標が、**原点である**ことを判定しましょう。条件を、「$x = 0$」で、しかも「$y = 0$」と考えます。

```
/* ifの入れ子 */
if (x == 0) {
    if (y == 0) {
        printf("原点である\n");
    }
}
```

```
/* 論理積 */
if (x == 0 && y == 0) {
    printf("原点である\n");
}
```

- 左上の例では、2 個の if を組み合わせました。このように、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。&& で結ばれた 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**原点ではない**ことはどのように判定すればよいでしょうか。条件は、「x != 0」と「y != 0」の、どちらが成り立っても原点ではありません。

```
/* 多重分岐 */
if (x != 0) {
    printf("原点ではない\n");
} else if (y != 0) {
    printf("原点ではない\n");
}
```

```
/* 論理和 */
if (x != 0 || y != 0) {
    printf("原点ではない\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。

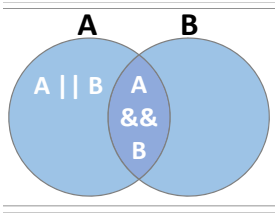
論理否定 (not) 演算子 ! は真偽を反転するので、「〜ではない」の条件を直接的に表現できます。原点の条件の直前に ! を書きます。演算の優先順位が高いので、否定される条件式全体をカッコで囲います。

```
/* 論理否定 */
if (!(x == 0 && y == 0)) {
    printf("原点ではない\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 A	入力 B	演算結果 (積) A && B	演算結果 (和) A || B	演算結果 (否定) !A
真	真	真	真	偽
真	偽	偽		
偽	真			真
偽	偽		偽	



論理演算（積・和・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、コロンで向きを表現していると思えば、納得の文法です。

4.3 論理演算の組合せと優先順位

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。C 言語の比較演算子は 2 つの値だけを比較するので、「$0 \leq x$」と「$x < 10$」に分解した上で、両方とも同時に成立して欲しいので、論理積で連結して `0<=x && x<10` とします。2 つの条件が目立つように `(0<=x) && (x<10)` と、カッコを活用するのもよいでしょう。

頻出ミス

`0 <= x < 10` は、エラーにならずに[a]、動作だけがおかしくなります。気づきにくい間違いですが、各条件をカッコで囲えばこの間違いを防げます。

```
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<=x)<10` と同じです。`(0<=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` を判定することになり、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。

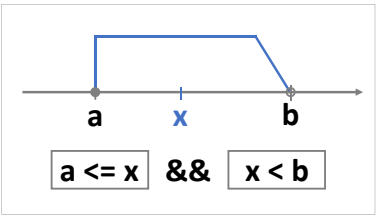
```
if ((x % 2 == 0) ||  (0 <= x) && (x < 10) ) {...} // && が || より優先
if ((x % 2 == 0) || ((0 <= x) && (x < 10))) {...} // 同じ動作（推奨）
```

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 <= x) && ...`」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。




```
if ( !(条件) ) {
  成立時;
} else {
  不成立時;
}
```

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。

```
!(A && B)  ⟺  (!A) || (!B)
!(A || B)  ⟺  (!A) && (!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れば同じ、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`」とすぐにわかります。

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意味です。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 < と <= 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、同じ意味の条件式を（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、**プログラムの制御構造**が理解しにくくなります。else を適切に用いて、制御構造を簡潔に表現しましょう。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**で表現できます。条件式を吟味して、ようやく動きがわかるようではバグの温床になってしまいます。

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);
}
```

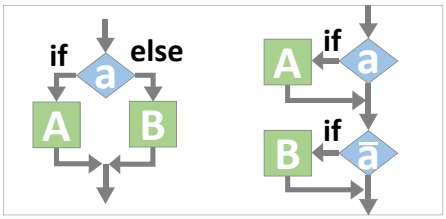
- 左上の if-else の例では、正ならば x、そうでなければ $-x$ を表示することで、絶対値を実現しています。2 つの printf() は else で振り分けられているので、どちらか片方が必ず実行されることがわかります。つまり、（何かしらの）値が 1 回だけ表示されることが、**プログラムの制御構造**から読み取れます。
- 右上の例は、else を使わずに 2 つの if を並べました。しかし問題が 2 つあります。
  (a) どちらか片方の printf() が必ず実行されるとは、**プログラムの制御構造**からは読み取れないので、動作を理解するには、条件式を吟味する必要があります。
  (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_if_else(x)` はもちろん正しく動作します。

`abs_if_if(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によっては警告してくれる場合はあります）、return に到達しないときの戻り値は**不定**です。（初期化忘れの変数と同じです。）



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、**プログラムの制御構造**からは判断できないからです。

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

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

ちなみに右上の例では、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 */   // ×失敗
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 3 通りに分類してみましょう。以下の 3 つのプログラムは、どれも同じ動作をしますが、どれがよいでしょうか。

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)

if の羅列は、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。「成立する if は最大でもひとつ」であることが**プログラムの制御構造**からは読み取れません。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 ヶ所の数値**を直す必要があるため、どちらかを直し忘れそうです。

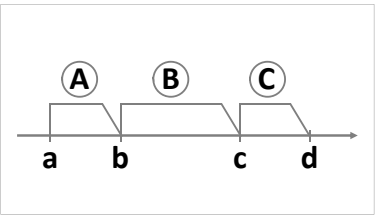
```
/* ifの羅列 */
if (12 <= a && a < 15) {
    printf("中学生");
}
if ( 9 <= a && a < 12) {
    printf("小学生高学年");
}
if ( 6 <= a && a < 9) {
    printf("小学生低学年");
}
```

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。そして、else でつながっていることから「成立する if は最大でもひとつ」という構造も（何とか）読み解けます。（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。）しかし、**インデントが深く**なり、見通しが悪くなりました。分類が増えれば、インデントは更に深くなります。

```
/* 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");
         }
      }
   }
}
```

多重分岐がこの場面の最適手法でしょう。`else if` の繰り返しでインデントの深さが 1 段に揃います。**同じ構文の繰り返し**であることがすぐにわかりますし、分類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。書き忘れでないことを示すために、コメントを残しておきました。

```
/* 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、{ と } を省略して、単文にできますが、お薦めはしません。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構造をおかしくする、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**ことを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[a]、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い出せるでしょう。

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[b]。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。この if は、else 節なのにブロックにしない、例外とも言えます。

[a] Perl のように、そもそも単文が許されない言語もあります。
[b] 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。

4.5　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。

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

```
double 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違和感のある人もいるでしょう。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わかります。**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。

```
/* 直接の条件式 */

if (x > 0) {
    printf("xは正です\n");
}
```

```
/* 変数経由の条件式 */
int c = (x > 0);
if (c) {
    printf("xは正です\n");
}
```

関数でも、変数と同じように、条件式を返せます。return に書いた式が、そのまま関数呼び出し部分に置き換わると理解しましょう。やはり右の `is_plus()` 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。

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_plus(double x) {
    return (x > 0);
}

if (is_plus(x)) { // (x>0) と同じ
    printf("xは正です\n");
}
```

```
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) のような名前にします。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。(☞4.5.2 項)

[*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。標準ライブラリ関数は、昔の名残りで、極端に文字数を節約しています。

偽	FALSE(=0)
真	TRUE(=1), 2, 3, ...

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。（**注意：比較には用いません**。）

```
#define FALSE 0
#define TRUE  1
```

```
int is_plus2(int x) {
    if (x > 0) { // 正なら表示する
        printf("正です\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 bool 型を紹介していません。int 型を流用する関数はいくつもあるので、この説明は外せません。加えて <stdbool.h> ヘッダの読み込みなど、ture false キーワードの互換性維持の仕掛け（☞B.10 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、57 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。なぜなら、int には TRUE(=1), FALSE(=0) 以外にもとりうる値がたくさんあって、条件式で **0 以外はすべて真**と扱われることに決まっているからです。TRUE 一つだけと比較したのでは網羅しきれません。逆に、偽であることを判定したければ、（確かに FALSE と比較しても正しく動作するのですが）直前に否定演算子！を置いて真偽を反転させるのが慣用句です。

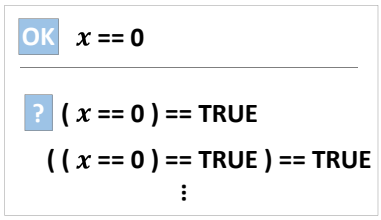

```
if (isalpha(c)) {   // ○ 真の判定
    printf("英字です\n");
}

if (! isalpha(c)) { // ○ 偽の判定
    printf("英字ではない\n");
}
```

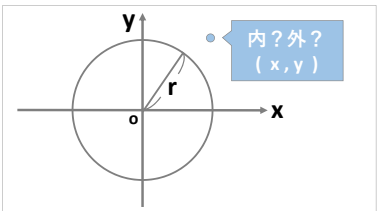
```
if (isalpha(c) == TRUE) {  // ×
    printf("英字です\n");
} // ↑どんなcでも実行されない

if (isalpha(c) == FALSE) { // △
    printf("英字ではない\n");
} // ↑動作は正しい（非推奨）
```

<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。上記の用例は、左側が正しいです。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) {          // 以下のものが真になる
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 }
```





ソースコード 4.2 は、円の内側判定を行います。原点を中心とする半径 r の円が、点 (x, y) を含むなら、関数 `is_in_circle(r, x, y)` は TRUE を返します。このプログラムでは **TURE や FALSE をマクロ定義していない**ことに注目してください。

ソースコード 4.2　円の内側判定

```
1  #include <stdio.h>
2
3  /* 原点が中心で半径rの円が、点(x,y)を含むなら論理型のTRUE、
4     そうでなければFALSEを返す */
5  int is_in_circle(double r, double x, double y) {
6      return (x * x + y * y <= r * r);
7  }
8
9  void print_is_in_circle(double r, double x, double y) {
10     printf("原点を中心とする半径 %g の円は、点(%g,%g)を", r, x, y);
11     if (is_in_circle(r,x,y)) {
12         printf("含む\n");
13     } else {
14         printf("含まない\n");
15     }
16 }
17
18 int main(void) {
19     print_is_in_circle(1.4143, 1.0, 1.0); // 含む
20     print_is_in_circle(1.4142, 1.0, 1.0); // 含まない
21     return 0;
22 }
```

コラム： 自分に厳しく、他人に優しく

TRUE は「1」と定義するのが慣用句ですが、「2」と定義しても構いません。（正確に言うと、これで動作が変わるなら TRUE の**使い方が間違って**います。）

C コンパイラは、比較演算で論理値を得るときには、自分に厳しく 0 / 1 に限定しますが、if で評価する際には、他人に優しく **0 以外はすべて真**だと受け入れます。真を 2 で示す、甘えた (?) 変数や関数も許容されます。

4.6　論理積と論理和の短絡評価（発展的内容）

整数 m, n (≥ 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 は実行されないので、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るのでしょうか。

C 言語の論理和と論理積は、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。具体的には次のような動作をします。

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、例えば「倍数ではありません」を表示させる改造も簡単です。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、一体化しすぎてて「倍数ではありません」を表示させるには手間がかかります。

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。

FALSE && B ⇔ FALSE
TRUE && B ⇔ B
TRUE || B ⇔ TRUE
FALSE || B ⇔ B

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 & あるいは || を | と書き間違えると、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

```
#include <stdio.h>

void print_multi_nest(int m, int n) { // (A) if の入れ子
    if (n != 0) {
        if (m % n == 0) {
            printf("%d は %d の倍数です\n", m, n);
        }
    } else { // n == 0 の場合
        if (m == 0) { // 0の倍数は0だけ
            printf("%d は %d の倍数です\n", m, n);
        }
    }
}

void print_multi_and(int m, int n) { // (B) 論理積
    if (n != 0 && m % n == 0) {
        printf("%d は %d の倍数です\n", m, n);
    }
    if (n == 0 && m == 0) { // 0の倍数は0だけ
        printf("%d は %d の倍数です\n", m, n);
    }
}

int main(void) {
    print_multi_nest(10, 5); print_multi_and(10, 5);
    print_multi_nest(2, 3);  print_multi_and(2, 3);
    print_multi_nest(7, 0);  print_multi_and(7, 0);
    print_multi_nest(0, 0);  print_multi_and(0, 0);
    return 0;
}
```

コラム： ブロックのスタイル

ブロックのスタイルには、いくつもの流儀があります。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、下の abs1() の例のように、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、失敗しています。初学者だけでなく、指導役の**熟練者までもが間違いを見抜けず**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れば、すぐにわかるでしょう。

```
int abs3(int x) // ×絶対値失敗
{
    if (x < 0);
    {
        x = -x;
    }
    return x;
}
```

4.7　練習問題

指示された関数以外にも、main() を作って動作を確かめよ。

1. **[1 要素による分岐（☞4.1 節）]**
   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% 割り引かれて 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。

   ヒント：複数の if を用いると、return に到達しないことがある。（☞4.4.2 項）

2. **[2 要素による分岐（☞4.2 節）]**
   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、a 才の子供に n 人の大人が付き添っている（$a \geq 0, n \geq 0$）。入場できるかどうかを、if と else と論理和（または論理積）を各 1 回ずつ用いて判定して表示せよ。論理積と論理和は、どちらか片方だけ用いてよい。

n人＼a才	0 ～ 5	6 ～
0	×	?
1 ～	?	○

   ヒント：「?」を埋めよ。

3. **[多重分岐（☞4.1.1 項、4.2 節）]**
   点 (x, y) を、「原点」「X 軸上」「Y 軸上」「軸上ではない」の 4 通りに分類して表示せよ。if と else を各 3 回ずつ用いよ。「X 軸上」「Y 軸上」には、原点を含めないものとする。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。

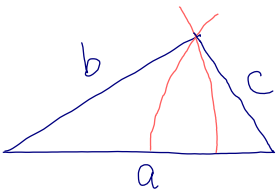
y＼x	x == 0	x != 0
y == 0	原点	?
y != 0	y軸上	?

   ヒント：「?」を埋めよ。

4. **[多重分岐（☞4.4.4 項）]**
   年度初めの年齢 a 才をもとに、学年を表示する次の関数を作れ。
   - `void print_grade(int a)` は、6 才以上なら「小学 x 年生」、12 才以上なら「中学 y 年生」、15 才以上なら「高校 z 年生」と表示する。学年の範囲は、小学生を $1 \leq x \leq 6$、中学・高校を $1 \leq y, z \leq 3$ とし、これ以外では何も表示しない。

   main() で 5, 6, 11, 12, 14, 15, 17, 18 才の動作を確かめよ。（☞72 ページのコラム）

5. **[論理型の関数（☞4.5.2 項、ソースコード 4.2）]**
   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。簡単のため $a, b, c > 0$ を前提とする。



   - `int exist_triangle(double a, double b, double c)` は、3 辺の長さが a, b, c の三角形が存在すれば論理型の TRUE、そうでなければ FALSE を返す。

   ヒント：return の式を工夫すれば、TRUE と FALSE をマクロ定義する必要はない。

6. **[0 以外]**

x 円を何枚かの硬貨で払いたい。次の関数を作り、main() から繰り返し呼び出すことで、右の**実行結果**を実現せよ。

- `void print_coins(int x)` は、合計が x 円になる硬貨の、各枚数を表示する。
  - x は 10 以上 990 以下の 10 の倍数であることを前提とする（呼び出し側の条件）
  - 使用する硬貨は 500 円，100 円，50 円，10 円の 4 種類
  - 各硬貨の枚数は十分にある
  - 硬貨の合計枚数が最小となる払い方を選ぶ
  - 0 枚のものは表示しない

実行結果

```
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
```

（ヒント）

- 高額の硬貨から枚数を決定すれば、自動的に合計枚数が最小になる。
- ある硬貨の枚数を決定したら、x からその金額を引く。残った x 円を、次の硬貨で支払うことにする。これを、4 通りの硬貨で繰り返せばよい。
- 硬貨の金額を変数にすると、4 回の処理が（金額を除いて）完全に同じになる。

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」のような間違いを疑うと効率的です[a]。

6. の問題では、例えば 490 円と 500 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おきましょう。

[a] これには前提があって、例えば「小学 x 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、同じように精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、繰り返し実行する処理は、while 文で次のように書けます。

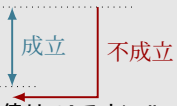
```
/* 文法 */
while (条件式) {
    文;
    ...
}
```

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


if とよくて似ていて、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、ブロックを何度で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、条件を検査します。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。

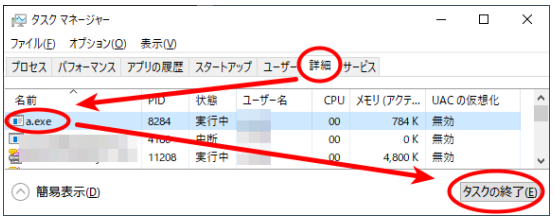


図 5.1　Windows のタスクの終了方法

次に「x を越える最小の自乗数」を求める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、$k = 1, 2, 3, \ldots$ 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 }
```

5

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。Cygwin を含む 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 がおかしなエラー（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。

5.1.1　ループ変数

プログラムでは、同じ処理を**10 回繰り返す**という場面がよくあります。これを while で実現すると、右のようになります。

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

変数 i の働きに着目してください。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。このため、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目**を担っているので、このループの**ループ変数** (loop variable) と呼ばれます。

5.2　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。右のように**ループ変数が 3 回まとまって出現**して、実用上はこちらがよく使われます。

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

for 文の文法は、以下の通りです。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;
}
```

5.2.1　for と while の使い分け

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。

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

[*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。

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 には、増分式をループ処理の最後に書くことになります。処理部分が長くなると、ループ変数の登場する 3 つの式が離れてしまうので、書き忘たり変数名を間違えたりしがちです。ですから**回数の決まっているループでは 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ないとか、ループ回数が事前に決まってい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。`l`（エル）は `1`（いち）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、同じ働きをします。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。

```
// 増加が前
y = ++x;
  ↕
++x;
y = x;
```

```
// 増加が後
y = x++;
  ↕
y = x;
x++;
```

しかし C 言語には、評価順序に動作保証のない場面も多くあります。2 項演算子の 2 つのオペランドや、関数の引数[a]などがそうです。このような場面では処理系依存になるので、単純な使い方にとどめておくのがよいでしょう。

```
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"
```

[a] 関数の引数の区切りのコンマは、コンマ演算子（順次評価）ではありません。

5.3　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。

5.3.1　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。

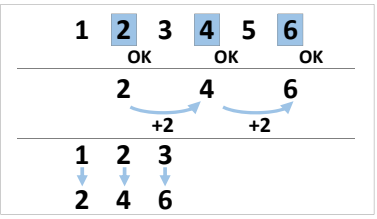
ソースコード 5.3　偶数を表示する

```
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と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) {         // 奇数なら
            i++;                  // 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」という特徴が台無しです。

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(にら)めっこしていても埒(らち)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。

5.3.2　個数を数える

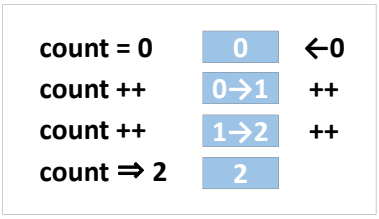
整数 n の**約数の個数**を数えてみます。n の約数とは、1 から n の間の整数のうち、n を割り切るものですから、1 から始まる n 回のループの中で、それぞれ割った余りを計算することで約数かどうかがわかります。約数を見つけるたびに 1, 2, 3, . . . とカウントアップしていきましょう[*2]。ループ回数が明確に決まっているので for を用いることにして、ループ変数は i にします。そして、**個数を数えるための変数**が別に必要になります。

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 $n = 2^a 3^b 5^c \ldots$ から $(a+1)(b+1)(c+1)\ldots$ 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。



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」を表していると、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が、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**することも工夫のひとつです。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ましょう。

[*3] 英文法的には 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。

5.3.3　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。

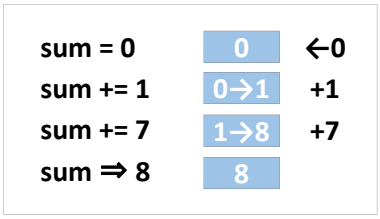
ソースコード 5.5　約数の和

```
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。約数がみつかると、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
```



実行結果は、「和は 24 です」だけでは正しいかどうかがわかりにくいので、8 行目で途中の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、どちらかというと、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目印でもあります。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。

5.3.4　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。

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

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。ルート演算には <math.h> の sqrt() を使います。

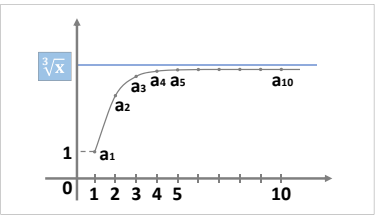
(A) cbrt4() では、電卓での a_4 の計算をそっくり再現しました。
ところで、変数 a3 や a4 の役目をよくよく考えると、a4 を求めるのに必要なのは 1 つ前の a3 の項のみで、それより前は不要です。それならば、異なる変数を用意する必要はなく、すべて同じ 1 つの変数を使いまわしできそうです。

(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 と言わずに a_{10} を求めてみましょう。このように、**回数の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 $|a^3 - x| > \mathrm{EPS}$ の間にループします。

実行結果には、<math.h> の本物の 3 乗根関数 cbrt() 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。

[*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ので、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

```
#include <stdio.h>
#include <math.h>
                    // 以下は電卓での操作（[MR]はメモリ呼び出し）
double cbrt4(double x) { // (A)      // xをメモリに記憶
    double a1 = 1.0;                // [1]
    double a2 = sqrt(sqrt(x * a1)); // [×][MR][=][√][√]
    double a3 = sqrt(sqrt(x * a2)); // [×][MR][=][√][√]
    double a4 = sqrt(sqrt(x * a3)); // [×][MR][=][√][√]
    return a4;
} // よく考えるとa1,a2,a3,a4は同じ変数でもよい

double cbrt10(double x) { // (B)     // xをメモリに記憶
    double a = 1.0; // 初項a1       // [1]
    for (int i=2; i<=10; i++) {
        a = sqrt(sqrt(x * a));      // [×][MR][=][√][√]
    }
    return a;
}

#define EPS 0.001   // 許容誤差
double cbrt_eps(double x) { // (C)
    double a = 1.0;
    while (fabs(a*a*a - x) > EPS) {  // aの3乗とxの差の絶対値を比較
        a = sqrt(sqrt(x * a));
    }
    return a;
}

int main(void) {
    double x = 8.0;
    printf("cbrt=%f, cbrt4=%f, cbrt10=%f, cbrt_eps=%f\n",
            cbrt(x), cbrt4(x), cbrt10(x), cbrt_eps(x));
    return 0;
}
```

ソースコード 5.6 の実行結果

```
cbrt=2.000000, cbrt4=1.978456, cbrt10=1.999995, cbrt_eps=1.999979
```

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 `cbrt4()` 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、`cbrt_eps()` のような動作ができるようになりました。

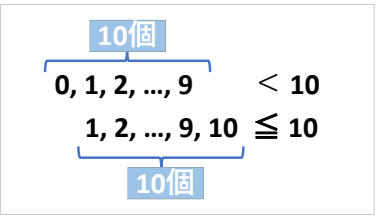
現在のほとんどのプログラミング言語は、この系譜に則っています。「プログラム上に現在実行中のところがあって、だんだんと進んでいく」という考え方は、ここからきています。

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**といいますが、英語圏では 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 言語では配列（☞8 章）を 0 から数え始めるため、0 オリジンがよく出てきますし、「何番目」かを計算する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すが、日常生活の 1 オリジンとギャップがあるため、この違いを埋め合わせるのに苦労するのは、プログラマの宿命でしょう。



5.4　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、これでは読み手に誤解を与えます。

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（ゼロ）オリジン）と、1 から始める（1（いち）オリジン）のどちらにするかは、場面によって判断します。このあとに出てくる配列を扱うなら 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。

5.5 練習問題

1. **[ループの実行順序（☞5.1 節）]**
   74 ページのソースコード 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. **[条件を満たすものを表示する（☞5.3.1 項）]**
   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。
3. **[条件を満たすものを表示する（☞5.3.1 項）]**
   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 の現れるものを、昇順に「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。（☞4.2 節）
4. **[合計を計算する（☞5.3.3 項）]**
   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。
5. **[デバッグライト（☞83 ページのコラム、5.3.4 項）]**
   85 ページのソースコード 5.6 の cbrt10() と cbrt_eps() 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() のループ回数も確かめてみよ。
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。82 ページのソースコード 5.5 の sum_divisor() を利用（または改造）して、10000 以下の完全数（4 個ある）をすべて求めよ。

   ＿＿＿＿ 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・double へのキャスト]**
   S_n $(n = 0, 1, 2, \ldots)$ を以下で定義する。

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

   $n = 0, 1, 2, \ldots, 100$ 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 $4 \cdot S_\infty$ は π（円周率）に収束すると知られている。

   ＿＿＿＿ $(-1)^k$ 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、ループするごとに `sign = -sign;` と符号を反転させてもよい。＿＿＿＿＿＿

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);
    }
}
```

入れ子になった 2 つのループを **2 重ループ** (nested loops) 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。

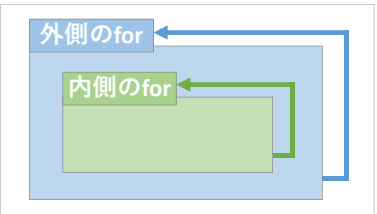
```
int i; // 共通のループ変数!?
for (i=1; i<=3; i++) {      // 外側のループ (1,2,3)
    for (i=5; i<=7; i++) { // 内側のループ (5,6,7)
        printf("i=%d, i=%d\n", i, i);
    }
}
```

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
    ...;
}
```


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（☞43 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、ぜひ活用しましょう。

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** (multiple nested loops) 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くなれば、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 重ループにあたるところ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。

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。
break 文 ループを抜け出します。

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
}
```


continue も break 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考え方が主流です。使用は、例外的な事象に、**最小限に留める**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。

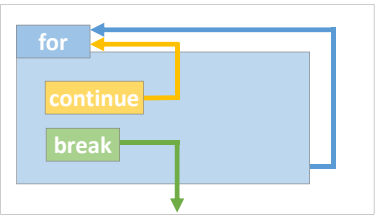
```
for (;;) {
    ...;
}
```

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

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

for 条件式（カッコ内の 2 番目）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、カッコ内のほかの式も不要になるので、通常は式をどれも省略して、**セミコロン 2 個だけ**を残します。

while 何かしらの条件式を書く必要があるので、**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、左のようにすると 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ってしまいましたが、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、書き忘れでは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 を使いたいので、k のスコープを広げるために、for ループの前で定義しています。

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、ループの処理部分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 はたまに目にする、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 = をつけるかどうかがわかりにくいので、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から、多用すべき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きます。ただし、変数定義を for のカッコ内で行うと、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で、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で、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（と思われている）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 → 4 → 2 → 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。

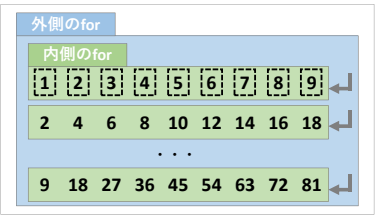
ソースコード 6.2　コラッツ予想

```
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 }
```

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。

[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、角谷（かくたに）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ので、**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 回実行されます。6 行目の printf は内側のループに入っているので、**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。

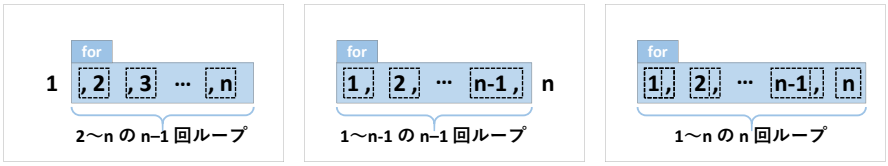
ソースコード 6.4　1 から n までの数をコンマ区切りで表示

```
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**（☞5.4 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、この表記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ってなかったとします。正しい値を入力してもらうよう、再入力を促しましょう。

A.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、また入力の場面に戻るのですが、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 =>
```

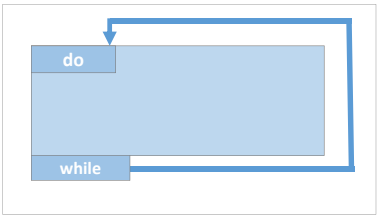

親切になったのですが、このプログラムには欠点があります。上限値の 10 を変更したくなったら、注意深く 2 ヵ所の条件式（に加えてメッセージの内容）を書き直す必要があります。つまり**メッセージを表示する条件**と、**ループを継続する条件**が別々になっています。これは崖っぷちでバランスをとってるようなもので、バグの温床になります。

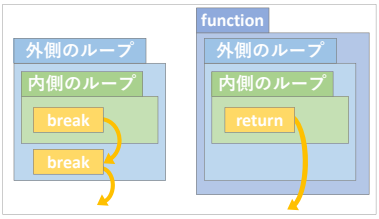
それでは、メッセージ表示とループ継続が、必ず**セットになる条件分岐**にしましょう。こうなると、もう do-while は使えず、無限ループの出番です。以下のようにすると、どちらの if が成立しても、メッセージを表示した上で break しません。

```
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、正しい入力のときに 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。

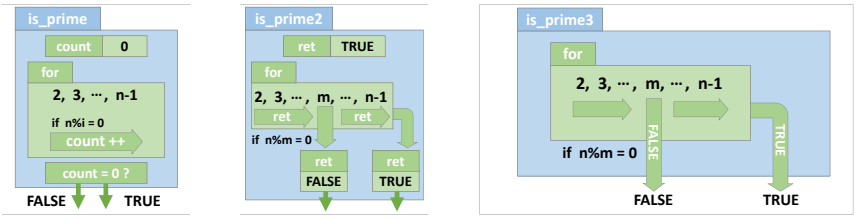
6.4.5　素数判定

与えられた 2 以上の整数 *n* が**素数** (prime number) かどうかを判定してみましょう。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、**下線部は共通**なので、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(>=2)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) {
9              count++;               // 約数が見つかったら1増やす
10         }
11     }
12     if (count == 0) return TRUE; // 約数が0個なら素数
13     else            return FALSE;
14 }
15
16 int is_prime2(int n) {            // (B)
17     int ret = TRUE;                // 素数であることを前提にスタート
18     for (int i=2; i<=n-1; i++) {
19         if (n % i == 0) {
20             ret = FALSE;           // 約数が見つかったら素数ではない
21             break;                 // ループを続ける必要もない
22         }
23     }
24     return ret;
25 }
26
27 int is_prime3(int n) {            // (C)
28     for (int i=2; i<=n-1; i++) {
29         if (n % i == 0) {
30             return FALSE;          // 約数が見つかったら偽を返す
31         }
32     }
33     return TRUE;          // 最後まで約数が見つからなければ素数
34 }
```



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ぬ証拠になります。

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（見つかれば）「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。そしてループ中では、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、定義どおりに 2 から $(n-1)$ までのループ処理で、約数の個数を数えてみます（7–11 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（12 行目）。そうでなければ FALSE を返します（13 行目）。

(B) しかしよく考えると、約数が 1 つでも見つかると、もっとたくさん見つかっても、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。（17 行目）。約数が見つかると ret を FALSE にします（20 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（24 行目）。

ところで、ループ処理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、**FALSE は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同じです（21 行目）。これで計算時間を節約できます。

(C) ここまで、ループ処理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（30 行目）、ループはもちろん関数ごと抜け出すことにします。**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（33 行目）。

6.5 練習問題

1. **[3 重ループ or 1 重ループ + 桁分解（☞6.1 節または 2 章の練習問題 6.）]**
   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。
2. **[逆ループ or 計算（☞6.3.3 項）]**
   97 ページのソースコード 6.3 の九九の表を、上下反転して表示せよ。
3. **[2 重ループ、回数の変化するループ（☞6.4.1 項）]**
   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。

```
*****
*****
*****
*****
*****
```

```
*
**
***
****
*****
```

4. **[例外処理]**
   98 ページのソースコード 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。
5. **[複雑なループ]**
   85 ページのソースコード 5.6 の cbrt_eps() におけるループ回数に上限を設けよ。
   (案 1) break を使用する (案 2) while の条件を厳しくする (案 3) for に置き換える
6. **[ある?ない?]**
   102 ページのソースコード 6.5 の is_prime3() 関数を利用して、次の関数を作れ。
   - `int is_sum_of_2primes(int n)` は、n が 2 つの素数の和で表せれば論理型の TRUE、そうでなければ FALSE を返す。（例：6 は 3+3 と表せるので TRUE）

   そして main() で、2 から 100 までの整数のうち、素数でもなく、2 つの素数の和でも表せないものを、すべて求めて表示せよ。（27 から 95 までに、9 個見つかる。）
   ヒント：n が素数でないことは「`if (!is_prime3(n))`」と検査する。（☞4.5.2 項）
7. **[数列の和]**
   以下の数列 $\{a_k\}$ を、a_k と a_{k-1} の関係に着目して $a_0, a_1, \ldots, a_9$ を求め、表示せよ。

$$a_k = \frac{1}{k!} \qquad (k = 0, 1, 2, \ldots)$$

   そして、数列 $\{a_k\}$ の和 S_n を以下で定義する。$S_0, S_1, \ldots, S_9$ も同時に表示せよ。

$$S_n = \sum_{k=0}^{n} a_k = \sum_{k=0}^{n} \frac{1}{k!} = \frac{1}{0!} + \frac{1}{1!} + \frac{1}{2!} + \ldots + \frac{1}{n!} \qquad (n = 0, 1, 2, \ldots)$$

   - $0! = 1$ である。計算方法によっては $k = 0$ に例外処理が現れる。
   - S_∞ はネイピア数 e (= 2.71828 $\cdots$) に収束すると知られている。
   - 書式文字列には`"%.10f"`などを指定して、表示桁数を増やすとよい。
8. **[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      ┐ 1 つの警告
     printf("正方形の面積は%fです。\n", square(1.0));      │ メッセージ
     ^                                              ┘
     ← 5 文字目の位置の目印

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      ┐ 1 つの
 double square(double x) { return x * x; }          │ エラー
        ^ ← 8 文字目の位置の目印                        │ 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、ということを**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。コンパイラは、関数呼び出しが正当であるかを検査して、例えば引数の個数が違っていればコンパイルエラーにしてくれますが、この検査に関数プロトタイプが必要です。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、後に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]が設けられています。

この順序制約を実質的に取り払う方法があります。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、違いは関数本体のブロックを**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、テキストエディタで完成している**関数の先頭部分をコピー**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で置き換えます。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、時代を感じずにはいられません。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こるのでしょうか。106 ページのソースコード 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。square() が double 型を返すので、暗黙的な宣言の int 型と食い違ったということです。

しかし、運悪くプロトタイプが矛盾しなければ、この段階は警告にもなりませんし、全体を通じてエラーが起こりません。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、簡単に起こることです。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 世代め (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 世代めの 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**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
```

しかし、このように引数名を省略したり、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マの役に立ちそ

[*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。

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

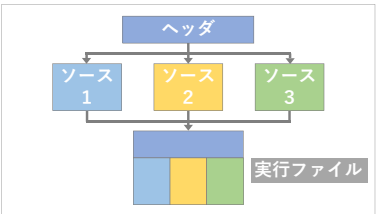
プロトタイプ宣言を書くには、テキストエディタで関数定義の**最初の 1 行**をコピーします。この作業に、マウス操作の得意 (?) な初学者が、**1 文字単位**で時間をかけて選択しているのをよく見かけますが、見ているだけでまだるっこしいです。

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 行の先頭から末尾まで）**行選択**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。書かずにすますこともできるプロトタイプ宣言で、わざわざ険しい道を行くわけです。

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」と、操作ミスを元に戻す「アンドゥ」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。Home と End の行頭・行末ジャンプも重要です。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に書いてきました。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が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、3.4 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（トップレベル）	プログラム全体	グローバルスコープ
static グローバル		1 つのソースファイル	ファイルスコープ
ローカル, static ローカル	関数内, 仮引数	1 つの関数	関数スコープ
	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

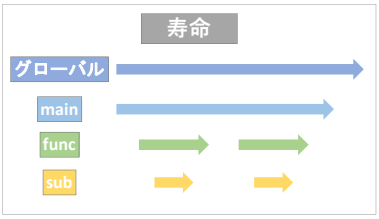
ここでは、プログラムの実行する部分が進むのに応じて、変数がどのタイミングで必要になり、どのタイミングで不要になるのかを見ていきます。つまり、変数の誕生から消滅までの、**寿命** (lifetime) を考えてみます。

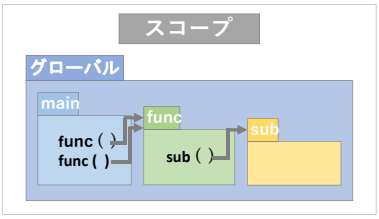
ローカル　ローカル変数は、定義されている**ブロックを実行**するときに誕生します。（記憶するための領域が確保されます。）そして、その**ブロックを抜け出す**ときに、不要になるので消滅します。（領域が開放されます。）ブロックをもう一度実行しなおせば、ローカル変数はまた新たに作り直されます。今まで気にすることはなかったかもしれませんが、このような動きになっていました。そして初期化は自動では行われず、プログラムで代入するまで、どんな値になっているのか保証がありません。

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、次回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で、**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（定義と同時に行う代入）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のものを関数スコープ、さらに内側のブロックなら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呼びます。）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。）変数名が重複しないように気を配る範囲が、1 つのファイルだけでよくなります。

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て、通常の関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、定義したところから、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、文脈によって意味の変わることがよくあります。static も void に並んで意味のバリエーションの多いキーワードです。

[*10] 普通の関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。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があります。

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。

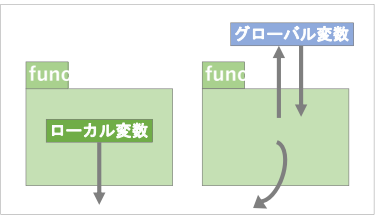
(B) `call_num_global()` ではグローバル変数を使いました。グローバル変数はスコープが広いので、この変数を使う関数名を接頭辞にして、長い変数名にしてみました。初期化（0 の代入）が行われるのはプログラム開始直後の 1 回だけで、関数の処理が終わっても**値を保持し続けて**、望みの動作をしてくれます。ひとまず目標達成です。しかし、変数名をいくら長くしたところで、プログラムの別の場所から値を（意図せず）変更してしまう危険が付きまといます。

(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。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のある変数なら、この方法を採用します。

ソースコード 7.3 の実行結果

```
local=1, global=1, static_local=1, static_global=1
local=1, global=2, static_local=2, static_global=2
local=1, global=3, static_local=3, static_global=3
local=1, global=4, static_local=4, static_global=4
local=1, global=5, static_local=5, static_global=5
```



ソースコード 7.3　呼び出された回数を数える関数

```c
#include <stdio.h>

int call_num_local(void) { // (A) 失敗
    int n = 0; // ローカル変数（短い変数名）
    n++;
    return n; // 1, 1, 1, ...
}

int call_num_global_counter = 0; // グローバル変数（長い変数名）
int call_num_global(void) { // (B)
    call_num_global_counter++;
    return call_num_global_counter; // 1, 2, 3, ...
}

int call_num_static_local(void) { // (C)
    static int n = 0; // staticなローカル変数（短い変数名）
    n++;
    return n; // 1, 2, 3, ...
}

static int count = 0; // staticなグローバル変数（少し長い変数名）
int call_num_static_global(void) { // (D)
    count++;
    return count; // 1, 2, 3, ...
}

int main(void) {
    for (int i=0; i<5; i++) {
        printf("local=%d, ",         call_num_local());
        printf("global=%d, ",        call_num_global());
        printf("static_local=%d, ",  call_num_static_local());
        printf("static_global=%d\n", call_num_static_global());
    }
}
```

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。**変数名や引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、一番内側**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[*11]。

```
    void x(void) { printf("x"); } // ここに関数がある(case A,B,C)
    void func(int x) {    // ここに引数があるので
//      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); // "引数の値"
//      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[*12]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。

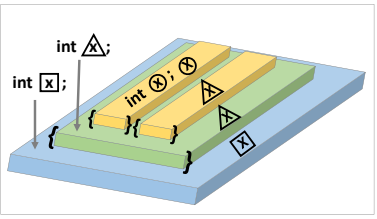
(case A) 同じ名前の関数を 2 つ作ることは（処理が同一でも）できません。

(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。

[*11] 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。

[*12] 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歎してしまいます。

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。コンパイラの設計を楽にするための、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。そのような時のプログラム上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	途中の単語の先頭が大文字
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

変数名や関数名（識別子）には**「名は体を表す」**よう、名前と役目を一致させるのが大前提です。例えば、変数の tmp が幅広くやり取りされるのは、避けるべきです。

命名上の制約もあります。多くのコンピュータ言語では、識別子に**スペースもハイフンも**使えません。そこで、複数の単語の連結方法に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[*13]。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。先頭の 1 文字も大文字にした**アッパーキャメルケース**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。GCD のような元から大文字の単語を使おうとすれば、get_GCD か get_gcd 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。

長い単語の省略パターンとして、initialize を init とするのは業界標準的です。number を num とするのは、本書でも採用していますが、時代とともに省略しない場面も増えてき

[*13] その言語の標準関数の命名規則に影響されているのでしょう。
[*14] オブジェクト指向言語の習慣でもあります。

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

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**否定を含む論理型**の名前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、否定が一度ですみます。

このような習慣は、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. **[仮引数のスコープ（☞3.4 節）]**
   42 ページのソースコード 3.6 について、5 行目の triangle_side() 関数の仮引数を (double x, double y, double z) に変更したとき、動作が変わらないように、最小限の修正を行え。（ただし、4 行目のコメント中の「a,b,c」は「x,y,z」に修正する。）
2. **[ループ vs. 公式、信頼性（☞5.3.3 項、2.6 節、2.9 節）]**
   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)` はループ処理で積算せよ。
   - `int sum2(int n)` は等差数列の和の公式 $\frac{n(n+1)}{2}$ を用いよ。

   この 2 つの関数の値が n の**奇数**のとき、**偶数**のとき、あるいは 50000 のような**大きな値**のときに一致するかを調査せよ。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、信頼性向上にどのように役立つかを考察せよ。
3. **[信頼性]**
   正の整数 x, y の最大公約数を 2 通りの方法で求めたい。（176 ページのソースコード 10.5 を参照してもよい。）

- `int gcd(int x, int y)` は、ユークリッドの互除法で実現せよ。
- `int gcd2(int x, int y)` は、共通する約数の最大値をループ処理で求めよ。

4. **[1 組の値を返すペアの関数（☞117 ページのコラム「複数の値を返す関数?」）]**
   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**した状態で得たい。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 個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 3. で作った関数を用いてよい。
5. **[プロトタイプ宣言、getter/setter（☞7.1 節、7.2.4 項）]**
   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、**目標**を達成するための、プログラム中での **2 つの変数への禁止すべき操作**を述べよ。
   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い。）
   そして、この 4 つの関数の**プロトタイプ宣言**を書き加えよ。
   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。

```c
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 */ }
```

6. **[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、値を保存する関数を setter と呼びます。慣用句としては、この 2 つ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、変数名が違うものであれば、何度も似たような処理を書き並べることになって煩雑です。

駐車場をいくつも管理するなら、駐車場ごとに駐車スペースの数は異なることでしょう。プログラムを駐車場ごとに修正するとしたら、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) と呼びます。[ ] の中に書いた数は**要素数** (number of elements) です。

```
/* 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、有効な添字は 0, 1, 2 です。

```
/* 普通の変数（定義と代入） */
int a0, a1, a2;
a0 = 0;
a1 = 10;
a2 = 20;
```

```
/* 配列変数（定義と代入） */
int a[3];   // [3]は要素数
a[0] = 0;   // [0]は添字
a[1] = 10;  // [1]は添字
a[2] = 20;  // [2]は添字
```

関数内で定義した**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。規則正しい値ならループ処理で代入できます。規則性のないデータを列挙するなら**初期化**（変数定義と同時の代入）が便利です。値を書き並べてブロックで囲います。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（初期化） */
int a0 = 0, a1 = 10, a2 = 20;
```

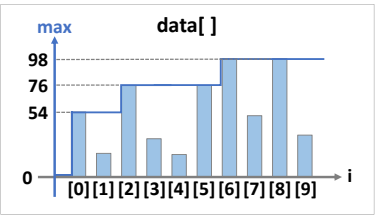
```
/* 配列変数（初期化） */
int a[3] = { 0, 10, 20 };
int b[]  = { 0, 10, 20 }; //省略可
```

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が**ループ処理**で求められます。

```
/* 普通の変数（合計） */
int sum = a0 + a1 + a2;
```

```
/* 配列変数（合計） */
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が、配列なら、ループ回数が違うだけの**同じ処理**ですみます。

頻出ミス

int a[3]; と用意した配列は a[3] の要素を使ってはいけません。a[-1] もダメです。しかし、使ってもコンパイル時には**エラーも警告も出ない**のが普通です。実行時は、動きがおかしくなる可能性があります。（Java なら確実に実行時エラーです。）C 言語では、実行効率を優先し、不正な添字を**確実に検出する手段がありません**。

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**してみます。

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 を暫定最大値とするので、0 で初期化します。最初は 0 でも −100 でも、小さい値なら何でもよくて、max はループ処理中に徐々に大きくなります。

ループ処理では、data[i] を順に調べて、max より大きいときに max を更新します。変数の値の変化をよく理解してください。

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] = 76, max = 76
data[6] = 98, max = 98
data[7] = 51, max = 98
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, 76, 98, 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];   // 暫定最大値を更新する
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 }
```

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 つが食い違うと、プログラムは正しい動作ができません。もっというと、5 行目のデータの個数を数えて、その値と一致させねばなりません。今は絶妙にバランスをとっていますが、将来、ちょっとしたことで破綻するでしょう。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、**意味のわかる名前**をつけます。例えば下の例のように、N_DATA というマクロを定義して、「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, 76, 98, 51, 98, 35 };

for (int i=0; i<N_DATA; i++) { ... }
```

配列要素数は**定数式** (constant expression) （コンパイル時に値の決まる式）で指定する必要があるので[*1]、このように**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。

[*1] 変数のように、実行時まで値の決まらない式でも、可変長配列（☞125 ページのコラム）によってコンパイルエラーにならない場合もありますが、今後のことを考えると、使用は控えたほうがよいでしょう。

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、式の中では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**行末のセミコロン** (;) は、文字の置き換え指示としては正当です。しかし、このマクロが配列の要素数に現れると**コンパイルエラー**です。「']' より前に';' が現れた」のようなメッセージからは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、要素数に**実行時に決まる式（変数）**を指定しても大丈夫です。**可変長配列** (variable-length array) という機能です。

```
/* 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
```

ただし、C11 でオプション機能に格下げになりました。Visual C++ は C99 に準拠しておらず、この機能も意図的にサポートしていません。C++ 規格にも取り込まれず、**セキュリティ上の懸念**もあるため、将来的には消え去る機能でしょう。

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 演算子で、**変数の占める領域の大きさ**（バイト数）がわかります[*2]。「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, 76, 98, 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
};
```

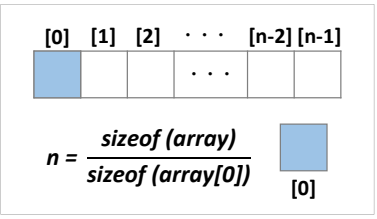
コラム：初期化子の過不足

配列変数の定義で、**要素数**と**初期化子**は、どちらか片方を指定すれば十分ですが、両方とも指定して初期化子が足りなければ、必要なだけ**末尾に 0 が補われます**。ただし、初期化子を空にすることは、言語規格ではなぜか許容されていません[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] 今でも警告しないコンパイラも多く、C23 では言語規格で許容される見込みです。
[b] C 言語の言語規格に強く依存した手法なので、ループ処理で 0 を代入するのもよいでしょう。

[*2] sizeof 演算子は、2.9 節では型に使いましたが、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。
なお、sizeof は size_t 型（符号なし整数、☞9.2 節）なので、演算結果を int 変数に代入する際には、厳密には int へのキャストが必要です。



個数を数えずに、末尾の目印に**番兵** (sentinel) と呼ばれる、データとしては出現するはずのない数値を配置する手法もあります。0 以上のデータを扱っていれば、例えば –1 を末尾に配置して、ループの条件を「データが 0 以上」とするわけです。

```
int data[] = { 54, 20, ,..., 98, 35, -1 };  // 末尾の-1が番兵
for (int i=0; data[i]>=0; i++) { ... }
```

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　受け取り側は、関数の仮引数に、変数名の後に **[要素数]** を書きます。変数定義と似てますが、**要素数を省略**できる場合があります。（☞8.3.1 項）

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。

```
/* 普通の引数 */
void func(int x);

int main(void) {
    int x = 0;
    func(x); // xはint
    ...
```

```
/* 配列の引数 */
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になります。（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
```

8.3.1　配列要素数は管理されない

ここで思い出したいのは、(関数と無関係な) C 言語のそもそもの要素数の扱いです。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**検出機能がなく**、多くの場合コンパイルエラーにも、実行時エラーにもなりませんでした。(☞123 ページの頻出ミス)

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[*3]。そのため、要素数が必要なときの慣用句としては、右の例のように、**別の引数**で渡します[*4]。

```
void func3(int num, int a[]);

int main(void) {
    int a[5] = {0,0,0,0,0};
    func3(5, a);
    ...
```

8.3.2　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、引数や戻り値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。関数が配列を返すには、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、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
}
```

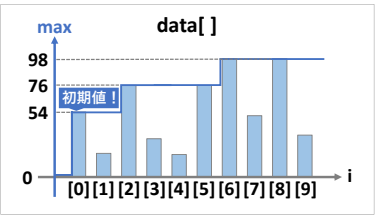
また、3.8.2 項でできなかった、関数でのスワップ（値の入れ替え）が、配列なら可能です。

```
/* 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。

[*3] 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)

[*4] C23 では `func(int num, int a[num]);` のプロトタイプで、a の要素数を num とした不正アクセスの検査をする提案があります。どのように実現されるのかは、まだ不透明です。



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

124 ページのソースコード 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 5 行目の x[0] が不正アクセスになるので、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];       // 暫定最大値を更新する
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         54, 20, 76, 32, 19, 76, 98, 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 }
```

8.4.2　0 始まり・1 始まり

毎月の日数を配列に保存します。（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と、ソースコード 8.3 のように、月の数字[*5]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）

```
#include <stdio.h>

#define N_MONTH 12
int days[N_MONTH] = { /* 0オリジン */
    31, 28, 31, 30, 31, 30, 31, 31, 30, 31,  30,  31
};//1月 2月 3月 4月 5月 6月 7月 8月 9月 10月 11月 12月

int main(void) {
    for (int i=1; i<=N_MONTH; i++) {
        printf("%d月は%d日あります\n", i, days[i-1]); // 調整項あり
    }
}
```

8

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、☞86 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 番めの初期化子は -1 と、明らかに無効な値にしました。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**します。

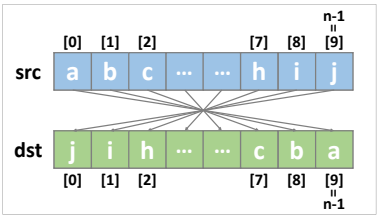
ソースコード 8.4　月ごとの日数（1 始まり）

```
#include <stdio.h>

#define N_MONTH 12
int days[N_MONTH + 1] = { /* 1オリジン */
     -1,   31, 28, 31, 30, 31, 30, 31, 31, 30, 31,  30,  31
};// dummy 1月 2月 3月 4月 5月 6月 7月 8月 9月 10月 11月 12月

int main(void) {
    for (int i=1; i<=N_MONTH; i++) {
        printf("%d月は%d日あります\n", i, days[i]); // 調整項なし
    }
}
```

コメント中の **dummy** は「場所だけ確保して、値に意味のない」くらいの意味のプログラミング用語です。カタカナで「ダミー」とも表記します。

[*5] 英語圏では、月を名前（Jan., Feb., ...）で呼ぶので、0 オリジンで違和感がないようです。そのためか（C 言語に限らず）日付取得のライブラリ関数（☞A.11 節）などで、月は 0 オリジン、日は 1 オリジンと、不統一なことがよくあります。

dst[] の添字			src[] の添字		
i の式	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	$n-3$	$(n-1)-i$
⋮	⋮	⋮	⋮	⋮	⋮
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 の式で表せれば、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、慣用句とも言えます。

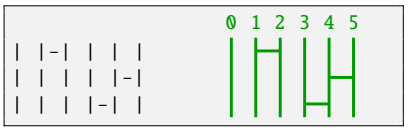
ソースコード 8.5　配列要素のコピー

```
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 #define NUM ((int)(sizeof(orig)/sizeof(orig[0])))
16
17 int main(void) {
18     int orig[] = { 1, 3, 5, 7, 9, 2, 4, 6, 8 };
19     int copy[NUM];     array_copy(NUM, copy, orig);       // コピー
20     int reverse[NUM];  array_reverse(NUM, reverse, orig); // 逆順
21     for (int i=0; i<NUM; i++) {   // 結果の表示
22         printf("%5d %5d %5d\n", orig[i], copy[i], reverse[i]);
23     }
24 }
```

8.4.4　あみだくじ

あみだくじの道を表示するプログラムです。ソースコード 8.6 では、6 人の行き先を示す縦の道を 6 本と、隣の人の入れ替えを示す横の道を 3 ヶ所描きました。

ソースコード 8.6 の実行結果



ソースコード 8.6　あみだくじの道を表示する

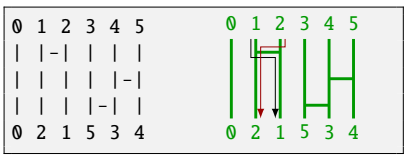
```
1  #include <stdio.h>
2  #define N_PEOPLE 6
3
4  /* 左から pos 番目と pos+1 番目の人の間には - を描く */
5  void bar_print(int pos) {
6      for (int i=0; i<N_PEOPLE; i++) {
7          if (i == pos) { printf("|-"); }
8          else          { printf("| "); }
9      }
10     printf("\n");
11 }
12
13 int main(void) {   // ハードコーディング
14     bar_print(1);  // 左から1番目と2番目の間に横線
15     bar_print(4);  // 左から4番目と5番目の間に横線
16     bar_print(3);  // 左から3番目と4番目の間に横線
17 }
```

横の道は、**1 段につき 1 ヶ所**、**隣同士**の入れ替えに制限したので、この例では 1 – 4 – 3 と簡単に表現できます。(0 オリジンで数えています。) ただし、ハードコーディングになっているので編集には不向きです。次のように、配列とループ処理に置き換えましょう。

```
int bar[] = { 1, 4, 3 };
int n_bar = sizeof(bar)/sizeof(bar[0]);
for (int j=0; j<n_bar; j++) {
    bar_print(bar[j]);  // 左からbar[j]番目とbar[j]+1番目の間に横線
}
```

次は、人の行き先も計算で求めましょう。ソースコード 8.7 に作りかけのプログラムを載せておきます。

ソースコード 8.7 の出力目標



人の並び順を記憶する people[] を用意して、**初期化**する people_init() と、**表示**する people_print() を作っておきました。

隣同士を入れ替える people_swap() の処理を作り、うまく main() から呼び出して完成させてください。

ソースコード 8.7　あみだくじの行き先を表示する（未完成）

```
#include <stdio.h>
#define N_PEOPLE 6
int people[N_PEOPLE]; // 人の並び順を記憶する

/* i 番目の人を i で初期化 */
void people_init() {
    for (int i=0; i<N_PEOPLE; i++) { people[i] = i; }
}

/* 人を表示する */
void people_print() {
    for (int i=0; i<N_PEOPLE; i++) { printf("%d ", people[i]); }
    printf("\n");
}

/* pos 番目と pos+1 番目の人を入れ替える */
void people_swap(int pos) {
    // ←ここを作る
}

/* pos 番目と pos+1 番目の人の間には - を描く */
void bar_print(int pos) {
    for (int i=0; i<N_PEOPLE; i++) {
        if (i == pos) { printf("|-"); }
        else          { printf("| "); }
    }
    printf("\n");
}

int main(void) {
    int bar[] = { 1, 4, 3 }; // ←ここでコースを変える
    int n_bar = sizeof(bar)/sizeof(bar[0]);

    people_init();
    people_print();
    for (int j=0; j<n_bar; j++) { // j段め
        bar_print(bar[j]);        // bar[j]番目に横線
        people_swap(0);      // ←ここを直す
        // people_print();   // ←デバッグに役立つ
    }
    people_print();
}
```

他にも自由に改造してみてください。例を挙げておきます。

- bar[] のデータの範囲外を検出する
- データを 1 オリジンにする
- 人を番号ではなく、アルファベット（A, B, C, ...）で表示する

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、ある 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。手順の詳細は 135 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 8.8 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 を代入します。11 行目では、小さな数から素数を探すのですが、0 と 1 は除外して、2 以上をループ処理で調べます。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。

is_prime[] は論理型の変数ですから、12 行目の if の**条件式では TRUE と比較しない**のが作法であることを、もう一度確認しておいてください（☞4.5.2 項）。

コラム： 添字の型

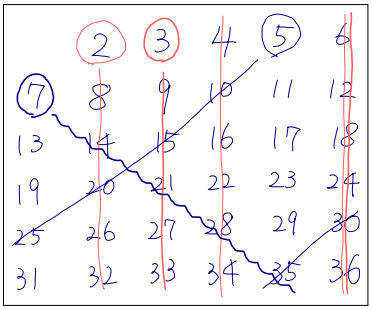
配列の添字は、**整数型**に限られます。浮動小数点型（double や float など）では、コンパイルエラーになります。小数部分が 0 の数値であっても、型で判定されます。

```
int a[10];
double x = 1.0;
// a[x] == 5; // コンパイルエラー
```

このこともあって、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。さらに浮動小数点型には、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という問題もあります。（☞A.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 8.8　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 ", 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     printf("\n");
20 }
```

ソースコード 8.8 の実行結果

```
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89
   97
```

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すみます**[*6]。

この作業をプログラムで実現しましょう。ソースコード 8.9 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）。

ソースコード 8.9 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
```

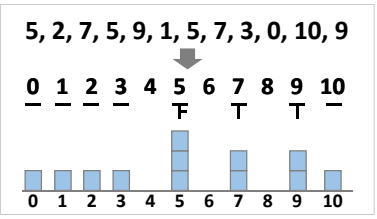

data[i] には、先頭から 5, 2, 7, ... が格納されているので、「正」の 1 画を書き加えることに対応する操作は hist[5]++; hist[2]++; hist[7]++; ... です。これをループで実現したのが 16–17 行目です。この 2 行はまとめて「hist[data[i]]++;」と短く書くこともできます。

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。

目的	配列名	ループ変数	ループ変数の範囲
データ	data	i	$0 \leq i < num$
出現回数	hist	j	$0 \leq j <$ POINT_MAX+1

10 点満点なら 11 通りの点数があるので、POINT_MAX はプログラム上に単独で現れることはなく、常に +1 が付きます。そして 12 行目と 19 行目の j のループは、良いループ（☞5.4 節）に合致します。

[*6] 逆に、「正」の字の場所（メモリ）を節約したければ、データを何度も読み直すことにして、0 点の人を見つけ終わったら、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。（☞137 ページの頻出ミス）



ソースコード 8.9　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+1; 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+1; j++) { // 結果表示
20         printf("%d点が%d人います\n", j, hist[j]);
21     }
22 }
```

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+1; 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);
    }
```

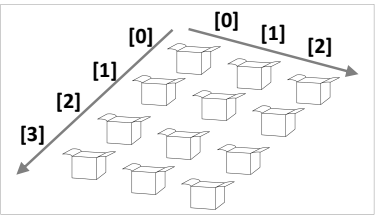
8.5.3　覆面算（発展的内容）

たとえば、$\begin{array}{r} \mathrm{SEND} \\ +\ \mathrm{MORE} \\ \hline \mathrm{MONEY} \end{array}$ のようなアルファベットに数字を割り当てて $\begin{array}{r} 9567 \\ +\ 1085 \\ \hline 10652 \end{array}$ 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 [10]。**覆面算** (alphametic) 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 8.10 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 used[] で管理しています。数字 x が使用済みなら used[x] が真です。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 8.10　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xが使用済みなら used[x]は真
10     for (int i=0; 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 }
```



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 次元の広がりで敷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（定義と代入） */
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（定義と代入） */
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（初期化子） */
int a[6] = {
  1, 2,
  3, 4,
  5, 6,
};
```
（a[6] ← 省略可）

```
/* 2次元配列（初期化子） */
int a[3][2] = {
  { 1, 2 },
  { 3, 4 },
  { 5, 6 },
};
```
（a[3] ← 省略可、a[3][2] ← 必須）

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仮引数では、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（引数渡し） */
int func(int a[6]);
int a[6];
func(a); // 実引数は変数名のみ
```
（a[6] ← 省略可）

```
/* 2次元配列（引数渡し） */
int func2(int a[3][2]);
int a[3][2];
func2(a); // 実引数は変数名のみ
```
（a[3] ← 省略可、a[3][2] ← 必須）

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もできます（☞9.4.1 項）。

8

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** (multi-dimensional array) 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[*7]、`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[*8]。関数に渡すにしても、次元数を合わせる必要があるので、ほぼ専用の関数になって、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（☞9.4.1 項）。

[*7] 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。

[*8] 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、最終的にはポインタ配列にして、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 8.11 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 8.11　2 次元配列の縦横合計

```
#include <stdio.h>

#define PERSON 3
#define SUBJECT 5
int data[PERSON][SUBJECT] = {
        { 80,  70,  40,  60,  80 }, // sum_p[0]
        { 50,  90,  60,  40,  30 }, // sum_p[1]
        { 70,  40,  70,  60,  50 }, // sum_p[2]
};// sum_s[0]  [1]  [2]  [3]  [4]

void calc_sum(int data[PERSON][SUBJECT], int sum_p[], int sum_s[]) {
    for (int p=0; p<PERSON;  p++) { sum_p[p] = 0; } // 初期化(人)
    for (int s=0; s<SUBJECT; s++) { sum_s[s] = 0; } // 初期化(科目)
    for (int p=0; p<PERSON;  p++) {
        for (int s=0; s<SUBJECT; s++) {
            sum_p[p] += data[p][s]; // 加算(人)
            sum_s[s] += data[p][s]; // 加算(科目)
        }
    }
}

int main(void) {
    int sum_p[PERSON], sum_s[SUBJECT];
    calc_sum(data, sum_p, sum_s);        // 合計を計算する
    for (int p=0; p<PERSON; p++) {
        for (int s=0; s<SUBJECT; s++) {
            printf(" %2d ", data[p][s]); // データ表示
        }
        printf("| %d\n", sum_p[p]);      // 合計表示(人)
    }
    for (int s=0; s<SUBJECT; s++) {
        printf("%d ", sum_s[s]);         // 合計表示(科目)
    }
    printf("\n");
}
```

8.7 練習問題

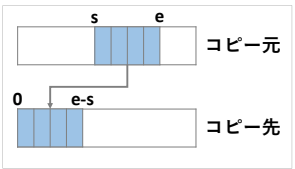
1. **[添字の範囲（☞123 ページの頻出ミス）]**
   右のプログラムで、配列の添字の不正なものをすべて指摘し、不正なものの確実な検出方法が存在するかを述べよ。

```
int main(void) {
    int a[10];
    a[-1] = -1;   a[ 9] =  9;
    a[ 0] =  0;   a[10] = 10;
    a[ 1] =  1;   a[11] = 11;
}
```

2. **[最大値（☞8.4.1 項）]**
   (i) 129 ページのソースコード 8.2 の 7 行目の暫定最大値との比較を <= に置き換えると、どのように動作が変化するのか説明せよ。**(ii) 最小値**を求める関数を作れ。
3. **[最大値の添字（☞8.4.1 項）]**
   (i) 129 ページのソースコード 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には、**どれか 1 つ**の添字を返せばよいが、どれを選んだかを**コメント中で説明**せよ。**(ii)** 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のか、2.(i) との**違いがわかるように**説明せよ。
4. **[配列の一部をコピーする（☞8.4.3 項）]**
   131 ページのソースコード 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。

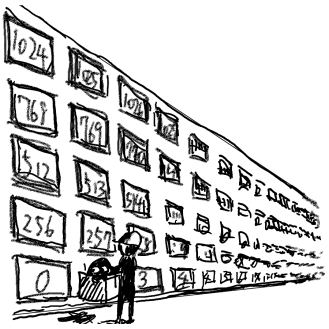
   簡単のため 0 ≤ start < end を前提としてよい。src[end] はコピーしないことに注意し、調整項のない**良いループ**（☞5.4 節）を実現せよ。



5. **[配列の要素をスワップ（☞8.4.4 項）]**
   133 ページのソースコード 8.7 を完成せよ。
6. **[度数分布（☞8.5.2 項）]**
   137 ページのソースコード 8.9 を参考に、100 点満点の成績データを作り、10 点刻みの度数分布を調べ、「0–9 点?人, 10–19 点?人, ...」のように表示せよ。（ヒント：100 点はどの区分に入るのかをよく検討せよ。）
7. **[2 次元配列（☞8.6.2 項）]**
   141 ページのソースコード 8.11 を改造して、3 人 5 科目の点数の総計（合計の合計）も求めて、出力される**表の右下**部分に追加して表示せよ。calc_sum() 関数の型を void から int に変更し、ループ処理で総計を求め、戻り値とせよ。main() からは calc_sum() を **1 度だけ呼び出せ**ば十分である。

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、基本的に配列なので、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。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、少し触れます。

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（文字コードの 0 番）を配置します。一種の**番兵**です。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配列の初期化 */
int a[4];  a[0]=10;  a[1]=20; ...
int b[4] = { 10, 20, 30, 40 };
int c[]  = { 10, 20, 30, 40 };
```

```
/* char配列の初期化 */     // △
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**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、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配列の初期化 */
// 対応する機能なし
```

```
/* char配列の初期化 */
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っても、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]; // int配列への代入
// f = { 10, 20, ... };    // NG
f[0] = 10;
f[1] = 20;
f[2] = 30;
f[3] = 40;
```

```
char f[4]; // char配列への代入
// 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。（実体としては 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は用いませんから、strlen() と int 型変数との間で代入や比較の際には、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、という意思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] sprintf() と snprintf()^{C99} 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。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、自作しなおしてみたのがソースコード 9.1 です。

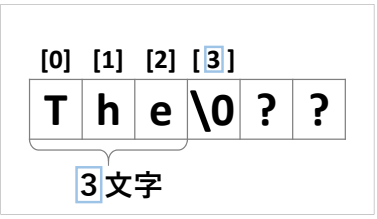
str_length(str) 配列の先頭から'\0' を探しています。str[i] が'\0' になったら、そのときの i が文字列の長さになります。配列の添字が 0 始まりのおかげで、-1 のような調整項が不要で、うまく計算できます。

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。文字列のデータ構造としては、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。実体が unsigned int や unsigined long などの可能性がありますから、書式文字列を`"%d"`や`"%ld"`と決めてしまうと、可搬性が失われます。`size_t` には、C99 で新設された`"%zu"`を使います。あるいは、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（72 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す。

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。

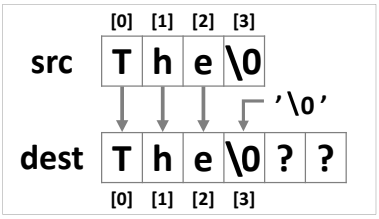
str_copy(dest, src)　ソースコード 9.1 の str_length() と同じ for ループで、1 文字ずつ src[] から dest[] にコピーします。src は source（源泉、情報源）、dest は destination（目的地）の省略形で、それぞれコピー元とコピー先を表す変数名としてよく使われます。

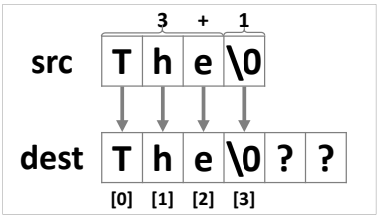
ループ終了後に、dest[] の末尾に終端のヌル文字を書き込みます。ループ変数 i は、最後の dest[i] への代入後にも 1 増えているので（7 行目）、ループ後のヌル文字の代入（10 行目）で i に +1 のような調整項が不要と、うまくできています。

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**（☞5.4 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**のが良いと言われています。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 は printf("%s", str); あるいは puts(str); で表示します。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にまで侵入すると、OS のメモリ保護機能の働きによっては（本当に表示する前に）プログラムが強制停止されます。

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。

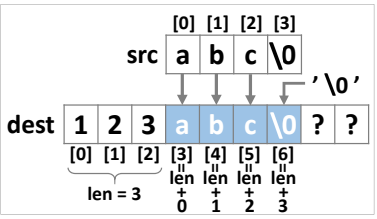
str_concat2(dest, src)　同じ動作ですが、ポインタの知識があれば、コピー先の位置をずらして strcpy() を呼び出すだけですみます。9.3 節を学んでから見直してください。

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[] を用意して、空文字列で初期化しました。そして、strcat() を繰り返して、この後ろに別の文字列を連結していきます。

ただし、格納先の buff[] があふれないことを保証せねばならないので、実用的には、strcat() の前に strlen(dest)+strlen(src)+1 <= sizeof(dest) のような検査が必要です。

ソースコード 9.3　文字列を連結

```
1  #include <stdio.h>
2  #include <string.h> // strlen(), 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]; // len だけずらしてコピー
9      }
10     dest[len + i] = '\0';
11 }
12
13 /* strcat()の自作関数：文字列を連結する */
14 void str_concat2(char dest[], char src[]) {
15     int len = strlen(dest);
16     strcpy(dest + len, src);// ポインタ演算でコピー先アドレスをずらす
17 }
18
19 int main(void) {                // 100 は適当な大きめの値
20     char buff[100] = "";         // buff : ""
21     strcat(buff, "123");         // buff : "123"
22     str_concat(buff, "abc");     // buff : "123abc"
23     str_concat2(buff, "XYZ"); // buff : "123abcXYZ"
24     puts(buff);
25 }
```



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。方法はいくつも考えられます。

```
/* 初期化（変数定義と同時） */

char s[100] = "";      // 直感的
char t[100] = {'\0'}; // 等価
```

```
/* 変数定義と分離した代入 */
char s[100], t[100];
strcpy(s, ""); // 文字列コピー
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。これを初期化子のブロックの代わりにするのが**直感的**です。変数定義と分離するなら、strcpy() で**空文字列をコピー**するのがとりあえずの作法です。関数呼び出しが大げさだと思えば、先頭要素にヌル文字を代入するのも**簡便**でよいでしょう。

ソースコード 9.3 では buff[] を空文字列で初期化していますが、最初の文字列の "123"だけ strcpy() でコピーすれば、初期化の必要がなくなります。

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。

長さと、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、逆に理論上は長さに上限ができます。

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

sprintf() は、文字列用の関数というわけではなく、標準入出力のための <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、sprintf() 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 sprintf() の変形で、書き込まれる文字列のバイト数を制限します。あふれた部分は書き込みません。最後に必ずヌル文字が付きます。

これらの書式変換の機能は、文字列操作にも役立ちます。まずは文字列連結です。

```
char s1[] = "123";
char s2[] = "abc";
char buff[100];

/* 文字列連結 */
sprintf(buff, "%s%s", s1, s2);                 // buff : "123abc"
snprintf(buff, sizeof(buff), "%s%s", s1, s2); // buff : "123abc"
```

次は、atoi() の反対の動作、つまり数値を文字列に展開に利用してみます。

```
/* 文字列→数値 */
int    i = atoi(buff);
double f = atof(buff);
```

```
/* 数値→文字列 */
snprintf(buff, SIZE, "%d", i);
snprintf(buff, SIZE, "%f", f);
```


snprintf() は C99 で導入された[*3]新しい関数ですが、単純な記述で**バッファオーバーフロー** (buffer overflow)（不正な配列操作などによるメモリ破壊）を防げますので、活用したいものです。strcpy() などのほうが実行効率はよいでしょうが、少し油断すると、すぐにヌル文字がつかなかったり、配列サイズをあふれてしまいます[*4]。

コラム：金額の 3 桁ごとのコンマ

大きな金額を表記する際に、3 桁ごとにコンマを入れて「123,456,789 円」などとすることがあります。ところが C 言語には、この表記を実現する直接の機能はありません。このような表記は、国や文化によって異なるため、サポートされなかったものと思われます。

ちなみに、小数点を表すのに、我々はピリオド（.）を使いますが、これにコンマ（,）を使う国もあります。

[*3] C99 で導入される前から、POSIX という Unix 系の規格に存在していて、一部では使われていました。

[*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が、POSIX 時代と C99 とで snprintf() の戻り値が微妙に異なるため、環境依存してしまいがちなのが惜しいところです。

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を指定できて、その文字列の終端には**必ずヌル文字**を書き込みます。

<string.h> に**古くから**ある関数の中にも、strncpy() や strncat() のように、領域のサイズを指定できるものがあります。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、**サイズ上限も終端とする**、少し変わった文字列のデータ構造[a]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、固定長領域のデータを扱うのには役に立つのでしょうが、画面表示には puts() が使えず、printf() で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[3] = "123";`」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で、要注意です。

なお、C11 で「境界チェックインタフェース」として追加された strcpy_s() など xxx_s() の関数群は、バッファオーバーフローを防ぐ工夫がされているものの、機械的な置き換えには難があったり、立場による思惑の違いからか、可搬性に問題のある状態が続いています。

[a] 文字列を部分的に書き換えることも意図されているようです。

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) { ... }             // × この表記ではアドレスの比較になる
if (strcmp(s1, s2) < 0) { ... } // ○ 0と比較する不等号は、上と同じ
```

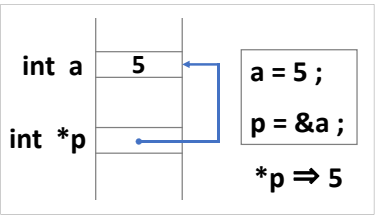
この取り決めに従うと、呼び出し側では、上の例のように「s1 と s2」の間で使いたい比較演算子を、「関数値と 0」の間で使えばよいことになります。上記は < の例ですが、== でも != でも同じ方法で大丈夫です。使用例は 160 ページのソースコード 9.8 を参照してください。

ソースコード 9.4 に strcmp() の実装例[*5]を載せておきます。4 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。この比較が、異なる文字だけでなく、短い文字列の末尾（つまりヌル文字）を小さいと判断するところまで、うまく処理してくれます。

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     for (int i=0; s1[i]!='\0' || s2[i]!='\0'; i++) {
4         int diff = (unsigned char)s1[i] - (unsigned char)s2[i];
5         if (diff != 0) { return diff; } // 異なる
6     }
7     return 0; // 等しい
8 }
```

[*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

通常は、変数に割り当てられたメモリ領域を「変数名」で区別します。領域はコンパイラによって、必要なときに自動的に割り当てられ、不要になれば開放されます。割り当てられた**メモリ領域が途中で変わることはありません**。これは配列でも同じです。領域がいつ、どのように割り当てられるかは、プログラムで気にする必要は（ほぼ）ありません。

ポインタ (pointer) 変数を使うと、メモリ領域の割り当てをプログラムで管理できます。メモリ領域を、CPU が管理する上で用いる「アドレス（番地）」で区別します。ポインタ変数の値（=アドレス）を変更すると、ポインタの指し示す**メモリ領域が変わります**。このおかげで、柔軟な処理ができる反面、指し示す領域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。アドレスの指し示すメモリ領域にある変数の型です。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。変数の定義では、int へのポインタであれば「`int *p;`」と、* を付けます。複数のポインタ変数をまとめるときは、「`int *q, *r;`」と、それぞれに * が必要です。（このおかげで、通常の変数の定義と混在できます。）

```
int a;     // intの普通の変数
int b, c;  // まとめて定義
```

```
int *p;     // intのポインタ変数
int *q, *r; // まとめて定義
int d, e, *s, *t; // 混在
```

このように、ポインタは、領域を「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。

表 9.1　ポインタの演算子

演算子	演算の種類
&	アドレス
*	間接参照

表 9.2　通常の変数とポインタ変数の関係

	変数定義	アドレスの参照	int の参照
通常	`int x;`	`&x`	`x`
ポインタ	`int *p;`	`p`	`*p`

9.3.1　ポインタを扱う演算子

表 9.1 にまとめたように、変数のアドレスを取り出すのが**アドレス** (address) 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** (indirection) 演算子 ***** です。この * は、先ほどの変数定義の * とは役割が違って**作用はほぼ正反対**ですが、次のように、混同すると（かえって）**つじつまが合う**ように巧妙に作られています。

「`int *p, x;`」と定義すると、`p = &x;`（アドレスの代入）と `*p = x;`（int の代入）が両辺の型の一致する操作です（表 9.2）。

「`int *p;`」は、もともと「**p は int ポインタ 型**」の意味ですが、（* を間接参照かのように）「*p」を変数名と考えると、「***p は int 型**」とも解釈できます。

ただし、**初期化**の場面での * には注意してください。変数への代入を短縮表記しているだけですから、* は変数定義の一部として解釈する必要があります。

```
/* ポインタ定義の2通りの解釈 */
int *p;  // 「p は int*」
         // 「*p は int」
int x = 5;
p = &x;  // アドレスの代入
*p = x;  // intの代入
```

```
/* 初期化 */
int x = 5;
int *p = &x; // アドレスで初期化
```

9.3.2　配列とポインタの関係

`int a[10];` の a は、ポインタとしての機能もあるので、`a[0]` と `*a` が同じ意味です。`a[1]` は `*(a+1)` が同じです。* の優先順位が高いので、アドレスの加算を先にするよう、このカッコが必要です。`int *p = a;` とポインタ変数を用意すると、`p[0]` や `*p` が `a[0]` と同じです。`int *q = a+1;` と先にアドレスを加算しておくと、`q[0]` と `a[1]` が同じです。a は配列のため、アドレスは変更できませんが、ポインタ変数は `p++;` や `q += 4;` のようにして、アドレスを変更できます。下の表の水色部分は、よく使う表現です。

変数定義（初期化）	`int a[10];`		`int *p = a;`		`int *q = a+1;`	
a の 0 番目要素	`a[0]`	`*a`	`p[0]`	`*p`	`q[-1]`	`*(q-1)`
a の 5 番目要素	`a[5]`	`*(a+5)`	`p[5]`	`*(p+5)`	`q[4]`	`*(q+4)`
アドレスの加減算	不可能		`p++;`		`q += 4;`	
a の 5 番目要素	`a[5]`	`*(a+5)`	`p[4]`	`*(p+4)`	`q[0]`	`*q`

9.3.3　関数に渡すポインタ

関数の引数をポインタにすると**参照渡し**[*6]になります（☞3.8.2 項）。48 ページのソースコード 3.8 で失敗したスワップ関数が、ソースコード 9.5 のように実現できます。呼び出す側では、実引数の変数に & を付けます。関数内では、*x と *y を int 変数と思って入れ替えます。これで main() 側の a と b が（main() では代入していないのに）入れ替わります。（値の変わるはずのない）定数に & を付けると、コンパイルエラーになります。

ソースコード 9.5　スワップ関数

```
#include <stdio.h>

void swap(int *x, int *y) {          //  x と  y はintへのポインタ
    int tmp = *x; *x = *y; *y = tmp; // *x と *y はint
}

int main(void) {
    int a = 1, b = 2;
    swap(&a, &b); // & でポインタにする（アドレスを取り出す）
    printf("main: a=%d, b=%d\n", a, b); // a=2, b=1
//  swap(&1, &2); // 定数には & は付けられない（コンパイルエラー）
}
```

scanf() が変数の値を書き換えることができるのも、そして引数の変数を & 付きで渡す必要があるのも、同じ仕組みです。

配列をコピーする関数は、131 ページのソースコード 8.5 で array_copy() を作りましたが、ソースコード 9.6 のように、ポインタとしてもコピーできます。仮引数では、int dst[] は int *dst と解釈されるので、array_copy() と pointer_copy() の役割はまったく同じです。

ソースコード 9.6　配列のコピー（ポインタ版）

```
void pointer_copy(int n, int *dst,  int *src) {
    //           (int n, int dst[], int src[]) でも同じ
    for (int i=0; i<n; i++) {
        *dst++ = *src++;                // ポインタでコピー（短縮）
     // *dst   = *src;  dst++;  src++;  // ポインタでコピー（分解）
     // dst[i] = src[i];                // 配列としてコピーもOK
    }
}
```

*dst++ = *src++; の表記は、*dst = *src; とアドレスのインクリメントをまとめたもので、CPU にとっても効率のよい動作に近いのですが、昨今の最適化技術の進んだコンパイラであれば、dst[i] = src[i]; も含め、どの表記でも効率に大差はないでしょう。

[*6] ポインタ変数としては、（相変わらず）値渡しですが、ポインタの指すもので考えると参照渡しです。

9.4 文字列とポインタ

文字列を扱うのに、char 配列にするか、char のポインタにするかで、2 通りの手法があります。ソースコード上の "The" は、char a[4] = "The"; なら**配列の初期化子**ですし、char *p = "The"; なら**文字列リテラルへのポインタ**と、役割が違います。

```
/* char配列 */
char a[4] = "The"; // 初期化子
char b[4] = {'T','h','e','\0'};
a[0] = 's';   // ○1文字の書き換え
//a = "the";  // NG アドレスの
//a++;        // NG 変更不可
printf("%zu", sizeof(a)); // 4
```

```
/* charポインタ */
char *p = "The"; // 文字列リテラル
//char *q={'T','h','e','\0'}; //NG
p[0] = 's';  // ×動作保証なし
p = "the";   // ○ポインタの代入
p++;         // "he"
printf("%zu", sizeof(p)); // 8 (64
    bitアドレスの場合)
```

char a[4]; では、a のメモリ割付が固定されるので、a = "the"; とか a++; のような**アドレスの変更ができません**。一方、char *p; では、文字列リテラルを指すポインタなので、p = "the"; とか p++; のような**アドレスの変更が可能です**。ただし、**文字列リテラルは ROM 領域**に割り当てられる可能性があるため、p[0] = 's'; のような 1 文字単位の書き換えの動作保証がありません。（しかも、実行時エラーになるとも限りません。）

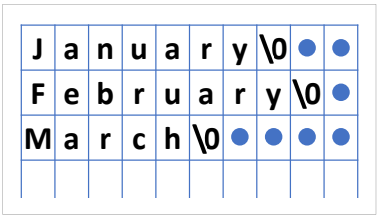
sizeof による大きさは、char 配列では (文字数)+1 です。+1 はヌル文字分です。char ポインタなら、文字数には影響されず、ポインタ変数に共通する大きさ（32bit アドレスなら 4、64bit アドレスなら 8）の固定値です。

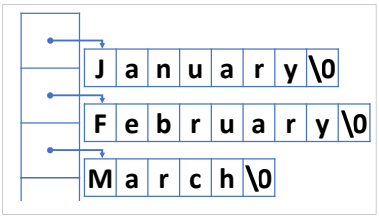
コラム：文字列リテラルの共有

「char *p = "The", *q = "The";」とした場合、コンパイラによっては、p と q のアドレスが同じになる、つまり同一の文字列リテラルを共有することがあります。C 言語規格では、**文字列リテラルは ROM 領域**に割り当てられることを想定しているので、共有を許す代わりに、逆に書き換えが保証されません。

9.4.1 文字列の配列

いくつも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ります。どちらを採用するかは (1) メモリに無駄がないか (2) プログラムの実行中に変更するか (3) 関数に渡しやすいか、などで判断します。現実には (3) を重要視して、これから紹介する**後者**を採用することが多いでしょう。ソースコード 9.7 を見ていきます。





char の 2 次元配列　month_name は char の 2 次元配列です。12 ヶ月分、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ました。短い名前の後ろには使わない無駄なメモリができますが、この長さを上限として、実行中に変更できます。

char のポインタの配列　month_name2 は char* 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の文字列リテラルが用意されます。メモリに無駄はありませんが、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、節約にならないこともあります。そして、ポインタの先の文字列リテラルは定数なので、(1 文字単位の) 代入の動作が保証されません。(ポインタの変更、つまり文字列の差し替えは可能です。)

16 行目のような場面では、使い方に区別はありません。C 言語の巧妙なところです。

ソースコード 9.7　月名の配列

```
1  #include <stdio.h>
2  #define N_MONTH 12
3
4  char month_name[N_MONTH][10] = { // charの2次元配列
5      "January", "February", "March", "April", "May", "June",
6      "July", "August", "September", "October", "November", "December"
7  };
8
9  char *month_name2[N_MONTH] = {   // charポインタの配列
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 }
```

ただし、関数に渡すことを考えると、1 次元配列ですむ、ポインタの配列が有利です。ソースコード 9.8 では、辞書順で比較した最小の文字列を返す `str_min(n, s)` を作りました。文字列の配列は `char *s[]` と、ポインタの配列で受け取ります。もしも 2 次元配列で受け取るなら、文字列のサイズごと（例えば `char s[][10]` と `char s[][11]`）に、別の関数を作らねばなりません。

ソースコード 9.8　辞書順で最小の文字列

```c
#include <stdio.h>
#include <string.h> // strcmp()
#define N_MONTH 12

char *month_name2[N_MONTH] = {   // charポインタの配列
    "January", "February", "March", "April", "May", "June",
    "July", "August", "September", "October", "November", "December"
};

char *str_min(int n, char *s[]) {
    char *min = s[0];
    for (int i=1; i<n; i++) { // 先頭要素を除外したループ
        if (strcmp(min, s[i]) > 0) { min = s[i]; }
    }
    return min;
}

int main(void) {
    printf("%s\n", str_min(N_MONTH, month_name2)); // April
}
```


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の原因の多くが、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であ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 のような**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である**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ないからか、スコープが該当ブロックに限られていているからか、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。反面、Java での類似の機能「無名配列」はよく使われています。

9.5 練習問題

1. **[可搬性（☞9.2 節、9.2.1 項）]**
   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、手元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（☞9.2.1 項）]**
   146 ページのソースコード 9.1 を参考に、4.5.2 項の isdigit() を用いて、次の関数を作れ。
   - `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（☞9.2.2 項）]**
   148 ページのソースコード 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。
     ヒント：2.5.2 項の `toupper()` は、小文字を大文字に変換し、それ以外の文字は変換せずにそのまま返す。
   - `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。

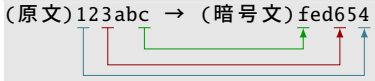
4. **[文字列比較（☞9.2.5 項、9.4.1 項）]**
   気象庁の発表する、6 種類の特別警報と 7 種類の警報について、所属機関が休講になるかどうかを表示せよ。
   警報の種類や、所属機関の規定は、Web 等を活用して調べよ[*8]。

   【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（☞8.4.3 項）]**
   スパイから君に、右の暗号文が届いた。スパイは、文字列の前後を入れ替え、文字の ASCII コードを +3 ずらして暗号文を作っているらしい。逆の操作で、原文に戻してくれ。

```
(暗号文) xndnxrj0xndqrpdp
```

```
(原文)123abc → (暗号文)fed654
```



6. **[文字列操作]**
   文字列で表された数値を int に変換する処理を自作せよ。
   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。下の char2int() も参照せよ。

   "1"　⇒　0 * 10 + 1 = 1
   "12"　⇒　(0 * 10 + 1) * 10 + 2 = 12
   "123"　⇒　((0 * 10 + 1) * 10 + 2) * 10 + 3 = 123

```
int char2int(char c) { return c - '0'; } //   1文字の数値変換
```

7. **[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

ユーザ定義型と構造体

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。

ここにきて、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

- ユーザ定義型・typedef・struct
- 構造体のメンバ変数
- 値渡し・参照渡し
- 型と関数名を直交させる
  →オブジェクト指向

10.1　基本データ型・ユーザ定義型

「**型**」の種類は、次の 2 つに大別されます。

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)
　プログラムで作った型です。基本データ型を組み合わせて作ります。

ユーザ定義型を作る方法はいくつかあります[*1]。まず typedef 命令の使い方を見てみましょう。このキーワードは type definition（型定義）の意味でしょう。次の文法[*2]で、既に存在する型に、別の名前を与えます。

```
/* 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]。なお、型名には、このように **“_t” の接尾辞**をつける習慣があります。

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。

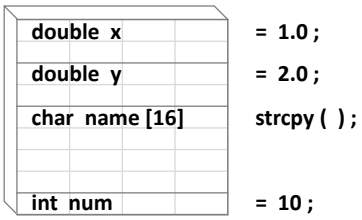
ある数値を扱うのに 32 ビットの領域が必要だとしましょう。int を使うと、環境によっては正しく動きません。そこで、例えば int32_t という型を使うことにして、環境に応じて以下のどちらか片方を有効にして、int か long かを選びます。

```
  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] enum, union もユーザ定義型を作りますが、本書では扱いません。
[*2] 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。そして定義の語順が逆です。
[*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　構造体

構造体 (structure) は、複数の変数をまとめるユーザ定義型です。例えば、*xy* 平面上の座標を扱う時には、*x* 座標と *y* 座標の 2 つの変数を、常にペアにして使いますから、このような場面で役立ちます。構造体を作るには、次のように `struct` 命令を用います。そして先ほどの typedef と組み合わせるのが簡便です。

```
/* 文法（短縮した慣用句） */
typedef struct {
    型名1 メンバ名1;
    型名2 メンバ名2;
    ...
} 新規の型名;
```

```
/* 実例 */
typedef struct {
    double x; // double x, y;
    double y; // でもよい
} point_t;
```

struct の後の { } の中に、ひとまとめにする変数を列挙します。これらの変数を、構造体の**メンバ** (member) といいます。右上の例では double 型の x と y です。この構造体に、typedef で型名を付けます。例では point_t にしています。

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**型**ですから、「double」のような型名を書くところに「point_t」と書きます。

```
/* 普通の変数の定義と代入 */
double x, y;
x = 1.0;  y = 2.0;
```

```
/* 構造体変数の定義と代入 */
point_t p;
p.x = 1.0;  p.y = 2.0;
```

構造体変数を1つ作ると、含まれるメンバすべてが1組作られます。右の例では、変数 p のメンバの x と y が作られるので、値を代入してみました。構造体の中のメンバを指すには、このように**構造体メンバ参照**の演算子 「.」に続けてメンバ名を指定します。メンバ変数も、普通の変数と同じように、代入したり、参照したりできます。

構造体の**初期化**は、1 次元配列と同じで、初期化子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;
```

```
/* 構造体変数の初期化 */
point_t p = { 1.0, 2.0 };
```

コラム： typedef と組み合わせない struct

構造体には、**構造体タグ名** (tag) をつける用法もあります。この場合、型名は「struct 構造体タグ名」の 2 つの単語になり、typedef は使わなくてよくなります。

```
/* 文法 */
struct 構造体タグ名 {
    型名1 メンバ名1;
    型名2 メンバ名2;
    ...
};

/* 変数定義と代入 */
struct 構造体タグ名 変数名;
変数名.メンバ名 = 値;
```

```
/* 実例 */
struct point {
    double x;
    double y;
}; // ←セミコロン忘れに注意


/* 変数定義と代入 */
struct point p;
p.x = 1.0;  p.y = 2.0;
```

この用法には、注意点がいくつかあります。

- struct のキーワードを記述する場面が多くなり、少々煩雑です。
- 構造体宣言の末尾のセミコロン（;）を忘れがちです。
- C++ との表記の乖離が大きくなります。C++ では、構造体タグ名が、そのまま型名でもあるので、struct を省略しがちです。先に紹介した、typedef との組合せは、C++ と表記を共通にするための手段でもあります。

ただし、メンバに同じ構造体（へのポインタ）を含む**再帰的なデータ構造**には必須の用法です。そして、次のように typedef と組み合わせることも多くあります。

```
/* typedef と分離 */
struct list {
    int data;
    struct list *next;
};
typedef struct list list_t;
```

```
/* typedef と同時 */
typedef struct list {
    int data;
    struct list *next;
    // list_t はここでは未定義
} list_t;
```

右の例でも、list_t はメンバの定義には使えないことに注意してください。

10.2.1　構造体のスコープ

構造体の（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。つまり、変数と同じです。しかし構造体は、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ほとんどありません。ですから、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。

10.3 構造体の使用例

構造体を作ったら、同時にメンバを表示する関数も作りましょう。デバッグのためにいずれ必要になります。ソースコード 10.1 を見ていきます。

4–6 行目の構造体の宣言は、**関数定義やプロトタイプ宣言よりも前**にすませる必要があります。順序を間違えると、不可解なコンパイルエラーになります。

10 行目では、printf() で構造体変数 p のメンバの x と y を表示しています。メンバ参照演算子の . でメンバの値を一つずつ取り出します。**printf() に構造体を自動的に表示する機能はない**ので、このようにメンバを一つずつ指定して表示する必要があります。

15 行目の関数呼び出しでは、構造体変数 a を引数に渡しています。int 型でも point_t 型でも、**引数への渡し方は同じ**です。

17 行目では、b に a を代入しています。**構造体変数を代入すると、すべてのメンバがコピー**されます。18 行目の表示から、メンバの x, y ともコピーされたことがわかります。

ソースコード 10.1　座標を表示

```
1  #include <stdio.h>
2
3  /* 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
4  typedef struct {
5      double x, y;
6  } point_t;
7
8  /* pを表示する */
9  void point_print(point_t p) {
10     printf("(%g, %g)\n", p.x, p.y); // .(ドット)でメンバーを取り出す
11 }
12
13 int main(void) {
14     point_t a = { 1.1, 2.2 }; // a.x = 1.1;  a.y = 2.2;  [初期化]
15     point_print(a);           // (1.1, 2.2)              [引数渡し]
16     point_t b;
17     b = a;                    // b.x = a.x;  b.y = a.y;  [代入]
18     point_print(b);           // (1.1, 2.2)
19 }
```

10.3.1　構造体を扱う関数を増やす

次に、point_t の点を移動する関数を作ってみましょう。ソースコード 10.2 です。

15 行目の point_move(p, dx, dy) は、点 p を (dx, dy) 方向に移動した点を返します。16 行目で、引数で受け取った p の x, y 座標を変化させて、17 行目で移動後の p を返します。

呼び出し側の main() では、23 行目で point_move() の戻り値を変数 a に代入して、値を更新します。

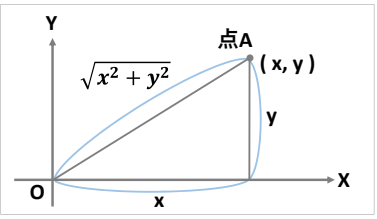
10.3.2　構造体のメンバを増やす

プログラムが一旦完成しても、後になって機能拡張したくなることは、よくあります。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、ソースコード 10.3 のように、構造体のメンバを増やします。

6 行目は、文字列を格納するメンバ変数 label を追加しました。10 行目からの point_print() は、引数はそのままですが、label を表示するための処理を追加しました。21 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。

ソースコード 10.2　点を移動する

```
1  #include <stdio.h>
2
3  /* 点を表す構造体 */
4  typedef struct {
5      double x, y;
6  } point_t;
7
8
9  /* pを表示する */
10 void point_print(point_t p) {
11     printf("(%g, %g)\n", p.x, p.y);
12 }
13
14 /* pを(dx,dy)方向に移動する */
15 point_t point_move(point_t p, double dx, double dy) {
16     p.x += dx;  p.y += dy;
17     return p;
18 }
19
20 int main(void) {
21     point_t a = { 1.1, 2.2 };
22     point_print(a);           // (1.1, 2.2)
23     a = point_move(a, 1.2, 3.4);
24     point_print(a);           // (2.3, 5.6)
25 }
```



メンバが増えても 15 行目からの point_move() は以前と同じです。22–24 行目の関数呼び出しもそのままです。もし構造体ではなく、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**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 10.3　点を移動する（ラベル付き）

```
1  #include <stdio.h>
2
3  /* 点を表す構造体 */
4  typedef struct {
5      double x, y;
6      char *label;                                  // *label 追加
7  } point_t;
8
9  /* pを表示する */
10 void point_print(point_t p) {
11     printf("点%s(%g, %g)\n", p.label, p.x, p.y); // p.label 追加
12 }
13
14 /* pを(dx,dy)方向に移動する */
15 point_t point_move(point_t p, double dx, double dy) {
16     p.x += dx;  p.y += dy;
17     return p;
18 }
19
20 int main(void) {
21     point_t a = { 1.1, 2.2, "A" };               // "A" 追加
22     point_print(a);              // 点A(1.1, 2.2)
23     a = point_move(a, 1.2, 3.4);
24     point_print(a);              // 点A(2.3, 5.6)
25 }
```

10.3.3　構造体の代入

これまで見てきたように、通常の int 型の変数と同じように、point_t a, b; と定義した**構造体の変数**は、a=b のような**代入**が行えて、構造体の**すべてのメンバがコピー**されました。関数の**引数渡し**も**戻り値**も、この代入相当の動作が行われていて、すべてのメンバがコピーされました。

この代入は便利そうに思える機能ですが、構造体が大きくなると効率性が問題になります。構造体のメンバに配列を含めることもできますが、動作が保証されるサイズには上限があります[*4]。このため、次に紹介するように、通常は**関数とは構造体をポインタでやりとり**します。

10.4　構造体によるデータ構造

10.4.1　構造体のポインタ

大きな領域のコピーを避けるために、関数とは**構造体をポインタで**受け渡すのが一般的な手法です。ポインタにするためには、次のようにします。

仮引数 受取側の関数では、引数名に * を付加します。

実引数 呼び出し側では、ローカルに定義した構造体変数なら、& を付加して、アドレスに変換します。初めからポインタ型であれば、変数名だけで大丈夫です。

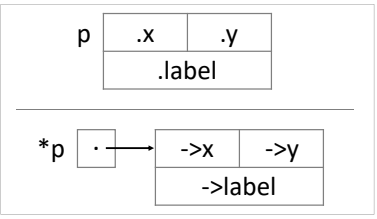
戻り値 構造体を戻り値とする必要がなくなります。関数の型は void にして、代わりに引数にし、関数で書き込みます。

ポインタ変数の point_t *p のメンバを指すには、文法通りには「(*p).x」と煩雑な表記になります。ここにカッコが必要なのは、間接参照の * の優先順位が低く、メンバ参照の . が先に評価されるからです。しかし、これはよく使う機能ですから**専用の演算子**が用意されていて、**ポインタ用のメンバ参照**の「->」を用いて「p->x」と簡潔に表記できます。

```
/* 引数が構造体 */
double point_sum_xy(point_t p) {
    return p.x + p.y;
}
```

```
/* 引数が構造体ポインタ */
double point_sum_xy(point_t *p) {
// return (*p).x + (*p).y; // 可能
   return p->x   + p->y;   // 通常
}
```

[*4] C99 で 64KB、C89 で 32KB を超えると、動作不良を起こす場合があります。そもそも最初期のコンパイラでは、構造体変数の代入相当の動作自体がサポートされていない場合もありました。



169 ページのソースコード 10.3 を、ポインタ渡しに書き換えたのがソースコード 10.4 です。これで実用的なプログラムになりました。変更点は以下の通りです。

(a) 関数の仮引数は、* を追加してポインタ型にしました。
(b) メンバ参照の演算子を、ポインタ型の場合には . の代わりに -> にしました。
(c) 構造体を戻り値にしているものは、代わりに引数に作用するようにしました。呼び出し側では、戻り値を代入せずとも、実引数が変化します。
(d) 関数の実引数は、ローカルに確保した場合は & でアドレスに変換しました。

ソースコード 10.4　原点からの距離を表示（ポインタ渡し）

```
1  #include <stdio.h>
2
3  /* 点を表す構造体 */
4  typedef struct {
5      double x, y;
6      char *label;
7  } point_t;
8
9  /* pを表示する */
10 void point_print(point_t *p) {                        // (a)* 追加
11     printf("点%s(%g, %g)\n", p->label, p->x, p->y); // (b).を->に変更
12 }
13
14 /* pを(dx,dy)方向に移動する */                        // (c)void に変更
15 void point_move(point_t *p, double dx, double dy) { // (a)* 追加
16     p->x += dx;  p->y += dy;                         // (b).を->に変更
17     /* return p; */                                  // (c)return 削除
18 }
19
20 int main(void) {
21     point_t a = { 1.1, 2.2, "A" };
22     point_print(&a); // 点A(1.1, 2.2)                // (d)& 追加
23     /* a = */ point_move(&a, 1.2, 3.4); // (c)a= 削除, (d)& 追加
24     point_print(&a); // 点A(2.3, 5.6)                // (d)& 追加
25 }
```

10.4.2　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、角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];

x[0] = 1.0;  y[0] = 3.0;
label[0] = "A";
```

```
/* 構造体の配列 */
typedef struct {
    double x, y;
    char *label;
} point_t;
point_t p[3];

p[0].x = 1.0;  p[0].y = 3.0;
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] = {"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" },
};
```

配列の 1 要素を、ポインタで関数に渡すときには、int でも構造体でも同じで、添字を指定した上で、& でアドレスに変換します。

```
/* 普通の配列要素のポインタ */
for (int i=0; i<3; i++) {
    point_print(&x[i], &y[i]);
}
```

```
/* 構造体の配列要素のポインタ */
for (int i=0; i<3; i++) {
    point_print(&p[i]);
}
```

配列全体を関数の引数として渡すこともできます。（構造体に限らず）配列はポインタと同じで、関数には参照渡しになり、要素数はポインタからはわかならいので、別の引数で渡します。

```
/* 普通の配列 */
point_print_array(3, x, y);
```

```
/* 構造体の配列 */
point_print_array(3, p);
```

10.4.3　メンバの同じ構造体

構造体は、宣言するたびに新しい型になります。たとえメンバが同じでも、宣言しなおせば、**異なる型**と扱われます。これはありがたい性質です。

double s[2]; と定義した変数なら、例えば、左右の視力でも、左右の握力でも格納できます。視力と握力が**同じ型**ですから、取り違えたおかしなプログラムも作れてしまいます。構造体にしていれば、取り違えるとコンパイルエラーになるので、間違いが未然に防げます。このように、構造体は**型で意味づけを与えて**います。

```c
/* double配列 */

  // 準備は不要



/* 視力表示 */
void sight_print(double s[2]) {
    printf("視力 左:%g, 右:%g\n",
           s[0], s[1]);
}

int main(void) {
    // 視力
    double s[2] = { 1.2, 1.5 };
    // 握力(kgf)
    double g[2] = { 15.5, 13.0 };
同じ型
    sight_print(s); // ○
    sight_print(g); // 無意味
}
```

```c
/* 構造体 */
typedef struct { // 視力
    double left, right;
} sight_t;

typedef struct { // 握力(kgf)
    double left, right;
} grip_t;                    同じメンバ

/* 視力表示 */
void sight_print(sight_t *s) {
    printf("視力 左:%g, 右:%g\n",
           s->left, s->right);
}

int main(void) {
    // 視力
    sight_t s = { 1.2, 1.5 };
    // 握力(kgf)
    grip_t g = { 15.5, 13.0 };
異なる型
    sight_print(&s); // ○
//  sight_print(&g); // エラー
}
```

10.5　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);
void point_move(point_t *p, double dx, double dy);
```

ところで、型が違っても、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);
```

○

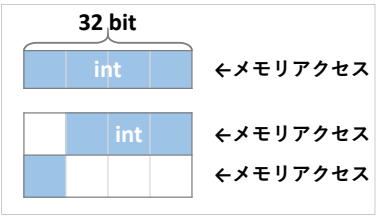
```
void point_print(point_t *p);
void hoge_print(hoge_t *h);
void fuga_print(fuga_t *f);
```

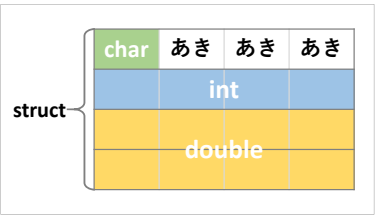
10

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「型とそれにまつわる関数」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。





10.6　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ことがあります。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 回のメモリのアクセス（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、続きのメンバを**次の 4 の倍数の番地**に割り当てる、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、考慮の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 char_and_int {
    char a;
    int b;
};
printf("%zu\n", sizeof(char));   // 1
printf("%zu\n", sizeof(int));    // 例:4（環境による）
printf("%zu\n", sizeof(struct char_and_int)); // 例:8（環境による）
```

上の例では、char が 1 バイト、int が 4 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 – (1 + 4) = 3 バイトのパディングが含まれているとわかります。

10.7　例題・分数計算

分数の和と差を計算してみましょう。120 ページの練習問題 4. では、和を受け取るのに分子用と分母用に 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 10.5 を見てみましょう。

型宣言 5–8 行目で、分数 $\frac{b}{a}$ を表す構造体を作ります。型名を、fraction（分数）の省略で frac_t としてみます。分子、分母ともに整数でよくて、メンバ名を単純に b と a にします。**分母 a を正**にすると決めておくと、この後の場合分けが少なくなります。

約分 26–30 行目の frac_reduce(*x) は、まず受け取った *x の分子と分母の**最大公約数**を求めます（27 行目）。分子は負の可能性があるので、絶対値をとります。そして分子・分母ともに公約数で割ります（28–29 行目）。**引数の *x を加工**しているので、関数自体は値を返す必要がなく、**void 型**にしています。

加算 33–37 行目の frac_add(*r, *x, *y) では、分数の和 $\frac{b}{a}+\frac{d}{c}=\frac{bc+ad}{ac}$ の計算をします。34–35 行目で、引数の *r の分子と分母に、約分前の計算結果を代入します。そして 36 行目で、**約分のために frac_reduce(r) を呼び出し**て完了です。r はポインタ型ですから、関数呼び出しに & は不要です。

減算 差の計算は、**符号を変えた和の計算**と同じです。40–43 行目の frac_sub(*r, *x, *y) では、41 行目で *y の分子の符号を変えた分数 tmp を作り出して、42 行目で**加算の関数を frac_add(r, x, &tmp) と呼び出し**ます。約分も加算の関数に任せていて簡単です。tmp はポインタ型ではないので、関数呼び出しには & が必要です。

10

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、それぞれの関数で求めていました。後の処理を、他の関数に任せること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 10.5　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
10 // プロトタイプ宣言
11 int gcd(int x, int y);
12 void frac_reduce(frac_t *x);
13 void frac_add(frac_t *r, frac_t *x, frac_t *y);
14 void frac_sub(frac_t *r, frac_t *x, frac_t *y);
15 void frac_print(frac_t *x);
```

```
16
17 /* xとyの最大公約数を返す (x,y>=0 かつ x+y>0 を前提とする) */
18 int gcd(int x, int y) {
19     while (y != 0) { // ユークリッドの互除法
20         int t = x % y;  x = y;  y = t;
21     }
22     return x;
23 }
24
25 /* 約分 */
26 void frac_reduce(frac_t *x) {
27     int d = gcd(abs(x->b), x->a);
28     x->b /= d;
29     x->a /= d;
30 }
31
32 /* 加算 (*r = *x + *y)  */
33 void frac_add(frac_t *r, frac_t *x, frac_t *y) {
34     r->b = x->b * y->a + x->a * y->b;
35     r->a = x->a * y->a;
36     frac_reduce(r); // 約分を利用
37 }
38
39 /* 減算 (*r = *x - *y)  */
40 void frac_sub(frac_t *r, frac_t *x, frac_t *y) {
41     frac_t tmp = { -y->b, y->a }; // 分子の符号を反転した分数を作る
42     frac_add(r, x, &tmp);         // 加算を利用
43 }
44
45 /* 表示 */
46 void frac_print(frac_t *x) {
47     printf("%d/%d\n", x->b, x->a);
48 }
49
50 int main(void) {
51     frac_t x = { 7, 12 }; // 7/12
52     frac_t y = { 1, 4 };  // 1/4
53     frac_t r;             // 結果の代入先
54     printf("add "); frac_add(&r, &x, &y); frac_print(&r); // 5/6
55     printf("sub "); frac_sub(&r, &x, &y); frac_print(&r); // 1/3
56     printf("0   "); frac_sub(&r, &x, &x); frac_print(&r); // 0/1
57 }
```

$$\frac{x_b}{x_a} = \frac{x_b/d}{x_a/d}$$

$$\frac{r_b}{r_a} = \frac{x_b}{x_a} + \frac{y_b}{y_a} = \frac{x_b y_a + x_a y_b}{x_a y_a}$$

10.8　練習問題

1. **[構造体の関数渡し（☞10.3.1 項）]**
   167 ページのソースコード 10.1 に、次の関数を追加せよ。

- `double point_norm(point_t p)` は、原点から点 p までの距離を返す。

2. **[構造体ポインタの関数渡し（☞10.4.1 項）]**
   171 ページのソースコード 10.4 に、次の関数を追加せよ。
   - `double point_norm(point_t *p)` は、原点から点 *p までの距離を返す。
3. **[メンバ変数の追加（☞10.3.2 項、10.4.1 項）]**
   171 ページのソースコード 10.4 を改造して、point_t 型に z 座標を追加して、3 次元座標上の点に拡張せよ。point_move() の引数には z 座標の移動量 dz を追加し、main() で必要になる z 座標の値は自由に設定せよ。
4. **[名は体を表す（☞7.3 節）]**
   ソースコード 10.5 の演算後の**約分**が不要になったとする。次の 3 通りの改修案は、いずれも frac_add() に同じ効果を与えるが、**他人がさらに改修する可能性を考慮**して、避けるべきものを 2 つ選び、その理由を述べよ。 **(i)** 22 行目を「`return 1;`」にする **(ii)** 28–29 行目をコメントにする **(iii)** 36 行目をコメントにする
5. **[関数の再利用]**
   ソースコード 10.5 では、frac_sub() は frac_add() の**関数を呼び出し**ているが、呼び出さずに関数の**処理をコピーして符号を反転**しても実現できる。この 2 つの手法のどちらが有利かを、次の 3 通りの評価基準で判断せよ。**(i. 実装効率)** プログラムが作りやすい **(ii. 信頼性)** 動作検証が容易 **(iii. 保守性)** プログラムの改造が容易
6. **[構造体の関数渡し]**
   ソースコード 10.5 に、*x の逆数を *r に代入する関数 frac_inv(*r, *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 *r 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。
   そして加算と同様の、乗算 frac_mul(*r, *x, *y) と、除算 frac_div(*r, *x, *y) の関数を追加せよ。いずれも、分母が正の、約分した分数を *r に代入せよ。
7. **[構造体の関数群]**
   2 次方程式 $ax^2 + bx + c = 0$ に関して、次の 2 つの構造体（型）を宣言せよ。
   - `quad_t` のメンバは double 型の a, b, c であり、方程式の**係数**を表す。
   - `sol_t` のメンバは double 型の alpha, beta であり、方程式の**実数解**を表す。

   そして、次の関数群を作れ。簡単のため、$a \neq 0$ かつ**判別式は非負**としてよい。
   - `void quad_unsolve(quad_t *q, sol_t *s)` は *s の alpha, beta を解とする方程式を *q に代入する。$(x-\alpha)(x-\beta) = x^2 - (\alpha+\beta)x + \alpha\beta$ を利用してよい。
   - `double quad_det(quad_t *q)` は *q の判別式 D $(= b^2 - 4ac)$ を返す。
   - `void sol_solve(sol_t *s, quad_t *q)` は *q の実数解 $(\frac{-b \pm \sqrt{D}}{2a})$ を *s に代入する。
8. **[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 (text format) 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）

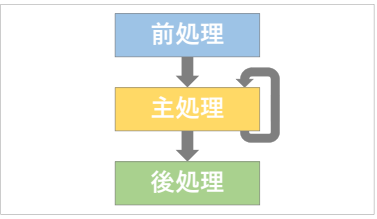
バイナリ形式 (binary format) そのままでは画面表示できず、何らかの変換をせねば意味のわからないものです。コンピュータ向きです。（例：実行ファイル、画像ファイル、圧縮ファイル、アプリケーションごとの独自形式など）

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）、Shift_JIS、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。

```
#include <stdio.h>

FILE *fopen(const char *path, const char *mode);
```

path　どのファイルを扱い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 (open mode) 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（表 11.2）を返すので、検出してエラー処理をせねばなりません。

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（☞187 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（☞B.3 節）。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()

```
#include <stdio.h>

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。

```
#include <stdio.h>

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。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。

コラム：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）。値は 0 であると、言語規格で定められています。（ちなみに、ヌル文字とは何の関係もありません。）

fclose(fp) でクローズした後の fp に NULL を代入する習慣があります。2 重にクローズすることを防いだり、クローズ忘れを検査するのに都合がよいためです。

[*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こりました。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() が便利ですので、ここで紹介します。

```
#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した。

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シェル（コマンドインタプリタ）が受け取ります。シェル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。

なお、C99 からは main 関数の `return 0;` を省略してよいことになりました。（C++ でも同じです。Java の main 関数は元々 void 型です。）

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[*6]。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 **NULL と比較**して検査します。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;」でも中断になりますが、通常は main() 以外の関数で行うので 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 ヶ所で必要です。これらが食い違うことのないよう、5 行目で変数 filename に代入してから使っています。**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`"AAA"`の文字列リテラル、fputc() は `'A'` 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

[*6] 混ぜてはいけない関数群もあります。例えば、`FILE*` によらず、バッファリングせずにバイト列を入出力する低水準関数があります。

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

```c
#include <stdio.h>
#include <stdlib.h> // exit()

int main(void) {
    char filename[] = "output.txt";  // 書き込むファイル名

    /* 前処理 */
    FILE *fp = fopen(filename, "w");
    if (fp == NULL) {
        printf("ファイルオープンに失敗しました '%s'\n", filename);
        exit(EXIT_FAILURE);  // プログラムを中断する（異常終了）
    }

    /* 主処理 */
    fputs("九九の表\n", fp);
    for (int i=1; i<=9; i++) {
        for (int j=1; j<=9; j++) {
            fprintf(fp, "%2d ", i*j);
        }
        fputc('\n', fp);
    }

    /* 後処理 */
    fclose(fp);
    return EXIT_SUCCESS; // 正常終了
}
```

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閲覧するときの注意事項です。（Unix 系や Mac は、ファイルロックの仕組みが違うので、**常に後者**になります。）

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、再実行する前に**ファイルを閉じる**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、**ファイルを開き直す**必要があるでしょう。（書き換えを検知して、開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

既存のファイルから 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。

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

- 正常に 1 文字を読み取ると、その文字コードを unsigned char の値として返します。
- 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) {


    ...;
```

```
/* 分解 */
for (;;) { // 無限ループ
    c = fgetc(fp);
    if (c == EOF) break;
    ...;
```

17 行 c を画面に表示[*7]します。これを繰り返して、ファイル全体を表示します。

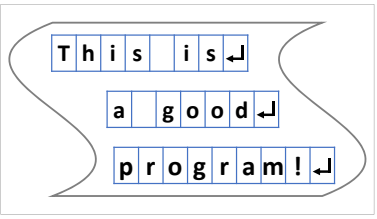
このプログラムを実行するには、**事前に**ファイル"input.txt"を作っておきます。実行すると、このファイルの内容がそのまま表示されるはずです。

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[*8]。

```
        printf("%c", toupper(c));  // 17行目の差し替え
```

[*7] (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。

[*8] 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）

```
#include <stdio.h>
#include <stdlib.h> // exit()

int main(void) {
    char filename[] = "input.txt";  // 読み取るファイル名

    /* 前処理 */
    FILE *fp = fopen(filename, "r");
    if (fp == NULL) {
        printf("ファイルオープンに失敗しました '%s'\n", filename);
        exit(EXIT_FAILURE);  // プログラムを中断する（異常終了）
    }

    /* 主処理 */
    int c;                // 読み取る1文字を保存する（char ではない）
    while ((c = fgetc(fp)) != EOF) { // EOF はファイルの終端の目印
        printf("%c", c); // cの1文字を画面表示
    }

    /* 後処理 */
    fclose(fp);
    return EXIT_SUCCESS; // 正常終了
}
```

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。☞11.3.6 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処理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。

```
#include <stdio.h>

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[*9]。

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[*10]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[*11]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 行の長さの最大のものを想定しておきます。とは言っても、わからないことも多いので、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。

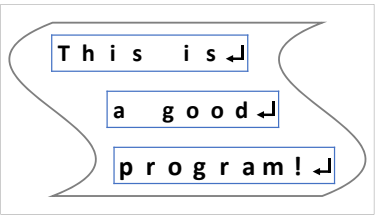
行単位で読み取るようになったので、各行に整数値の書いてあるファイルを読み取って、合計と平均を計算してみましょう。文字列から int への変換は <stdlib.h> の atoi() という関数[*12]が便利です。ソースコード 11.3 の主処理（17–20 行）を以下で差し替えます。

[*9] ファイルの末尾に'\n' がないときや、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。

[*10] 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。

[*11] 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。

[*12] long への変換なら atol()、double なら atof() です。☞B.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> // exit()
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（17〜20行の差し替え） */
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

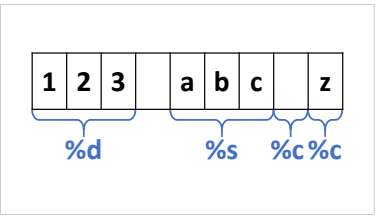
```
'1
'を読み取りました
'3
'を読み取りました

～(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 行をシングルコーテーション (') 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

この処理ではデータ 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もあります。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、それほどありません。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。

```
#include <stdio.h>

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、それの文字列版といったところです。A.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() を組み合わせると、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すみます。

また scanf() 系の関数で文字列を"%99s"などで受け取る際には、このように**長さの上限を設定**する必要がありますが（☞B.8 節）、sscanf() なら fgets() の段階でも制限されるので、直接 fscanf() で受け取るよりも安全です。

11.3.5　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、ファイルポインタ変数を 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 では変数名を 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 }
```

11.3.6　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。

```
#include <stdio.h>

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() などで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
    ...
}
```

11.4 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、キーボードから対話的に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** (command line argument) 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、以下のようにします。

```
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。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）をすれば、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。（☞A.1 節）

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

ここまで、ファイル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と、現実は少し違います。

エラーチェック付きオープン　オープン失敗時に単純にプログラムを中断させてよいなら、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。（☞A.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[*13]。

[*13] 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. **[複数の手段の使い分け（☞11.3.1 項）]**
   （a）`fputc(c, fp)` は `fprintf(fp, "%c", c)` で代用できる。
   標準ライブラリに、この単機能の関数がなぜ用意されているのか、考察せよ。
   （b）数学関数の `sqrt(x)` は `pow(x, 0.5)` で代用できる。
   標準ライブラリに、この単機能の関数がなぜ用意されているのか、考察せよ。
2. **[文字単位の読み取り（☞11.3.2 項、189 ページと 192 ページの頻出ミス）]**
   189 ページのソースコード 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）
3. **[行単位の読み取り（☞11.3.3 項）]**
   191 ページのソースコード 11.3 を参考に、1 行に 1 つの整数の記述されたファイルを読み取り、読み取った数値の**最大値**と**最小値**を表示せよ。
4. **[ファイル出力の使い道]**
   画面（標準出力）に表示される文字は、リダイレクトの機能（☞A.8.3 項）を使えば、簡単にファイルに保存できる。では、プログラムで `fopen(outfile,"w")` などを用いてファイルを保存することが有益なのは、どのような場面か、例を挙げよ。**ヒント**：プログラム中で扱うファイルがいくつかに着目せよ。

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。）
- 末尾の改行文字（`'\n'`）を削除します。

しかし、gets() の代替品を標準ライブラリ関数から探しても、末尾の `'\n'` を削除するものが見当たらず、fgets() の後に次のような処理を加えるしかなさそうです。

```
while (fgets(buff, sizeof(buff), stdin) != NULL) {
    int len = strlen(buff);
    if (len > 0 && buff[len-1] == '\n') { len--; buff[len]='\0'; }
    ...
}
```

なお、新設された gets_s() はオプション扱いのため、例えば 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）の自動精算機の動作を模倣（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精算機に通します。
- 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 日のうちに必ず精算することにします。駐車料金は「10 分まで 100 円」を繰り返し適用します。（つまり 11 分は 200 円です。）支払いに使えるお金は、5000 円，1000 円，500 円，100 円の 4 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。

コラム： 単体テスト・結合テスト

関数を一つ作るごとに、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（☞72 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、テストそのものも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。

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(5000, 1000, 500, 100円のみ) => 10
お金を投入してください(5000, 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(5000, 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(5000, 1000, 500, 100円のみ) => 1000
お釣りは 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。0 秒は 0 円です。マイナス秒は考えなくて構いません。

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、小数点以下は切り捨てられます[a]。

整数に切り上げる a/n の慣用句は「`(a+n-1)/n`」です。n-1 の意味は、2 章の練習問題 7. を思い出して考えてください。

[a] 数学関数の `ceil()` は（double のまま）整数値へ切り上げます。しかし、さらに int への型変換も必要ですから、かなり遠回りになります。

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論理型の `TRUE` を、そうでなければ `FALSE` を返します。

value が 5000, 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 217 ページのソースコード A.4 の `input_int()` を使えばいいでしょう。受け取り可能な金額かどうかの判定に `is_coin_note()` を呼び出します。**必ず return に到達する**関数になるよう注意してください。

受け取れない金額に「受け取れません」と表示することは、最初の段階では考えなくて構いません。（6.4.3 項のように、例外的な処理が必要になります。）

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

が起こり、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)`[b]
  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。
  （☞34 ページの問題 6.）

[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。この値を 32 ビット符号付きの型で扱うと、2038 年 1 月 19 日 3 時 14 分 7 秒の次にオーバーフローするというのが「2038 年問題」です。

[b] 本来は h 時 m 分 s 秒に変換した h, m, s の 3 つの値を返したいところですが、関数 1 つで 3 つの値を返す手法をまだ習得していませんので、ここでは即座に表示することにします。

9 章

- `void put_change(int x)`
  x 円のお釣りを返すのに、1000 円札と 500 円と 100 円硬貨を組み合わせて、「a 円 b 枚」の形式で表示します。紙幣と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 6. 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しているところを、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。こうしておくと、紙幣や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

- `struct tm get_current_time(void)`
  現在時刻を返します。実装は以下の通りとします。

```
#include <time.h>
struct tm *get_current_time(void) {
    time_t t = time(NULL);
    return localtime(&t);
}
```

戻り値の `struct tm` は <time.h> で定義されていて、メンバの `tm_hour` に時 (0–23)，`tm_min` に分 (0–59)，`tm_sec` に秒 (0–59) が入ります。(☞A.11 節)
204 ページのソースコード 12.1 も参考にして、main() 関数からこれを呼び出し、現在時刻を表示してください。

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、次の部品の関数と `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。

12 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、現在時刻との差分から経過秒数を求め、駐車料金を決定して返します。
- `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。

3 章で作った `sec2fee()` などを呼び出すと、簡単に駐車料金が決定できます。お金を受け取るためには、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。

```
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
}
```

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やプロトタイプ宣言も参考にしてください。

ソースコード 12.1　駐車場の自動精算機（概形）

```
#include <stdio.h>  // BUFSIZ
#include <stdlib.h> // atoi(), exit()
#include <time.h>   // time(), struct tm

#define FALSE 0
#define TRUE  1

/* プロトタイプ宣言 */
int input_int(void);
int sec2fee(int sec);
int is_coin_note(int value);
int get_coin_note(void);
int hms2sec(int h, int m, int s);
int sec_diff(int h1, int m1, int s1, int h2, int m2, int s2);
void put_change(int x);
struct tm *get_current_time(void);
int decide_fee(void);
int receive_fee(int fee);

/* 付録A さらなる成長に向けて */
int input_int(void) {
    char buff[BUFSIZ];
    if (fgets(buff, BUFSIZ, stdin) == NULL) exit(EXIT_FAILURE);
    return atoi(buff); // 文字列→int変換
}

/* 3章 関数 (1) */
int sec2fee(int sec) {
    /* ここを作る */
}

/* 6章 繰り返し処理 (2) */
int is_coin_note(int value) {
    /* ここを作る */
}

/* 6章 繰り返し処理 (2) */
int get_coin_note(void) {
    /* ここを作る */
}

/* 7章 関数 (2) */
int hms2sec(int h, int m, int s) {
    /* ここを作る */
```

```
}

/* 7章 関数（2） */
int sec_diff(int h1, int m1, int s1, int h2, int m2, int s2) {
    /* ここを作る */
}

/* 9章 文字列とポインタ */
void put_change(int x) {
    printf("お釣りは %d 円です。\n\n", x);
    /* ここを作る */
}

/* 10章 構造体 */
struct tm *get_current_time(void) {
    time_t t = time(NULL);
    return localtime(&t);
}

/* 12章 最終目標 */
int decide_fee(void) {
    printf("駐車開始時刻を入力してください\n");
    printf("時 => ");  int h = input_int();  // 付録A
    printf("分 => ");  int m = input_int();
    printf("秒 => ");  int s = input_int();

    struct tm now = *get_current_time();  // 10章
    printf("現在時刻は %2d:%02d:%02d です\n",
           now.tm_hour, /* ここを作る */ 0, 0);

    int sec = /* ここを作る */ 0;
    printf("経過時間は %d 秒です。\n", sec);

    int fee = /* ここを作る */ 0;
    printf("駐車料金は %d 円です。\n\n", fee);
    return fee;
}

/* 12章 最終目標 */
int receive_fee(int fee) {
    /* ここを作る */
}

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
}
```

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、単に画面に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。

12

[*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。

付録 A

さらなる成長に向けて

ここでは、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。

作ったプログラムが思い通りに動作すれば、それでよいのでしょうか。確かに最初の目標としてはよいのですが、会社のチームでプログラムを共同で開発したり、あるいは個人の活動でもオープンソースソフトウェアとして公開すると、他人からの改良提案をもらったりして、一般的なコーディングスタイルに合わせていく必要が生じます。「動く」だけのプログラムではなく、「メンテナンスしやすい」「共同開発しやすい」プログラムの作法も、徐々に身につけてもらえたらよいでしょう。

後半では、C 言語のテクニカルな性質を説明します。理解できると有用なのですが、幅広い知識が必要なため、資料として参照してもらえば十分です。

コラム： クマさんに相談

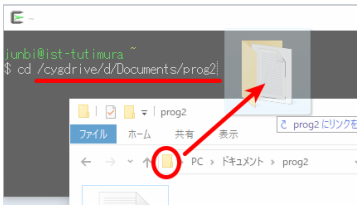
思うようにプログラムが動かず、原因がわからなくて頭を抱えているときには、誰かに相談したくなります。ある研究室では、部屋の片隅にクマのぬいぐるみが置いてあって、まずクマさんに相談することになっています。それでも解決しなければ、ようやく研究室のメンバーに相談します。

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、それだけで問題点が見えてきます。

A.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
  - フルパス名を入力：アイコンのドラッグ



- テキストエディタの基本操作（☞110 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

A.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（☞234 ページの B.11 節）
- デバッグライトを活用する（☞83 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、実演しようとすると、「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。

A.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。

- スペーシング（☞6 ページの 1.3.5 項、81 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（☞43 ページのコラム、91 ページの 6.1.1 項）
- 変数の初期化は使う直前（☞81 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）
- 変数名・関数名の命名規則（☞118 ページの 7.3 節）
- マジックナンバーを排除（☞124 ページの 8.2 節）
- 見慣れた処理を組み合わせる（☞79 ページの頻出ミス、87 ページの 5.4 節）
- コメントでは見慣れない動作の理由を説明する（☞149 ページのコラム）

A.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　3.6 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ことになって、上記の「同じ処理は 2 度書かない」を破ることにもなります。

コード・リーディング　身近な人とプログラムを読み合いましょう。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。

A.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。

↑ 変化の頻度が少ない

- 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

↓ 変化の頻度が多い

A.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 199 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（☞72 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 プログラムを、変数の値などを確認しながら、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

メモリリーク検出 配列あふれなど、メモリの不正アクセスを検出するツールです。方式がいくつもあり、万能のものは存在しません。例: Electric Fence, 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にも有用です。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。

A.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。double には、図 A.1 のように 2 進数表記で、符号に 1 ビット、指数部に 11 ビット、仮数部に 52 ビットを割り当てるのが典型的です。

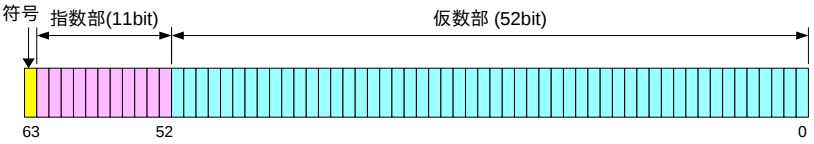


図 A.1　IEEE 754 による double のビット割り当て

このように精度が限られるため、数値誤差が常に付きまといます。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。

A.7.1　定数の誤差・丸め誤差

浮動小数点が 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 には定数としての誤差**が含まれます。循環小数を途中で打ち切っているためです。そして、例えばソースコード A.1 は 0.1 を 10 回足していますが、1.0 になりません。演算ごとに**最下位ビットの丸め誤差**も加わるためです。

ソースコード A.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 A.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、統一的な手段は確立されていません。

ソースコード A.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 A.2　浮動小数点の誤差を見込んだ比較

```
#include <math.h>  // fabs(), fmax()
#include <float.h> // DBL_EPSILON

/* 等しいとみなす相対誤差 */
#define RELATIVE_ERROR (DBL_EPSILON * 8)

/* xとyがほぼ等しければ0，x>yなら正，x<yなら負を返す */
int fcmp(double x, double y) {
    double diff = x - y;
    double max  = fmax(fabs(x), fabs(y));        // 絶対値の大きい方
    if (fabs(diff) < max * RELATIVE_ERROR) return 0; // 許容誤差以下
    if (diff > 0) { return +1; } else { return -1; }
}
```


A.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 → 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 → 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。

A.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。

A.7.4　定数

浮動小数点型には、23 ページの表 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。

正の最小の正規化数 (normalized number) 浮動小数点数は、仮数部が 1 以上かつ基数未満になるよう、指数部で調節して正規化しますが、限界もあります。DBL_MIN は double の限界値を表す定数です。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からです。

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前の今では 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() で変更でき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。

A.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、簡単にすます方法があります。ファイルの出力にも、簡便な方法があります。

A.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。スペース文字が区切りとなって、第 1 引数, 第 2 引数, ... と分解されます。プログラムで受け取るためには、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）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 で扱うためには <stdlib.h> の関数を利用して atoi(argv[1]) と変換します。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を付け忘れたときの対策として、argc が必要数ではない時に**使用法を表示してプログラムを中断**するのが習慣です。配列の添字溢れを防ぐためです。

ソースコード A.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) {        // 引数が2個以外なら
6          printf("使用法: %s 整数 小数\n", argv[0]); // コマンド名
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("a = %d, b = %f\n", a, b);
12 }
```

A.8.2　キーボード

キーボードからの入力は scanf() で受け取るのが簡便ですが、**問題も多く**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代入されます。

scanf() は、printf() と似た書式文字列に従って、キーボードから受け取った数値や文字列を、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 ☞B.8 節
- % ではない、普通の文字に対しては、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　引数をアドレスで渡すために & が必要です。さらに double の書式には l（エル）を追加して %lf にします。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[*3]。

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で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、その後に 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 A.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。

[*3] 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 A.4　キーボードから数値を受け取る

```
#include <stdio.h>
#include <stdlib.h> // atoi(), atof(), BUFSIZ

int input_int(void) {
    char buff[BUFSIZ];
    if (fgets(buff, BUFSIZ, stdin) == NULL) exit(EXIT_FAILURE);
    return atoi(buff); // 文字列→int変換
}

double input_double(void) {
    char buff[BUFSIZ];
    if (fgets(buff, BUFSIZ, stdin) == NULL) exit(EXIT_FAILURE);
    return atof(buff); // 文字列→double変換
}

int main(void) {
    printf("整数を入力してください> ");
    int a = input_int();

    printf("小数を入力してください> ");
    double b = input_double();

    printf("a=%d, b=%f\n", a, b);
}
```

A.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、シェルから起動するプログラムには、たいてい**キーボード**と**コンソール画面**を、**ファイル**として統一的に扱う仕組みがあります。

シェル（コマンドインタプリタ）は、プログラム起動時に、表 A.1 の 3 つのファイルを開きます。通常はキーボードやコンソール画面に接続されていますが、本物のファイルに切り替えることもできます。この操作を**リダイレクト** (redirect) といいます。

標準入力 (standard input)　ファイルポインタ名は `stdin` です。ですから `scanf("...")` と `fscanf(stdin, "...")` が同じです。普段はキーボードにつながっていますが、あらかじめ入力する内容を`"input.txt"`のようなファイルに記録しておけば、プログラム起動時の不等号でリダイレクトするだけで、キー入力したことになります。

```
$ ./a.exe < input.txt // input.txtにキー入力する内容を保存してから
```

標準出力 (standard output)　ファイルポインタ名は `stdout` です。通常は画面に表示されますが、ファイルに保存したければ、やはり（逆向きの）不等号でリダイレクトします。

```
$ ./a.exe > output.txt // 画面出力を output.txt に上書き保存
```

この操作では、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。既存ファイルに追記するなら >> と不等号を二重にします。

標準入力と標準出力の**両方を同時にリダイレクト**できます。

```
$ ./a.exe < input.txt > output.txt // 入出力同時にリダイレクト
```

標準エラー出力 (standard error output)　画面出力のうち、エラーメッセージや、時間のかかる処理の途中経過など、ファイルにリダイレクトされたくないものに使います。

2 つのプログラムで、標準入力と標準出力を直接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て、複雑な動作を実現することは、UNIX の思想に合致します。

表 A.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

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 (EOF) を示すことができるはずです。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。

A.9 疑似乱数

擬似乱数 (pseudo-random number) を生成する関数が <stdlib.h> にあります。再現可能であるため、正確には擬似乱数ですが（☞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得られます**。RAND_MAX $\geq 2^{15} - 1$ (= 32767) であることが言語規格で保証されています。欲しい乱数の範囲や種類に合わせて、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[*4]。

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、目標の範囲に入ります。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まであったため、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 a2 では、まず frand() を 6 倍して、0 以上 6 未満の浮動小数点数を得ます。そして int にキャストして小数部分を切り捨てます。これで目的の範囲の整数が得られます[*5]。

1 以上 6 以下 (int)

```
int b1 = 1 + (rand() % 6);          // x + (rand() % (y-x+1))
int b2 = 1 + (int)(frand() * 6);    // x + (int)(frand() * (y-x+1))
```

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。

[*4] Java なら Math.random() が同じ範囲の乱数を生成します。

[*5] 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 になることに注意してください。

A.9.1　乱数系列

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が同じになります。右の例は GCC 7.5 (glibc-2.27) の場合です。srand() で指定しなければ、種は 1 が使われます。

```
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() で 1 度だけ**行なえば十分です。
- `<time.h>` の `time(NULL)` の値を種にすると、典型的には 1 秒ごと[*6]に異なる乱数系列になります。(☞13 ページのソースコード 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[*7]を用意します。

A.10　関数の引数の可変長

printf() や scanf() 関数では、状況によって**引数の個数や型**が変わります。このような引数を**可変長引数** (variable length arguments) といいます。本書の守備範囲を越えますが、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。

可変長引数をとる関数では、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[*8]。

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 (☞B.8 節)。しかし、scanf() 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。

[*6] 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。)
[*7] 松本 眞氏の Mersenne Twister がおすすめです。
[*8] 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。

A.11 時刻

<time.h> に時刻を扱う関数があります。現在時刻と、プログラムの消費したプロセッサ時間の、2 通りの時刻を扱います。

プロセッサ時間（CPU タイム）は、入出力待ち（例えばキー入力待ち）の間は止まります。逆にマルチスレッドで動作すれば、実際の経過時間よりも早く進みます。

```
#include <time.h>

定数
  CLOCKS_PER_SEC
  TIME_UTC
```

```
型
  clock_t
  time_t
  struct tm
  struct timespec
```

```
関数
  clock_t clock(void);
  time_t time(time_t *timer);
  int timespec_get(struct timespec *ts, int base);

  double difftime(time_t time1, time_t time0);

  time_t mktime(struct tm *timeptr);
  struct tm *gmtime(const time_t *timer);
  struct tm *localtime(const time_t *timer);
```

clock() プログラムの消費したプロセッサ時間を返します。戻り値は clock_t 型で、CLOCKS_PER_SEC で割ると秒単位の値になります。プログラムの実行開始後に **0 から始まるとは限らない**ので、プログラムの 2 ヶ所での差を取る必要があります。現実の環境では精度が 0.01 秒ぐらいと（コンピュータの動作速度からすれば）粗いことも多く、0 秒と測定されることもよくあります。

```
    clock_t start = clock();
    routine_taking_long_long_time(); // 時間のかかる処理
    clock_t end = clock();
    printf("所要時間は %g 秒です。\n",
           (double)(end - start) / CLOCKS_PER_SEC);
```

time() 現在時刻を time_t 型で返します。ポインタ型の引数にも（同じ値を）書き込みます。ただし NULL を指定すれば書き込みません。戻り値か引数か、どちらか片方を使用すれば十分です。

```
time_t t = time(NULL); // 戻り値
  または
time_t t;  time(&t);   // 引数
```

time_t は、典型的に long（32bit または 64bit）が割り当てられ、Unix 系の時刻と同じく、協定世界時（UTC）の 1970 年 1 月 1 日 0 時 0 分 0 秒からの経過秒数を格納することが多いですが、言語規格上は、型も起点も精度も、何も規定されていません。現実には、1 秒単位であることを前提にしたプログラムもよく見かけます。

difftime() 引数で受け取った 2 つの時刻の差を、double 型の秒数で返します。時刻は time_t 型なので、この関数の内部で double 型への変換が行われることになります。

```
time_t start = time(NULL);
routine_taking_long_long_time(); // 時間のかかる処理
time_t end = time(NULL);
printf("経過時間は %g 秒です。\n", difftime(end, start));
```

mktime() 構造体 struct tm のポインタを引数で受け取り、格納されている時刻を time_t 型に変換して返します。struct tm には右のメンバーが規定されていますが、ここでは tm_wday と tm_yday は利用されません。
次の localtime() 関数と、ちょうど逆の動作をします。

```
// struct tm のメンバ
int tm_sec;  // 秒 [0, 60]
int tm_min;  // 分 [0, 59]
int tm_hour; // 時 [0, 23]
int tm_mday; // 日 [1, 31]
int tm_mon;  // 月 [0, 11]
int tm_year; // 西暦年-1900
int tm_wday; // 曜日 [0, 6]
int tm_yday; // 通算日数 [0,365]
int tm_isdst;// 夏時間フラグ
```

gmtime(), localtime() time_t 型の時刻をポインタ引数で受け取り、構造体 struct tm に変換して、ポインタで返します。struct tm の実体は、固定的な領域に割り当てることが許されています。つまり、繰り返し呼び出すと上書きされる可能性があるので、すぐにローカル変数にコピーしてから用います。gmtime() は協定世界時、localtime() は現地時刻に変換します。

```
time_t t1 = time(NULL);
struct tm s1 = *localtime(&t1);  // ポインタの指す実体をコピー
printf("今日は%d年%d月%d日です。\n",
       1900 + s1.tm_year, 1 + s1.tm_mon, s1.tm_mday);
```

tm_year は、西暦年数から 1900 を引いた値です。
tm_mon は 0 オリジンで数えます。（☞8.4.2 項）
tm_wday は、0 が日曜日を示します。1 が月曜日です。
tm_yday は、1 月 1 日を 0 として、経過日数を数えたものです。
tm_sec は、通常は 59 までです。60 は閏秒のために用意されています。

timespec_get()　C11 で追加された、time() の高精度版です。新しい関数だけあって、関数名が「引数の構造体名 + 動詞」と、現代的な命名規則（☞10.5 節）に則っています。

```
struct timespec ts;
timespec_get(&ts, TIME_UTC)
double sec = ts.tv_nsec * 1e-9;
```

第 1 引数の timespec 構造体の変数には 2 つのメンバが規定されています。time_t 型の tv_sec には time() に相当[*9]するものが、long 型の tv_nsec には秒未満がナノ秒単位で格納されます。tv_nsec に 10^{-9} を掛けると秒単位の小数部分になります。

第 2 引数には、定数 TIME_UTC[*10]を指定します。

この関数をサポートしている環境はまだ少なく、また（言語規定による）精度の保証もありませんが、今後は利用価値が高まるものと思われます。

A.11.1　実行時間の測定

プログラム全体の実行時間を測るのであれば、Unix 系の time コマンドも有用で簡便です。`time ./a.out` とすれば、a.out の実行にかかったプロセッサ時間などが表示されます。time コマンドは、シェルの種類によっては内部コマンドにもあるので、外部コマンドを明示するには `/usr/bin/time ./a.out` とフルパスで指定します。外部コマンドのほうが、表示内容が詳細です。

[*9] 言語仕様上は、time() と型は同じですが、値まで同一であるとは定められていません。起点や精度が異なる可能性を考慮したものと思われます。

[*10] C11 では、これ以外の定数は用意されていません。C23 では種類が増えて、時間の測り方を指示できるようになる見込みです。

付録 B

一覧表

B.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	:	コロン
/	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		

B.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きました。

B.3 エスケープシーケンス

「\」で始まる文字の並びは**エスケープシーケンス**と呼ばれ、画面には表示できない制御文字を表すなど、以下のような特別な機能があります。

表記	ASCII コード	意味
\a	0x07	警報音 (BEL)
\b	0x08	後退 (BS)
\t	0x09	水平タブ (HT)
\n	0x0a	改行 (LF)
\v	0x0b	垂直タブ (VT)
\f	0x0c	改頁 (FF)
\r	0x0d	復帰 (CR)
\\	0x5c	文字としての \
\'	0x27	文字としての '
\"	0x22	文字としての "
\?	0x3f	文字としての ?
\0	0	ヌル文字 (NUL)
ooo	0*ooo*	文字コードが 8 進数 (*ooo*)
\x*hh*	0x*hh*	文字コードが 16 進数 (*hh*)

「文字としての」というのは、特別な機能を打ち消した、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るので、この文字自体を表すには、「\\」と、2 度繰り返します。
  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の機能があるので、文字定数にこの文字自体を含めるには、「\」を付けて終了の機能を打ち消します。
  「"」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。今ではほぼ不要なため、コンパイラによっては独自に無効にしている場合もありますが、「\」を付けると可搬性が高まります。
  例 (\?)：`printf("答えはいくつでしょう?\?2\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、C23 で廃止される見込みです。

B.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： 文字コード依存の処理

`'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）

B.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	→
x [y]	配列参照
x (y)	関数呼び出し
x . y	構造体メンバ参照
x -> y	構造体メンバ間接参照
x ++	後置インクリメント
x --	後置デクリメント
(T){ x } *C99*	複合リテラル
単項演算とキャスト	←
++ x	前置インクリメント
-- x	前置デクリメント
& x	アドレス
* x	間接参照
+ x	単項プラス
- x	単項マイナス
~ x	1 の補数（ビット反転）
! x	論理否定
(T) x	キャスト
sizeof x	サイズ
sizeof(T)	サイズ
_Alignof(T) *C11*	境界調整サイズ
四則演算	→
x * y	乗算
x / y	除算
x % y	剰余
x + y	加算
x - y	減算

（↗ 右上に続く）

（↗ 左下より続く）

ビットシフト	→
x << y	左シフト
x >> y	右シフト
比較演算	→
x < y	小なり
x > y	大なり
x <= y	以下
x >= y	以上
x == y	等しい
x != y	異なる
ビット演算	→
x & y	ビット AND
x ^ y	ビット XOR
x | y	ビット OR
論理演算	→
x && y	論理積
x || y	論理和
条件演算	←
x ? y : z	条件
代入演算	←
x = y	代入
x ⊗= y	複合代入
⊗は右のいずれか	* / % + - << >> & ^ |
順次評価演算	→
x , y	順次評価

- 　は演算子、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。

B.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、戻り値の型と同じです。
- 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	
int	abs(x)
long	labs(x)
long long	llabs(x) *C99*

絶対値（正確には <math.h> の関数）	
float	fabsf(x) *C99*
double	fabs(x)
long double	fabsl(x) *C99*

文字列→整数（簡易）	
int	atoi(s)
long	atol(s)
long long	atoll(s) *C99*

文字列→浮動小数点数（簡易）	
—	—
double	atof(s)
—	—

文字列→整数	
—	—
long	strtol(s,p,b)
long long	strtoll(s,p,b) *C99*

文字列→浮動小数点数	
float	strtof(s,p) *C99*
double	strtod(s,p)
long double	strtold(s,p) *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 strtoul() と strtoull() 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 ☞A.9 節

B.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。（☞B.11 節）

- 以下の関数は、戻り値と引数の x, y のいずれも 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	
fabs(x)	絶対値 $\lvert x\rvert$
fmod(x,y)	$x \div y$ の剰余（x と同符号）
fmax(x,y)^{C99}	x と y の最大値
fmin(x,y)^{C99}	x と y の最小値
整数への丸め	
ceil(x)	切り上げ $\lceil x\rceil$
floor(x)	切り捨て $\lfloor x\rfloor$
trunc(x)^{C99}	0 方向への切り捨て
round(x)^{C99}	絶対値の四捨五入[*4]
べき乗（累乗）	
pow(x,y)	べき乗 x^y
sqrt(x)	平方根 $\sqrt{x}$
cbrt(x)^{C99}	立方根 $\sqrt[3]{x}$
hypot(x,y)^{C99}	直角三角形の斜辺長 $\sqrt{x^2+y^2}$

三角関数（角度はラジアン）	
sin(x)	x[rad] の正弦
cos(x)	x[rad] の余弦
tan(x)	x[rad] の正接
asin(x)	x の逆正弦 $[-\pi/2, +\pi/2]$
acos(x)	x の逆余弦 $[0, \pi]$
atan(x)	x の逆正接 $[-\pi/2, +\pi/2]$
atan2(x,y)	y/x の逆正接 $[-\pi, +\pi]$
指数関数・対数関数	
exp(x)	指数関数 e^x
exp2(x)^{C99}	2^x
log(x)	自然対数 $\ln(x)$
log10(x)	常用対数 $\log_{10}(x)$
log2(x)^{C99}	二進対数 $\log_2(x)$

- 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}	
fpclassify(x)	FP_NAN などの定数
isnan(x)	非数なら真（0 以外）
isinf(x)	$\pm\infty$ なら真（0 以外）
isfinite(x)	非数, $\pm\infty$ 以外なら真
isnormal(x)	正規化数なら真
signbit(x)	負 ($-\infty$, -0 含む) なら真

浮動小数点数の種別を表す定数 ^{C99}	
FP_NAN	非数 (Not a Number)
FP_INFINITE	$+\infty$ または $-\infty$
FP_ZERO	$+0$ または -0
FP_SUBNORMAL	非正規化数（絶対値が正規化数の最小値以下）
FP_NORMAL	上記以外（正規化数）

[*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、戻り値と引数の double 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（エル）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、接尾辞の代わりに、引数の型で切り替わるようになります。tg は type-generic（総称型）の意味です。

[*4] round(x) と同じ値の整数値を、lround(x)^{C99} は long 型で、llround(x)^{C99} は long long 型で返します。

B.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。

	整数[*5]					浮動小数点数[*6]			文字（列）	
変数の型	short	int	long	long long	size_t[*7]	float	double	long double	char	char*
printf 系	%hd	%d	%ld	%lld C99	%zu C99	%f (%lf C99[*8])		%Lf C99	%c	%s
scanf 系	%hd	%d	%ld	%lld C99	%zu C99	%f	%lf	%Lf C99	%c	%s[*9]

- 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（☞A.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。（☞B.11 節の `-Wall` や `-Wformat`）

*5 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 printf 系では % 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*.*y* d の形式で書きます。*x* も .*y* も省略可能です。scanf 系では、読み取る最大文字数 *x* のみを指定できます。

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 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。（ただし scanf 系では動作に影響せず、どの浮動小数点の形式でも読み取ります。）また printf 系では % 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*.*y* f 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。scanf 系では、読み取る最大文字数 *x* のみを指定できます。

f, F	小数表記
e, E	指数表記
g, G	f と e の自動切り替え
a, A	16 進指数表記 C99

例:("%6.3f", 1.2) → "␣1.200"
例:("%6.3e", 1.2) → "1.200e+00"
例:("%6.3g", 1.2) → "␣␣␣1.2"

*7 size_t 型は符号なしのため、表には符号なし 10 進数を前提とした zu を記載しましたが、文法上は符号ありの zd や、当然ながら 8 進数の zo なども可能です。

*8 printf 系は可変長引数（☞A.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。

*9 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。）

表には記載していませんが、ポインタ変数を %p で扱えます。printf 系の表示形式に決まりはありません。scanf 系では、その形式を読み取れるのですが、読み取ったアドレスを参照すると不正アクセスになりがちですので、注意してください。

B.9　文字の分類・変換 <ctype.h>

標準ライブラリにある、1 文字単位で分類と変換を行う関数のすべてです。ソースコードに #include <ctype.h> が必要です。*C99* 印は C99 で導入されました。

- いずれの関数も、戻り値と引数は int です。
- 引数の c は、unsigned char の値か、EOF のいずれかで動作が保証されます。

分類	c が該当すれば真（0 以外）、それ以外で偽（0）を返す	
isalpha(c)	英字	英小文字か英大文字
isalnum(c)	英数字	英小文字か英大文字か数字
islower(c)	英小文字	（a〜z）
isupper(c)	英大文字	（A〜Z）
isdigit(c)	数字（10 進数）	（0〜9）
isxdigit(c)	16 進数	（0〜9，a〜f，A〜F）
isblank(c)*C99*	空白	スペース (␣) かタブ (\t)
isspace(c)	間隔	スペース (␣)，タブ (\t)，改行 (\n) など
iscntrl(c)	制御文字	☞B.2 節（表中の 2 文字以上のもの）
isprint(c)	表示文字	制御文字以外（スペース (␣) は含まれる）
isgraph(c)	図形文字	制御文字以外から、スペース (␣) を除く
ispunct(c)	区切り文字	制御文字以外から、スペース (␣) と英数字を除く
変換	c を変換して返す（変換できなければそのまま返す）	
tolower(c)	小文字に変換	
toupper(c)	大文字に変換	

B.10　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。このヘッダを読み込んだときのみ有効で、`bool` は `_Bool` の別名です。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。今後策定される C23 では、予約語に昇格する見込みです。

B.11　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。

`-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。（「`O0`」は「オー・ゼロ」です。）

`-O2`　最適化を行います。`-Wall` と組み合わせると、初期化忘れなどの警告が増えます。

`-g`　デバッグ情報を埋め込みます。実行時エラーが詳細に表示されます。

`-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`　中間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*. https://www.open-std.org/JTC1/SC22/WG14/www/docs/n1256.pdf, September 2007.

[6] WG14/N1570. *Programming languages — C*. https://www.open-std.org/JTC1/SC22/WG14/www/docs/n1570.pdf, April 2011.

[7] WG14/N3047. *Programming languages — C*. https://www.open-std.org/JTC1/SC22/WG14/www/docs/n3096.pdf, April 2023.

[8] 株式会社アンク. **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] 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.

[10] 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.

[11] 高橋 麻奈. **やさしい C 第 5 版**. SB クリエイティブ, 2017.

[12] 松浦 健一郎, 司 ゆき. **C 言語 [完全] 入門**. SB クリエイティブ, 2022.

[13] 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.

ソースコード一覧

索引

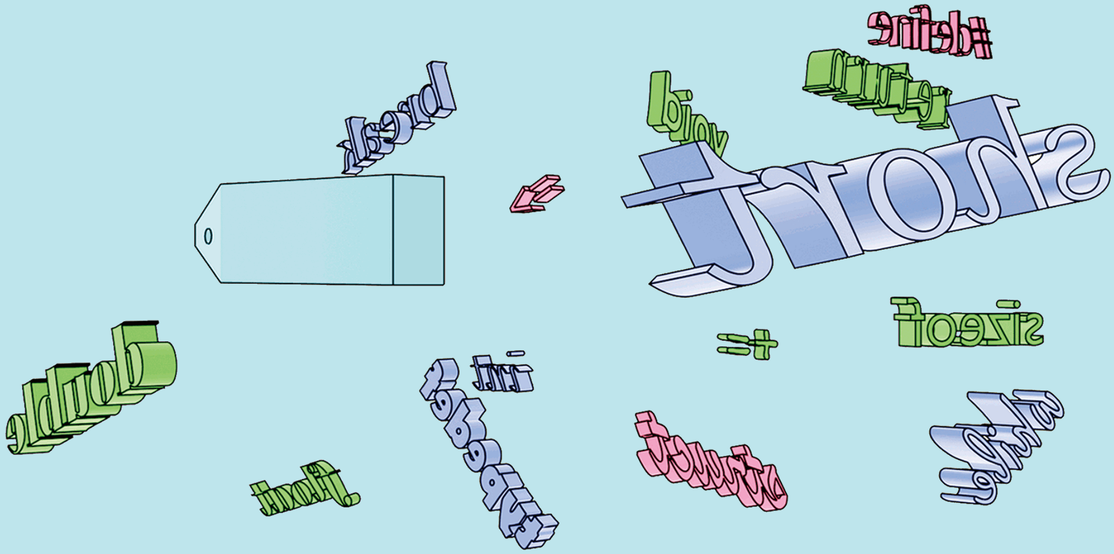
著者

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)



#define
return
void
break
short
sizeof
+=
int
double
float
unsigned
struct
while