

理論と実習の両面から学ぶ

C言語総合講座15講 ~理論編~

Theorems in
C

達哉ん 著

はじめに

プログラミングは楽しい。その思いを伝えたいと思って、後輩にC言語を教えてきた。意欲ある後輩がプログラミングを嫌いにならぬよう、実力がつくよう、微力ながら手助けができればと思って、多くのプリントを作り、解説をしてきた。それをひと通り集めたテキストを書こうと考えて執筆したのがこのテキストである。

実は、このテキストの執筆を始める2年ほど前から、別のテキストを執筆して、教える際に使っていた。そのテキストは高校生を対象にしたものであるが、無料という点を除いてさしたる成功を収めたとは言いがたいものであった。自身がテキストを用いて解説すると不満がよくわかる。それらの不満を解消し、また、様々な制約を外し、自身が教えるスタイルをそのままテキストにしよう、と心がけた。

理科系の大学生を対象にして書いたものであるが、これは具体的には以下のような知識・技能・環境を前提としている。

- 初等関数の微分積分・数列や級数の極限・ベクトルや行列の計算ができる。
- パソコンのタイピングができ、オフィス・メール・ブラウザ・テキストエディタ等を利用できる。
- ファイルパスや拡張子の意味を理解しており、CUIを苦にしない。
- 中学生レベルの英語の読み書きができる。
- インターネット環境がある。

このうち、インターネット環境以外については多少の学習で身につけられるものであるので、本書を読まれる際に迷った場合があれば適当な書籍等で復習されたい。理系の方でなくとも、上記を知っていれば、あるいは並行して学習すれば本書を糧と出来るだろう。

筆者がC言語を教える際には、文法にとどまらず、プログラミング全体のセンスが身につくように努めている。それを本書でも活かすため、アルゴリズムなどの回を取り入れた。逆に演習は全て別に任せ(この理論編と合わせて、演習を中心として学ぶ「実習編」も編みたいと考えているが)、本書においては全15回でプログラミング全体に対する基本が身につくように徹底したつもりである。なお、自習または補習の目的として第0講を配し、学習の前に知っておくと良いと考えられる知識と、学習に必要な環境を構築する方法を述べるようにした。

プログラミングは一つのツールであるが、同時に一つの趣味をも提供しうるものである。この事例はまさしく筆者であり、一方で物理の理解のためのシミュレーションにプログラミングを用い、他方で競技プログラミングに参加している。これはプログラミングの工芸的側面を示している。つまり、役立つものを作るという工学的観点と、美しいものを作るという芸術的観点である。創作活動たるプログラミングは、パズルをとくものから自由気ままな創作に至るまで幅広く、それは単に美しいだけではなくて実利的側面を兼ね備え得る。本書はこの見地にたって、ツールとして実用に耐えうるうものでありながらそれを趣味として楽しめるように解説を記すよう心がけた。実用としてプログラミングを学ぶ必要もあろうが、その創作的楽しみを十分味わってもらえるように切に願っている。そして、本書を読んだ後に、実用的にプログラミングを使いつつ競技プログラミングを楽しむ方がいらっしゃったなら幸甚である。

なお、先に述べたとおり、本書は思い切って演習を省き、演習については姉妹書「実習編」にまとめた。同時利用を前提としての執筆であり、先に記した筆者のスタンスはこれら2冊によって為される。共にご活用されることを切に願う。

また、本書のタイトルについてであるが、これは私が指導に当たった方からの「この講座は理論を損なうことなく、実践も丁寧にやっている講座だ」という声を基にしたものである。このスタンスを本書でも貫き、巷間の専門書とはまた一味違ったものになっていれば、楽しく読んでいただけるのではないか。その思いから本書のタイトルを「理論と実習の両面から学ぶ」とした。

最後になったが、前テキストの共同執筆者の方々、筆者の拙い解説を聴きに来てくれた後輩たち、本テキストを利用してくださる方々に感謝の意を表して、本書の序としたい。

中島みゆき「誕生」のかかる部屋にて
達哉ん

公私ともさまざまにあり、姉妹書「実習編」の執筆作業もあまり進んでいない中ではあるが、公開より1年余を経た2013年9月、誤植の修正をはじめ、内容のまずい部分(生硬な部分・内容が不十分である部分)の修正、一部内容の追記を行った。

幸い、自身の状況が落ち着いてきたので、この修正版公開を皮切りに実習編の執筆作業に本腰を入れていきたいと考えている。

中島みゆき「月はそこにいる」を口ずさみながら
達哉ん

本書の利用方法

本書は全 15 講の解説と、付録からなっている。また、前提知識をざっと学ぶ第 0 講も用意している。

第 1 講から第 12 講までは C 言語の文法を学ぶ。通読し、姉妹書「実習編」を用いて演習をこなすことで、C 言語およびプログラミングの基礎が身につくように構成したつもりである。

第 13 講以降は、アルゴリズムなどについての基礎を学ぶ。これを通じて文法の復習をすると共に、実装できる幅を広げることができるように構成した。

基本的には解説を通読されることをお勧めしたい。その時、各講の学習の後に姉妹書「実習編」で実際に手を動かすことにより復習されることをすすめる。十分な演習の重要性は今ここで再強調するまでもないことだろう。継続的に、復習も含めた演習を行うことでプログラミング力の底上げができることは疑いない。十分な活用を願っている。

通読の際、ソースがあればそれは必ず手を動かして打ち込み、実行することをすすめる。実際の動作なくしてプログラミングを習得することはできない。また、動作確認そのものがプログラミングの理論の理解にもつながることだろう。それ以外は、普通に読んでもらって構わない。まとめながらも良いし、線を引ながらも良い。

筆者がプログラミング初学者にすすめている学び方を紹介しておく。

- 基本的な文法事項、関数、アルゴリズムなどはノートにごく簡単にまとめておき、辞書のように使えるようにする。
- ソースに出会ったら、必ずそれを手ずから打ち込んで、動作を確認する。
- プログラミング中は紅茶などを用意し、ゆっくり落ち着いて、時間と心に余裕をもって考えること。
- パソコンばかりでなく、紙とペンも用意し、内容を整理しながらプログラミングすること。
- 多くの問題に挑戦し、自由に作品を作り、出来ればプログラミングの出来る人に批評を求めること。

これらを守ってプログラミングの学習をされることで、飛躍的に効果が出ることだろう。これからの学習が順調に進むことを祈っている。

目次

第0講	プログラミングの前に	1
0.1	記数法とコンピュータ	1
0.1.1	記数法の一般論	1
0.1.2	記数法の変換	2
0.1.3	ビットとバイト	4
0.2	パソコンの仕組み	5
0.2.1	パソコンを構成する装置	5
0.2.2	主記憶装置	6
0.2.3	ソフトウェアの分類	8
0.3	Cプログラミング環境構築	9
0.3.1	Windows での環境構築	9
0.3.2	Linux について	10
0.3.3	Windows ホスト環境への Linux の導入	10
0.3.4	Linux の端末操作	11
0.3.5	Linux での環境構築	14
第1講	プログラミングとは何か	17
1.1	プログラミングとは何か	17
1.2	プログラミング言語と C 言語	18
1.2.1	高級言語・低級言語と処理の流れ	18
1.2.2	C 言語の特徴	19
1.2.3	C 言語の歴史	19
1.3	プログラム作成の一連の流れ	21
1.3.1	コーディングまでの体験	21
1.3.2	コンパイル・実行	22
1.3.3	「はじめてのプログラム」の解説	23
1.4	標準出力への出力	26
第2講	変数の概念	29
2.1	変数とデータ型	29
2.1.1	変数とは	29
2.1.2	識別子の命名	30
2.1.3	整数型の仕組み	31
2.1.4	浮動小数点数型の仕組み	33
2.1.5	文字型の仕組み	34
2.2	変数の入出力と演算	35

2.2.1	変数の出力	35
2.2.2	変数の入力と簡単な計算	38
第 3 講	変数の活用	44
3.1	計算の誤差	44
3.1.1	丸め誤差	45
3.1.2	桁落ち	45
3.1.3	情報落ち	46
3.1.4	計算機イプシロン	46
3.1.5	打ち切り誤差	46
3.2	ビット演算	46
3.3	数学関数と複素数演算	49
3.3.1	数学関数の利用	49
3.3.2	複素数演算と型総称数学関数	51
第 4 講	分岐構造	56
4.1	分岐構造とは	56
4.2	if 文と else 文	56
4.2.1	if 文	56
4.2.2	else 文	58
4.3	論理演算子	60
4.4	switch～case 文	62
4.5	条件演算子	65
4.6	_Bool 型の利用	66
第 5 講	反復構造	69
5.1	反復構造とは	69
5.2	while 文	69
5.3	do-while 文	71
5.4	for 文	73
5.5	反復制御文と goto 文	75
第 6 講	自作関数とプリプロセッサ	80
6.1	関数の呼び出しと利用	80
6.1.1	関数化の意義	80
6.1.2	関数の要素	81
6.2	自作関数の利用	82
6.2.1	自作関数の作成方法	82
6.2.2	関数プロトタイプ宣言	85
6.2.3	型の別名定義	85
6.3	スコープと寿命	86
6.3.1	オブジェクトのスコープ (有効範囲)	86
6.3.2	変数の寿命	88
6.4	前処理命令と分割コンパイル	89

6.4.1	マクロ	90
6.4.2	ヘッダファイルの実態とインクルード	94
6.4.3	分割コンパイルと複数ファイルでのスコープ	96
第7講	様々な関数	100
7.1	標準ライブラリの利用	100
7.1.1	時間計測	100
7.1.2	文字の分類に関する関数	102
7.1.3	プログラムの終了	103
7.1.4	型に関する標準ライブラリ	105
7.1.5	代替綴りとトライグラフ	108
7.1.6	man コマンド	109
7.2	再帰関数	110
7.2.1	再帰関数のメモリ上での動作	111
7.2.2	再帰関数の注意点	112
7.3	可変引数関数	113
7.3.1	stdarg.h の利用	113
7.3.2	ターミネータを用いる方法	114
7.3.3	可変引数より前の引数で個数を示す方法	115
7.3.4	可変引数を一括処理する関数	116
7.4	インライン関数	116
第8講	派生型 (1) 静的配列と文字列	119
8.1	基本型と派生型	119
8.1.1	派生操作の組み合わせ	119
8.1.2	基本型でも派生型でもない型	120
8.1.3	型についてのまとめ	120
8.2	静的配列	120
8.2.1	配列に関する用語	121
8.2.2	1次元配列	121
8.2.3	多次元配列	125
8.2.4	一次元静的配列を引数に取る関数	127
8.3	文字列	131
8.3.1	文字列とその取り扱い	131
8.3.2	文字列操作関数	135
第9講	派生型 (2) 構造体・共用体他	138
9.1	構造体	138
9.1.1	構造体の概念	138
9.1.2	構造体の利用	138
9.1.3	ビットフィールド	145
9.2	共用体	147
9.2.1	共用体の概念	147

9.2.2	共用体の利用	147
9.3	列挙型	149
9.3.1	列挙型の概念	149
9.3.2	列挙型の利用	149
第 10 講	派生型 (3) ポインタの概念	154
10.1	メモリ領域再論	154
10.2	アドレスとポインタ	155
10.2.1	アドレスの取得と意義	155
10.2.2	アドレスのための変数=ポインタ	160
10.3	ポインタ演算	165
10.4	ポインタを用いる関数	168
10.4.1	ポインタを用いる自作関数	168
10.4.2	ポインタを用いる標準ライブラリ関数	170
第 11 講	派生型 (4) ポインタの活用	175
11.1	各種派生型へのポインタ	175
11.1.1	配列とポインタの関係～ポインタは配列エイリアスか～	175
11.1.2	文字列リテラルについて	180
11.1.3	関数へのポインタ	182
11.1.4	ポインタへのポインタ	184
11.2	メモリの動的確保	187
11.2.1	動的配列の概念と利用法	187
11.2.2	フレキシブル配列メンバ	190
11.3	関数間で配列をやり取りするには	191
11.3.1	一次元配列に帰着させる方法	191
11.3.2	要素数を固定して渡す方法	192
11.3.3	配列エイリアスのポインタを用いる	192
第 12 講	ストリームと入出力	194
12.1	ファイルとストリーム	194
12.1.1	ファイルとは	194
12.1.2	ストリームとは	195
12.1.3	リダイレクトの意味	195
12.2	ストリームの取り扱い	196
12.2.1	ファイルの開閉と入出力	196
12.2.2	ファイル・ストリームを取り扱う関数	201
12.3	バイナリファイルの入出力	202
12.4	コマンドライン引数	204
12.4.1	main 関数の引数	204
12.4.2	第 3 のコマンドライン引数	206

第 13 講 データ構造の基礎	208
13.1 データ構造とは	208
13.2 基本的なデータ構造	208
13.2.1 スタック	208
13.2.2 キュー	209
13.3 リスト	211
13.3.1 片方向リスト	211
13.3.2 双方向リスト	212
13.3.3 ループのチェック	212
13.4 木 (Tree) 構造	213
13.4.1 二分木	214
13.4.2 木の探索	214
13.4.3 二分ヒープ	215
13.4.4 二分探索木	217
13.5 グラフ	218
13.5.1 グラフの基礎概念	219
13.5.2 グラフの実装と連結性チェック	220
13.5.3 最短路問題	221
13.5.4 閉路問題	225
第 14 講 ソートとサーチ	228
14.1 ソート総論	228
14.1.1 ソートとは何か	228
14.1.2 ソートの解析～バブルソートを例に～	228
14.2 基本的なソート	230
14.2.1 挿入ソート	230
14.2.2 選択ソート	231
14.2.3 シェーカーソート	232
14.2.4 ノームソート	232
14.2.5 シェルソート	233
14.3 高速なソート	234
14.3.1 マージとマージソート	234
14.3.2 クイックソート	237
14.3.3 ヒープソート	239
14.3.4 コムソート	239
14.3.5 ストランドソート	240
14.3.6 イン트로ソート	241
14.3.7 ティムソート	241
14.3.8 バケットソート	242
14.4 サーチとその手法	243
14.4.1 リニアサーチ	243
14.4.2 バイナリサーチ	243

第 15 講 数値計算の基礎	246
15.1 1 元方程式を解く	246
15.1.1 逐次探索法	247
15.1.2 二分法	247
15.1.3 割線法	248
15.1.4 はさみうち法	250
15.1.5 Newton 法	250
15.2 数値積分法	254
15.2.1 台形則	254
15.2.2 中点則	255
15.2.3 シンプソン則	255
15.2.4 モンテカルロ積分	256
15.3 連立方程式と行列	257
15.3.1 Gauss 法	257
15.3.2 Gauss-Jordan 法	260
15.3.3 行列式や逆行列への応用	260
付録 A 簡易リファレンス	263
A.1 assert.h(プログラム診断)	263
A.2 ctype.h(文字の分類)	263
A.3 errno.h(エラー)	264
A.4 float.h(浮動小数点数型属性検査)	264
A.5 limits.h(整数型属性検査)	266
A.6 locale.h(地域情報管理)	267
A.7 math.h(数学関数)	267
A.8 setjmp.h(非局所分岐)	270
A.9 signal.h(シグナル処理)	271
A.10 stdarg.h(可変引数)	271
A.11 stddef.h(共通定義)	271
A.12 stdio.h(標準入出力)	272
A.13 stdlib.h(ユーティリティ)	275
A.14 string.h(文字列操作)	277
A.15 time.h(時間)	279
A.16 iso646.h(代替綴・C95)	279
A.17 wchar.h(ワイド文字・C95)	280
A.18 wctype.h(ワイド文字変換・C95)	282
A.19 complex.h(複素数演算・C99)	283
A.20 fenv.h(浮動小数点数環境・C99)	284
A.21 inttypes.h(整数型書式・C99)	285
A.22 stdbool.h(論理・C99)	285
A.23 stdint.h(整数型管理・C99)	285
A.24 tgmath.h(型総称数学関数・C99)	287

付録 B	一部解説の付記	288
B.1	ASCII 文字コード一覧	288
B.2	C 言語の予約語一覧	289
B.3	演算子の評価順序	291
B.4	マクロの詳細な文法	291
B.4.1	文字列化演算子#	291
B.4.2	字句連結演算子##	292
B.4.3	可変引数関数マクロ	292
B.5	volatile 修飾子	293
B.6	ローケル	293
B.7	マルチバイト文字とワイド文字	293
B.7.1	マルチバイト文字	294
B.7.2	ワイド文字	294
付録 C	C に関連した Web サービス	295

第0講 プログラミングの前に

第0講では、プログラミングに関係するパソコンの基礎知識を説明する。これから車の運転を習おうという人がブレーキやアクセルとは何だと訊くことはまずないだろう。だが、プログラミングの場合、ブレーキやアクセルに相当する部分を知らずに学ぼうということが少なくない。そこで、ここではそれらの必要最小限の説明を付しておく。もちろん、知っているならば読み飛ばしてもらって良い。

ここでは、情報数学の一環として、まず記数法について説明する。次いで、パソコンの仕組みについて触れ、最後に学習環境の構築について記しておく。

0.1 記数法とコンピュータ

我々が物を数える際には手の10本の指を用いる。ここから物を数える位取りとして、指がいっぱいになったら位を1増やすということで**10進法** (decimal system) が生まれた。10進法とは、10(十、ten) 毎に位がひとつ上がるような数え方であり、我々が普段使っている数の表記法である。10進法により表された数のことを**10進数** (decimal digit) と呼ぶ。

一方、コンピュータはON/OFFで表すという事から、2つの状態(=2つの指)を持つといえる。このことから、コンピュータは2毎に位がひとつ上がり、0,1,10,11,... と数える**2進法** (binary system) を中心に使っている。

これらのような、数の表記にあたりどのようにして位をとるかの方法を**位取り記数法** (positional notation) あるいは単に**記数法**と呼ぶ。ここでは、記数法の一般論と、コンピュータでよく使われる記数法の利用について学ぶ。

0.1.1 記数法の一般論

先までの例と同様に、指の本数から考えよう。指が $n(\geq 2)$ 本しかないと仮定する。このときは、 n 本がいっぱいになったら次の位に移る。たとえば、指が3本しかなければ、3になったら次の位に移るようにして数えることができる。

一般の自然数 n に対して、 n 進法とは、 n を位取りにする方法である。つまり、数えていって、 n に達する毎に上の位に移るという事である。

例えば、123という数字を考えよう。これは、123個の1であるが、同時に12個の10と3個の1である。更に分ければ、1個の100と2個の10と3個の1と分かれる。つまり、 10^k の束がいくつあるかを数え、それが10個に達したら 10^{k+1} を1増やす。これによって、各位は 10^{k+1} の束にならない余りとなる¹。

¹これは後に学ぶ記数法の変換と桁わけの意味を理解するのに重要である。

上記に従って位取りをしてみよう。10進法では、小数点の直上を 10^0 とし、その上の桁を $10^1, 10^2, \dots$ 、小数点以下を $10^{-1}, 10^{-2}, \dots$ としている。この10を n に変えたのが n 進法である。たとえば、2進法における11111は $2^4 \times 1 + \dots + 2^0 \times 1 = 31$ である。

n 進法で使われる数字は、 n 種類ある。2進法なら0と1だし、10進法なら0, 1, 2, 3, 4, 5, 6, 7, 8, 9であり、16進法の場合は0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, fである²。

位取り記数法は筆算と相性がよく、四則演算は何れも10進法の場合同様に筆算を用いて計算することができる。これにより、記数法を変更しても、計算は苦勞なくできるはずである。(但し、後に学ぶ方法を用いて、10進法に一度変換してから行なっても問題はない)。

何進法であるかを明確にするときには、数字の右下に $33_{(n)}$ のように記す。たとえば、 $101.1_{(2)} = 5.5_{(10)}$ である。

0.1.2 記数法の変換

ここでは、記数法を変換する方法について考えよう。まず、一般論として10進法を介して変換する方法を述べ、次いでコンピュータでよく用いられる2進/8進/16進の直接変換法を見てみる。

【10進法との相互変換】

n 進法から10進法への変換はどのように行うだろうか。これは n 進法の仕組みを考えてみればわかる。

n^k の位が a_k であるような数字 A の10進法での値は次のように求められる。

n 進法から10進法への変換

n^k の位が a_k であるような数字 $A_{(n)}$ は、10進法において

$$A_{(10)} = \sum_k a_k \times n^k$$

と計算できる。

これは、 n 進法の定義を考えれば直接出てくる方法である。

逆に、10進法から n 進法に変える場合は、どうすればよいだろうか。

1つには、単に「取り尽くす」方法がある。つまり、 n^k のうち、元の数を超えない最大のものを見つけ、その数で元の数割った商を n^k の位にし、余りについて n^{k-1} で割り…と繰り返すのである。特に、小数点がある場合はこの方法を利用すると楽である。

だが、この計算はやや面倒に感じることもあるだろう。そこで、主に整数向けであるが、先に書いた位取りの考えを用いる方法がある。

n^k の位が a_k であるという事は、 n^k で割った後に、小数点以下を切り捨て、 n に対する剰余を取ると a_k になる、という事である。例えば、10進法で100の位の数字を出したい

²アルファベットは大文字を用いることもある。C言語ではどちらを用いても構わない。

とすれば、100 で割って 10 による余りを取れば良い。これと同じ性質が一般の n 進法にも成立する。この性質を利用し、次のような手順を踏んで記数法を変換できる。

10 進法から n 進法への変換

1. 変換したい 10 進数を n で割り、その余りをメモする。
2. これを、商が 0 になるまで繰り返す。
3. メモした余りを逆から順に並べると n 進数での表記になる。

この方法は、 n 進有限小数で表せる場合、 n の整数乗をかけてから実行することで有限小数の場合にも適用できる。

一例として、 $12.25_{(10)}$ を 8 進法に変換してみよう。

まず、これを 8 倍してやれば、98 になる。8 倍するというのは 8 進法で 1 桁上にあげたことを意味するので、最後に 1 桁下げることで辻褄を合わせる。

次いで、98 を 8 で割っていく。順に書くと

$$98/8 = 12 \dots 2$$

$$12/8 = 1 \dots 4$$

$$1/8 = 0 \dots 1$$

となるので、これを下側から順に読んで、142 となる。1 桁上げていることを考慮すれば、 $12.25_{(10)} = 14.2_{(8)}$ である。逆に変換してみれば正しいことが明らかだろう。

いくつかの数を変換してみればわかるが、10 進整数は他の記数法でも整数である。一方、ある記数法で有限小数であるからと言って他の記数法で有限小数であるとは限らない。例えば、 $1/3$ は、10 進法では無限小数だが、3 進法であれば $0.1_{(3)}$ とシンプルに表せる。

【2 進/8 進/16 進の直接変換】

コンピュータの世界では、内部的な処理の関連で 2 進法をよく使うが、これを見やすくするために 8 進法や 16 進法が使われる。また、添字形式はコンピュータで表現しづらいため、8 進法は 0 をつけて 025 などと、16 進法は 0x ないし 0X をつけて、0x3a や 0X5B などと記す (x の大小と英字の大小が対応している)。以下では、この記法を用いて記す。

2 進法を見やすくするために 8/16 進法を使うと書いたが、これは、2 進法を 3/4 桁毎にまとめたものが 8/16 進法だからである。

具体例を見てみよう。2 進法で 110010011111 という数字を考える。まず、これを 8 進法に変換してみよう。

3 桁ごとに分けて書けば 110 010 011 111 となる。ここで、一番下の 3 桁はそのまま 8 で割った余りである。次の 3 桁は、8 で割り切れるが 64 では割り切れない部分である。同様に考えていけば、2 進法 3 桁ごとに 8 進法 1 桁と同じ値を表していることがわかる。そこで、これを 3 桁ごとに 10 進法に直すのと同じ要領で計算すれば 06237 と変換できる。

これを今度は4桁ごとに分けてやると、1100 1001 1111 となる。同じ要領で16進法に変換することができ、0xc9f となる。

このように、2進法での表記は8/16進法に簡単に変換できる。逆に、8/16進法を用いて書いたものを2進法に直したいならば、各位を3/4桁の2進数に変換して並べれば良い。

例えば、0xbea2 などという数字があった時、10進法を介すると大変だが、直接変換すれば、b は 1011, e は 1110, a は 1010, 2 は 0010 なので、これを並べて 1011111010100010 と変換することができる。

一般に、 n 進法と n^k 進法の間では、このように k 桁毎の直接変換を行うことができる。実際に使うのは、2/8/16 進法の間での直接変換ぐらいだろうが、知っておくと便利である。

0.1.3 ビットとバイト

コンピュータはどのようなデータも2進数で表している。この2進数1桁(あるいはその情報量)を1ビット(bit, Binary digiT の略)という。このことから、例えば16bit の情報は、全部で $2^{16} = 65536$ 通りあることがわかる。また、モデム回線の56kbps や光ファイバーの100Mbps といった回線速度の単位 bps は bit per second の意味で、それぞれ 56×1000 ビット、 $100 \times 1000 \times 1000$ ビットの情報を1秒間にやり取りできることを示している³。

だが、データの単位としてビットを使っただけだと大きくなるし、まとまりも悪い。また、13ビットの情報などは、先に書いたような8/16進表記もしづらい。そこで、16進法を使って簡単に表せるよう16進2桁=8bit の情報を1オクテット(octet)として扱うことになっている⁴。そして、データの大きさは通常、オクテットを用いて表す。だが、実際に聞くことがあるのはオクテットではなくてバイトであろう。

バイト(byte)は、一般的にはオクテットと同じ単位であり、1byte=1octet=8bit である⁵。但し、環境によっては1byteが1octetではない場合もある。この為、8bitであることを明確に示すための1octetという言い方が生まれたのである。なお、本書では特に断りない場合、1byte=1octet とする。

1byteは8bitであるので、1byteによって表せる情報は $2^8 = 256$ 通りあることがわかる。これだけあれば世界共通で使われる文字などを表すことができるため、これらの文字は1バイト文字と呼ばれている。

以上のように、Byte というのは2進法(=bit)から決められた単位であるので、その大きさを計算するためには、bit に直してやれば良い。

³k(キロ)やM(メガ)は通常 10^3 や 10^6 である。だが、コンピュータの世界では 2^{10} が1024で 10^3 に近いので、これを用いて表すこともある。コンピュータの世界でもこれらは混同されており、500GB と書かれたHDDを買ってきて接続すると、コンピュータ側では536GBと表示されるなどの違いが出ることもある。なお、混同をなくすために 2^{10} を使っている場合には接頭辞の後ろにiをつけ、kiB(キビバイト)やMiB(メビバイト)として表すこともある。

⁴鉛筆12本を1ダースとして数えるのと同じような感覚である。

⁵一般にbはbitを、BはByteを表す。従って、56kbpsと7kBpsは同じ速度である。

0.2 パソコンの仕組み

パソコンの仕組みについて、簡単に紹介しておく。

0.2.1 パソコンを構成する装置

パソコンは大きくって次のような装置から構成されている。

コンピュータを構成する装置

- **入力装置 (input device)**: コンピュータ (や実行中のプログラム) にデータや情報、指示などを与えるための装置。キーボードやマウス、スキャナ・マイク等。
- **出力装置 (output device)**: コンピュータの計算結果などを何らかの形で出力する装置。ディスプレイ・スピーカー等。
- **中央演算処理装置 (CPU, Central Processing Unit)**: 厳密には**論理演算装置 (arithmetic logic unit)** と **処理装置 (control unit)** に分かれるが、実際にはこれらをまとめて CPU として取り付けているのが普通である。コンピュータの演算や処理を行う、頭脳と言える装置である。
- **主記憶装置 (main memory/main storage)**: 後に示す**補助記憶装置**とあわせて**記憶装置 (memory/storage)** と呼ばれる。揮発性 (通電されなくなると記憶内容が消えてしまう) の記憶装置で、CPU から高速にアクセスすることができる部分である。これは、CPU が作業を行うための机のような役割を果たす装置である。これは C 言語プログラミングで重要であるので、後でより詳しくにする。
- **補助記憶装置 (external memory/external storage)**: 先に記した主記憶装置とあわせて、記憶装置と呼ばれる。不揮発性 (通電されなくなっても記憶内容が残っている) であり、CPU からのアクセス速度が主記憶装置より遅いが、安価で大容量である。HDD や SSD がこれに相当し、データ収納の役割を果たす。なお、補助記憶装置の一部を主記憶装置のようにみなし、主記憶装置で容量不足が起こった時などに用いることができるようにした技術として**仮想メモリ**がある^a。

^aLinux の場合、HDD 上に swap 領域と呼ばれる領域を作成し、その部分を仮想メモリとして用いる。Windows でも、明確に swap 領域としてパーティションを作ることはないが、この仕組みを使っている。USB フラッシュメモリによる ReadyBoost もこの技術の一例である。

これらの、入力装置・出力装置・論理演算装置・制御装置・記憶装置をまとめて**コンピュータの5大装置**などという事もある。

これらの装置がどのようにしてデータを処理するかを見ていこう。

まず、CPU が入力装置に対して入力を受け付けるように制御する信号を出す。これによって入力装置は入力を受け付けるようになる。その後、データの入力が行われ、主記憶装置に保存される。これに対して CPU が適切に処理を行い、必要に応じて外部記憶装置に格納したり出力装置に渡して出力したりする。CPU は処理と共に、データの格納や出力、あるいは外部記憶装置からの読み出しなどの制御も行なっている。

以上に述べた装置の動作を図示したものが図 0.1 である。

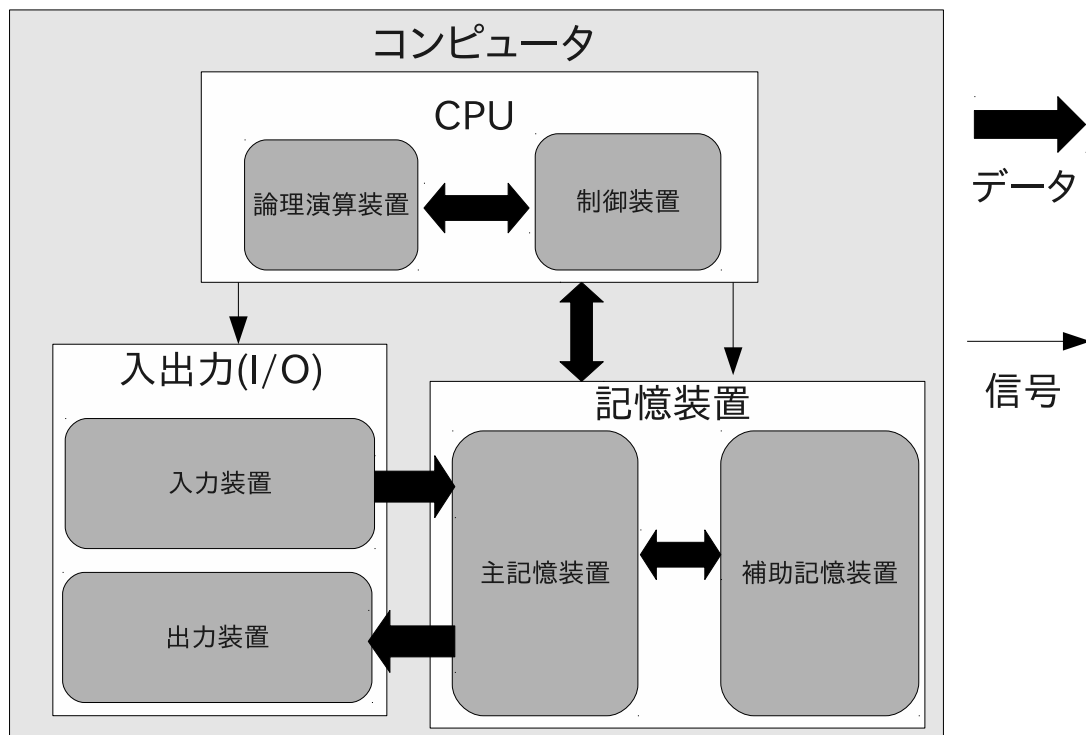


図 0.1: コンピュータの装置間の動作

0.2.2 主記憶装置

今後学んでいく C 言語では、メモリをはじめとした主記憶装置について意識しないといけない場面が数多く出てくる。この為、主記憶装置については他の装置以上に深い理解が要求される。C 言語そのものを学ぶ前に、主記憶装置への理解を深めておこう。

【主記憶装置の種類】

主記憶装置は、CPU からの距離に応じて大別される。

CPU はその最も近いところにレジスタ (register) と呼ばれる主記憶装置を持っており、これが一番高速に用いられる主記憶装置である。次いで、キャッシュメモリ (cache memory) と呼ばれる主記憶装置がある⁶。そして、我々が一般にメモリ (memory) と呼んでいる主記憶装置 (実際に区別するためにはメインメモリと呼ぶことが多い) があり、この 3 つによって主記憶装置が成り立っている。

このように 3 つに分けているのは、CPU から近ければ近いほど高速であるが、その分容量が減るためである。例えば、レジスタはメモリに比べるとずっと小さい。従って、必要に応じて、「どの主記憶装置を用いるべきか」も考慮しなければならない。

⁶さらに、キャッシュメモリ自体も近い順に 1 次、2 次、3 次と大別される

【RAM】

メインメモリは**RAM**(Random Access Memory)とも呼ばれる⁷。これは、前から順に「どこに書きこめばよいか」を探さなければならないシーケンシャルアクセス (sequential access) を用いると、メモリへのアクセス速度を損ねてしまうからである⁸。どこか空いているところを電氣的にすぐに判定し、そこに書きこむようにするランダムアクセスを用いて速度を損ねることなく利用できるようにしているのである。

また、ランダムアクセスすると、メモリ全体の使われる頻度が均一になり、長持ちしやすいという利点もある。

メインメモリには、ここに述べたようにランダムアクセス方式を用いる。このため、RAMと呼んだ場合はメインメモリのことを指すのが一般的である。

【メモリの領域】

C言語を用いてプログラミングを行う場合、メモリの領域は次の4つに大別される。

Cにおけるメモリ領域

- **プログラム領域**: プログラムを実行するためのプログラムコードが置かれる領域。プログラムを実行したい場合、そのプログラムはメモリ上にロードされ、順次実行されることになる。このロードに使われる領域の事。
- **静的領域**: 外部変数や静的変数といった、プログラムの実行中に変化しない (実行中ずっと寿命が保たれている) 変数を格納する領域。寿命については後に学ぶ。
- **スタック領域**: 一般の変数、関数の引数や返却値、長い計算式の一時変数などが置かれる領域。
- **ヒープ領域**: プログラム中で動的にメモリが確保される場合に使われる領域。

自作のプログラムを実行したとしよう。実行されたプログラムは、まずプログラム領域にロードされる。この時、静的領域とスタック領域が割り当てられる。これらの大きさはプログラム実行中不変であるが、静的領域は領域内の使用量が実行中に変化しないのに対し、スタック領域では領域内の使用量が実行中に変化する。このことから、スタック領域の割り当ては「スタック領域をどれだけ使っていいか」という、限度の割り当てである。

以上でプログラムがロードされたら、このあとはプログラム領域の命令を順次読み取ってCPUが処理を実行する。この際、プログラム側が動的にメモリを確保したいと要求すればヒープ領域を用いることになる。

現状ではまだピンと来ない説明かもしれないが、本書を学んでいくうちに、必要に応じてここに戻ってきていただければ、理解できるだろう。

⁷似た言葉に**ROM**があるが、これはRead-Only Memoryの略で、一度書き込んだら書き換えられないような領域を指す。

⁸なお、HDDは比較的シーケンシャルアクセスに近いアクセスで、目的のファイルを読み出すのに時間がかかる。一方、SSDはランダムアクセスの性能が高く、HDDより高速なアクセスを実現している。

【メモリとアドレス】

メモリはランダムアクセスであるが、その場所がわからないと困る。そこで、メモリにはアドレス (address) という番号がふられており、これによってメモリ上に配置されているデータの場所が示される。

プログラミング言語を機械に近づけていくと、このアドレスを用いたメモリいじりが主体であることがわかる。逆に、近年生まれた「人間に近い」言語はアドレスいじりを直接目に見える形では行わない場合が多い。

ひとまずここでは、メモリの番地を表すアドレスという指標があるという事を知って置いていただきたい。

【スタック領域とヒープ領域の特徴】

先に説明したスタック領域とヒープ領域の特徴について、少し記しておく。

まず、ヒープ領域はある種の共通領域で、割り当てなどがないのに対し、スタック領域には割り当てがあるという点が違う。メモリが空いている時、ヒープ領域は空いている限り使うことができるが、スタック領域はいくらメモリが空いていても割り当てられた量までしか使うことができない。この為、スタック領域を使い過ぎると、スタックオーバーフロー (stack overflow)、日本語に訳すと「スタック溢れ」が起こり、プログラムが強制終了されてしまう。なお、スタックオーバーフローは、メモリ上のアクセス禁止領域にアクセスした事を意味するセグメンテーション違反 (segmentation fault) として検出される場合もある。

一方で、ヒープ領域はそう簡単には一杯にならないが、プログラマがその利用を指示しなければならない他、後始末 (使用後にメモリを解放する) など必要な処理が自動化されていない (C 言語の場合) などの欠点がある。また、ヒープ領域がいくら多いといえど有限であるので、一杯になってしまう場合もある。悪いことに、ヒープ領域が一杯になっても検知されないことがあり、パソコンの動作が不安定になったり、問題が起こったとしてパソコン自体が強制シャットダウンされてしまったりする場合もある。

また、スタック領域はメモリのアドレスの大きい方から小さい方に向かって、ヒープ領域はメモリのアドレスの小さい方から大きい方に向かって使われるという特徴がある事を付記しておく。

0.2.3 ソフトウェアの分類

以下、ソフトウェアの分類について簡単に説明しておく。

【基本ソフトウェアと応用ソフトウェア】

ソフトウェアは大きく分けて基本ソフトウェアと応用ソフトウェアに分かれる。

基本ソフトウェアはオペレーティングシステム (Operating System, OS) と呼ばれ、基

本的なシステムを司るソフトウェアである。つまり、キーボード入力や画面出力といった入出力機能やディスクやメモリの管理など、多くのアプリケーションソフトから共通して利用される基本的な機能を提供し、コンピュータシステム全体を管理する役目を持つ。Windowsをはじめ、Mac, Google Chrome OS や後述する Linux、スマートフォン向けの Google Android, Firefox OS などが OS の具体例である。

応用ソフトウェアは文書の作成、数値計算など、ある特定の目的のために設計されたソフトウェアのことを指す。これらのソフトは、OS がまとめている基本的な機能を応用する形で、ユーザに必要な機能を提供する。

本書で学ぶ C 言語は基本ソフトウェアを作ることにも可能な言語である。だが、そのレベルまで達するのはまだ先のことである。ひとまず本書では、Windows/Linux 上で動かす応用ソフトウェアを作成することにする。

【ユーザインターフェイスによる分類】

ユーザインターフェイス (User InterFace, UI) とはユーザに対する情報の表示様式やデータ入力方式を規定する、コンピュータシステムの「操作上の見た目」「操作するための装置の配置」「操作する際の感覚」のようなものを意味する言葉である。これは、文字をベースとした CUI(Character-based User Interface) とマウスなどで視覚的に扱うことができる GUI(Graphical User Interface) に大別される。これは操作の見た目であるため、ソフトウェアの分類でもある。つまり、コマンドを打って操作を実行するような CUI ソフトと、マウスでクリックするなどして操作を行う GUI ソフトがある。

本書では初心者向けという観点から、作成の簡単な CUI のみを対象として説明する。GUI は CUI よりも複雑になりがちなためである。

0.3 C プログラミング環境構築

ここでは、本書で学ぶ C 言語プログラミングの学習環境を構築するための方法を説明する。なお、本書では Linux Mint+emacs+gcc という環境を元に記しているが、必ずしもこれに従う必要はない。

0.3.1 Windows での環境構築

Windows については、「苦しんで覚える C 言語」というサイトの管理人である MMgames 氏が学習用 C 言語開発環境というものを公開しており⁹、これをインストールするだけで学習環境が手に入る。Windows での本書の学習については、この環境を利用すれば良いだろう。

あるいは、Web 上のサービスにも "ideone" 等 C 言語の開発環境を提供してくれるものがあるので、こちらを利用しても良いだろう (詳しくは付録 C を参照)。

⁹http://9cguide.appspot.com/p_9cside.html

0.3.2 Linux について

先に書いたとおり、本書では Linux Mint という OS を元にして記している。これは Linux という OS の一種である。

Linux は主として開発者に使われることが多い OS で、オープンソースの OS である。Linux は UNIX という OS を真似し、これに様々な拡張機能を付加したものであり、これらは共に C 言語で書かれている。C 言語を学んだ後、これらのソースを読めば文法的に読めないことはないはずである (処理などは難しいが)。なお、Linux は様々な OS のベースとなっており、モバイル向けの OS などの中にも Linux を元としたものが見られる。

Linux はその用途に応じて様々な配布形式 (ディストリビューション) があり、用途に応じて選択することができる。筆者の環境は Linux Mint というディストリビューションのバージョン 13 であり、デスクトップ向けの Linux ディストリビューションである。ディストリビューションは他にも多くの種類があり¹⁰、DistroWatch¹¹や DistroFreak¹²といった、ディストリビューション情報を専門としたサイトも見られる。ディストリビューションの選択については自分で情報を集めてもらってもよいが、初心者向けにすすめるならば Linux Mint, Ubuntu, Vine 辺りが使いやすく情報も多いだろう。

なお、本書は原則 Linux のディストリビューションとは独立して解説を行うが、この講については Linux Mint 13 を元に記す (Linux Mint 13 でなければいけないというわけではない)。

0.3.3 Windows ホスト環境への Linux の導入

Windows が元々入っている環境に Linux を入れるには、次のような方法がある。

Windows ホスト環境に Linux を導入する方法

- 仮想マシンを用いる。VMware や Virtual Box が有名である。
- 簡易インストーラーが付いている Linux (Ubuntu (Wubi) や Linux Mint (Mint4win) など) を用いてインストールする。
- HDD の領域 (パーティション) を分けて、分けた部分の領域に Linux をインストールする。

この内、仮想マシン以外の方法は起動時にどの OS かを選ぶマルチブート (multi boot) になり、両方の OS を同時に使うことはできない。一方、仮想マシンを用いれば2つの OS を同時に使うことができるようになるが、マシンパワーがなければ動作が重くなる。

Linux のインストール方法はディストリビューションやバージョンによって様々に変わるため、ここでは記さない。ディストリビューションとインストール方法を決めたら「Linux

¹⁰例えば、筆者が試したことのある Linux ディストリビューションは Linux Mint の他、Ubuntu, Vine, CentOS, Scientific, Fedora, RHEL, Mageia, Momonga, Sabayon, Calculate, SolusOS, Stella, arch, puppy, windos, pear, knoppix, KUbuntu, LUbuntu, openSUSE, snow, ultimate edition, zorin OS 等多岐に渡る。

¹¹<http://distrowatch.com/>

¹²<http://www.distrofreak.com/>

mint インストール」などとしてインターネットを検索し、適切な情報を探しだしてインストールしていただきたい。

0.3.4 Linux の端末操作

Linux をインストールしたら「端末」を探して開いてみよう。端末は Linux の CUI 環境を GUI 上で再現するソフトで、標準で用意されているものの他、機能をつけた様々な端末も出回っている¹³。

端末ではコマンドと呼ばれる様々な命令を打つことで、様々な処理を行うことができる。以下、これについて見ていこう。

【パス】

端末操作をする前に、端末操作に必要な「場所」の概念について説明をしておく。

端末を起動した場合、ユーザーのホームディレクトリと呼ばれるディレクトリ (=フォルダ) から始まるのが基本である。これが「現在見ているディレクトリ」(カレントディレクトリ)である。一般に、ホームディレクトリは~と、カレントディレクトリは. と、カレントディレクトリの一つ上の階層のディレクトリは.. と表される。

ディレクトリやファイルの場所を示す、いわばファイルの住所をパス (path) と呼ぶ。パスには、現在いる位置を起点として記す相対パスと、すべてのディレクトリの根本に当たる/からその場所を示す絶対パスがある。例えば、./Document などと記せば、カレントディレクトリの下に Document というファイルを示すし、/home/user/Document のように記せば、/home というディレクトリの下に user というディレクトリの更に下にある Document というファイルを示す。Linux の場合、パスの区切りは/によって行われる。

【端末のコマンド操作の例】

端末には多くのコマンドがある。ここでは、プログラミングに必要なコマンド操作を体験してみよう。

まず、カレントディレクトリを調べることにしよう。この時には

```
pwd
```

というコマンドを実行すれば良い。その後、

```
ls
```

というコマンドを実行すれば、カレントディレクトリの中身を見ることができる。より詳細に見たい場合には

¹³例えば筆者は Guake という名前の端末を使っている。

```
ls -al
```

と`-al` オプション¹⁴をつければ良い。

この下に `test` ディレクトリを作ってみよう。ディレクトリの作成には

```
mkdir ディレクトリ名
```

というコマンドを用いる。この場合は

```
mkdir test
```

とすれば、カレントディレクトリの下に `test` というディレクトリが作られる。

次いで、今作ったディレクトリの中に移動しよう。フォルダを移動するには

```
cd 移動先フォルダのパス
```

というコマンドを実行する。従って、ここでは

```
cd ./test
```

のように打てば良い。この際、パスをいちいちすべて打つのは大変なので、`tab` キーを押しながら打つことをすすめる (特段なければ、`./t` まで打った後に `tab` を押すと良いはずである。)。 `tab` キーは入力を補完してくれる機能で、コマンド、パスを含め端末の様々な場面で使うことができる。

ここに、空のファイル `empty` を作成してみよう。これには

```
touch ファイル名
```

とすれば良い。

更に、このファイルをコピーして、`empty2` というファイルを作成してみよう。ファイルのコピーは

```
cp コピー元ファイル名 (パス) コピー先ファイル名 (パス)
```

とすれば良い。

次は、`empty` をリネームして `empty3` にしてみる。リネーム及びファイルの移動は

```
mv ファイルの移動 (リネーム) 元 ファイルの移動 (リネーム先)
```

と記す。

最後に今作ったファイルとディレクトリを削除してみよう。ファイルの削除には

¹⁴オプションとは、実行するコマンドに条件等を付す機能で、通常`-`や`--`等の後に決まった文字 (列) を記して指定する。

`rm` 削除したいファイル名

を用いる。ここでは `empty2` と `empty3` を削除しよう。これを 1 行ずつ実行しても良いのだが、面倒であるので、

`rm empty?`

というコマンドを実行してみると良い。これにより両ファイルが消えるはずである。ここで用いた `?` は後に示すワイルドカード表現と呼ばれる表現である。

ディレクトリを消すため、まず、一つ上のディレクトリに移る。これは

`cd ..`

を実行すれば良い。最後に

`rmdir` 削除したいディレクトリ名

とすれば、ディレクトリが消える。

なお、コマンドを調べるためには

`man` コマンド名

と記せば良く、これによってコマンドのマニュアルを見ることができる。また、多くのコマンドは `--help` オプションを付して実行することで簡易ヘルプを表示してくれる。

ここで、プログラミングに使うコマンドはひと通り体験したつもりであるが、これ以外のコマンドで知りたいものがあれば `man` コマンドなどを利用して調べて欲しい。なお、Linux のコマンド一覧はインターネットで探せば多く見つかる¹⁵。

【ワイルドカード表現】

先に使った `?` はワイルドカード表現と呼ばれる表現であり、`?` 以外にもうひとつ `*` という記号を用いる場合もある。これは、似たような名前のファイルに対して同じ操作をする場合などに、まとめて行うことができる手法である。

`?` のワイルドカードは、1 文字の任意文字を示す。例えば、`empty?` と先に書いた例は、`empty1` や `empty9` など、`empty` の後に 1 文字を付した形のファイル、という意味になる。

`*` のワイルドカードは、0 文字以上の任意文字列を示す。例えば、`*.txt` と書いた場合、`code.txt` や `temp.txt` など、最後が `.txt` で終わる全てのファイル/ディレクトリを示している。

¹⁵例えば、<http://itpro.nikkeibp.co.jp/article/COLUMN/20060224/230573/> など。

【リダイレクト】

コマンドを実行した場合、その結果はコマンドラインに出力される。また、実行されたコマンドによっては、更に入力を要求される場合もある。この入出力先を変更する方法としてリダイレクト (redirect) がある。

リダイレクト

リダイレクトを行う場合には、コマンドに続けて

>出力先ファイル名

や、

<入力元ファイル名

を記す。これらを両方共用いることもできる。

試しに、

```
ls -al > ls_list.txt
```

というコマンドを実行してみると、ls_list.txt というファイルに ls -al コマンドの実行結果が出力されているはずである (more コマンドや less コマンドで確認してみよ)。

作成した CUI ソフトのテスト時などに、これらのリダイレクト入出力が役立つ。

0.3.5 Linux での環境構築

最後に、Linux での C 言語の環境構築方法を述べておく。

まず、

```
which gcc
```

により、gcc というコマンド (ソフト) がインストールされているかどうか確認する。このコマンドを実行して、もしもパスが表示されなかったならば、後でテキストエディタをインストールする際に gcc を同時にインストールする必要がある。gcc はコンパイラというソフト¹⁶で、簡単に言うと C 言語で書いたプログラムのソースから実行ファイルを生成するためのソフトである。

次いで、自分の好みのテキストエディタをインストールする¹⁷。以下では emacs を例にとるが、gedit, tea, leafpad, editra, vim など、好きなテキストエディタを用いれば良い (その際は、emacs を他のテキストエディタ名に読みかえること)。

ソフトをインストールする場合、Linux Mint や Ubuntu ならば

```
sudo apt-get install ソフト名
```

¹⁶gcc の他、Intel や AMD, Microsoft, Embarcadero などからもコンパイラが出ている。

¹⁷インターネットに接続する必要がある。

を実行し、その後にユーザーパスワードを入れれば良い。

Vine の場合は

```
su
```

コマンドを用いて (管理者のパスワードを打ち込み) 管理者モードになった後

```
apt-get install ソフト名
```

によりインストールを行う。

この他、ディストリビューションによっては、`apt-get` の部分が `yum` 等、別のコマンドに変わる場合がある。

上記より、Linux Mint に `gcc` と `emacs` を導入する場合は

```
sudo apt-get install gcc emacs
```

というコマンドを実行すれば良い。

`emacs` をインストールしたら、端末を再起動した後、

```
emacs ファイル名
```

を実行すれば、`emacs` が立ち上がり、引数で指定したファイルを編集できる。

本講の要点

本講ではプログラミングを行うための知識・環境の準備を行った。以下、知識のみまとめておく。

記数法

- n 進法とは、 n を位取りに用い、 n^k を各位とする記数法である。
- n 進法から 10 進法に変換する場合には、単に各位の数とその位の元となる n^k の積の総和を取れば良い。
- 10 進法から n 進法に変換する際には、順次 n で割って剰余を列挙し、この剰余を逆から順に記せば良い。
- 8/16 進法は 2 進法と相互に直接変換することができるため、コンピュータでよく用いられる。
- 2 進法 1 桁の事を 1bit と呼び、8bit のことを 1octet と呼ぶ。
- 通常のコンピュータでは、1octet のことを 1Byte と呼び、データの単位に用いている。

コンピュータの仕組み

- パソコンは入力装置・出力装置・CPU・記憶装置からなる。
- 主記憶装置は、CPU からの「近さ」に応じて、レジスタ、キャッシュ、メモリなどと分けられる。
- メモリへのアクセスは通常ランダムアクセスである。
- メモリはプログラム領域・静的領域・スタック領域・ヒープ領域に分けられる。
- メモリ上での位置は、アドレスを用いて示される。
- スタック領域はアドレスが大きい側から小さい側へ向かって取られ、プログラム毎に使用限度が定められる。一方ヒープ領域は、アドレスが小さい側から大きい側へ向かって取られ、空いている限り使うことができる。
- ソフトウェアはその機能から、OS と応用ソフトウェアに分かれる。
- ソフトウェアの UI は、GUI と CUI とに分かれる。

第1講 プログラミングとは何か

それでは、プログラミングについて学んでいこう。ここではまず、プログラミングとは何かという概論を話す。次いで、どのようにして作るのかという基本的な手順を学んだ後、標準出力への出力という、最も単純なプログラムを組んでみる。

1.1 プログラミングとは何か

「プログラミング (programming) とは何か」というタイトルはこれからプログラミングを学ぼうという人にとって大変親切で、かつ適切なタイトルであるが、同時に様々な意味をもち、ある程度プログラムに慣れた人にとってもある種の意味をなす質問だろう。プログラミングとは何か、その質問に対して、もちろん、一般的に辞書的に意味を説明するのが本講の目的だが、その質問は辞書的以外の意味をも十分持ちうるのである。例えば「私にとってプログラミングとは何か」という、人の思いに対する質問とも捉えられるし、「社会的にプログラミングとはどういった価値をもつのか」という、位置づけを問うこともできる。ある種のジョークのように聞こえるかもしれないが、哲学的に「プログラミングとは一体何なのか?」と考察することもできよう¹。ともかく、「プログラミングとは何か?」という問いはその文脈によって大きく変わりうるものである。そして、読者諸賢には、本書の内容を身につけていただくにしたがって「私にとってプログラミングとは一体何か?」ということも考えていただきたい。

さて、プログラミングとは一体何なのか、辞書的にみていくこととしよう。手持ちの「新明解国語辞典(三省堂)」を引くと、次のように書かれている。

- プログラミング：コンピュータにさせる仕事の手順を詳しく分析し、プログラムを作ること。
- プログラム：コンピュータに実行させる計算処理の手順を、コンピュータに受け入れ可能な一連の命令文の形で並べて書いたもの(あるいは書くこと)。

一方、Wikipedia には次のように書かれている。

- プログラミング：プログラムを作成することにより人間の意図した処理を行うようにコンピュータに指示を与える行為である。

¹ここではジョークのように聞こえると書いたが、決してプログラミングと哲学が無関係なわけではない。例えば Prolog という言語は AI(人工知能)の研究によく用いられるプログラミング言語であるが、AIの根幹の部分ではルートヴィヒ・ウィットゲンシュタインの「論理哲学論考」などが大きな役割を果たしている。哲学に不案内であっても、「AI というものがあるとき、それをそもそも作ったプログラミング言語というのは一体何か?」という問いかけは、「人間の知能のプログラミング言語にあたるものは存在するか?」という哲学的に思える問いに直結することがわかるだろう。

- プログラム：コンピュータに対する命令処理を記述したもの。

見比べればわかる通り、どちらも大差ない説明である。そして、どちらも間違っていないが、どうも無味乾燥であり、いかにも難しいといったもののようになってしまう。

プログラミングを好んでいる人がいて、それを趣味としている筆者のような人もいるほどなのだから、こんなにも冷たい説明では実感がわきにくい。辞書の説明だから冷たいのは当然であるが、もう少し具体的に、躍動感のある意味を追求してみよう。我々が日常的に目にするドラマ・映画などはお芝居であり、脚本がある。芝居のときに人間がどのように動けばいいかを記したのが脚本であるのと同様に、何かをなすとき、例えばゲームを動かすときに、コンピュータがどのように動けばいいかを記したものがソースコードというプログラムの”もと”である。マニュアルの作成と言ってもいいが、手順書を作成し、コンピュータを動かすことで何かをなす、それがプログラミングである。その何かはゲームでもいいし、音楽でもかまわない。あるいは文書処理や翻訳かもしれない。とにかく、そういった行為を行う際に動く手順を定め記すのがプログラミングである。それは脚本を書くように創作的な行為であり、でき上がりのドラマを見る楽しみをコンピュータ上で味わえるものである。プログラマーは脚本家と監督を兼任し、コンピュータが俳優の役割をする。そうしてでき上がったドラマをプログラムと呼んでいる。このように記せば、プログラミングが生きたものとして伝わってくるのではないだろうか。

1.2 プログラミング言語と C 言語

先に記したとおり、プログラミングとはコンピュータがどのように動けばいいかを記す行為である。何かを記すには、言葉が必要である。プログラミングを行うための言語をプログラミング言語 (programming language) という。プログラミング言語は多種多様であり、各言語はそれぞれに特徴を持っている。そのため、一般には利用目的に適したプログラミング言語を選ぶ。

プログラミング言語にはいくつもの分類があるが、ここでは、C のソースコードを書いてそれが実行されるまでの流れを理解するのに重要な「高級言語」「低級言語」を紹介する。

1.2.1 高級言語・低級言語と処理の流れ

通常、プログラムは1と0からなる機械語およびそれをもう少しまとめたバイトコードとしてメモリにロードされ、それが実行される。これは、コンピュータが唯一理解、実行できる形態である。プログラミングとは、この1と0を適当な順番に配置し、希望する機能を実現することである。だが、直接1と0を配置するのは問題がある。というのは、CPU に応じて命令セットが違うためである。また、この形式では当然、人間にはわかりにくい。そこで、これらの汎用性を高め、人間にわかりやすいようにした。このようにして生まれた、人間にわかりやすく、CPU の命令セットによらずにプログラミングできる言語のことを高級言語・高水準言語と呼ぶ。

高級言語で書かれたソースは、各々の CPU において、その機械語と1対1に対応する命令によって書かれたバイトコードに変換される。このように、機械語及びそれと1対1

対応する、機械毎に固有の言語 (命令セット) を低級言語・低水準言語と呼ぶ。また、この変換 (翻訳) 行為をコンパイル (compile) と呼ぶ。我々は高級言語、ここではそのひとつである C 言語を用いてプログラムを記述し、そのソースをコンパイル、低級言語で書かれたバイトコード (実行ファイル) を生成して、プログラムを実行するのである。

1.2.2 C 言語の特徴

C 言語は高級言語であるが、その特徴から俗に「高級言語の顔をした低水準言語」や「中級言語」などと呼ばれることもある。その特徴には、次のようなものがある。

C 言語の特徴

- C 言語は、構造化手続き言語^aである。
- メモリのレベルまで含めた、かなり機械に近い処理ができる。
- システム記述に向いている言語である。
- プログラミング人口が多い。Tiobe programming community^bによれば、2002 年以後世界で 1~2 位のシェアを誇る。
- 非常に自由度が高く、大抵の機能を実装することができる。これは、悪く言えば、コンピュータにとって危険な処理も作れてしまうということである。
- 基本的にプログラマに信頼を寄せた言語となっており、自動的に行われる処理は少ない^c。これは、自由度が高いがゆえにプログラマに重責があるということでもある。
- 様々な言語の基になっている。

^a構造化手続き言語とは、E.W.Dijkstra(1930-2002) の提唱した「構造化プログラミング:あらゆるアルゴリズムは連接 (順次)・分岐・反復の 3 構造により実現でき (構造化定理)、その 3 構造を関数などによって適度に分離、ルーチン毎に細部を記述していくというプログラミングパラダイム」に従い、手続きを順に記すことでプログラムを作成する言語である。

^b<http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html>

^c逆に、Java は危険な処理を自動的に検知し、些か口煩いと思う人もいるぐらいに忠告をする言語である。

とりわけ、自由度が高く、低水準命令が可能でシステム記述に向いていることから、UNIX を始めとした OS カーネルや組み込みプログラミング²に利用されている。

1.2.3 C 言語の歴史

C 言語は現在一般に使われているプログラミング言語ではかなり古い部類に入る言語である。C++や Java などの言語が C の構文を元に行っていることからわかるとおり、非常に多くの分野で使われている言語であるが、この言語にも元の言語が存在する。それが BCPL と B である。前者はオペレーティングシステムやコンパイラを書くために開発さ

²家電等「いわゆるコンピュータ」ではない様々な機器のシステム等のプログラミング。

れた言語で、これをモデルにして UNIX オペレーティングシステムの初期バージョン開発に向けて作られたのが B であった。1960 年後半～70 年代にかけてのことである。

アメリカ・ベル研究所の D.M.Ritchie(1941-2011) は B を元にして C 言語を作った。これが C 言語のはじめである。この時代における C 言語は、小型で効率的であり、しかも強力なプログラミング言語であったため (現代でも強力と言い切れるかどうかは、特に C++ と比べてそう言えるかどうかは、疑問の余地が残るが)、多く使われるようになっていった。だが、この広まり方は C 言語の“方言”とでも言うべき、様々な「規格」を生み出す要因となってしまった。この是正のため、1982 年、ANSI(American National Standards Institute) 内で委員会が組織され、その時点で最も広まっていた C の方言である「K&R C」³を基に標準規格を定める運びとなった。委員会発足から 7 年を経た 1989 年、規格化された C 言語、現在に至るまで ANSI C(C89) と呼ばれる C 言語が誕生したのである。現在のコンパイラは少なくともこの規格に則っているものとして問題ないし、また、この規格より以前の C 文法に従ってコードを書く意味はないと言って良い。この規格がそのまま国際標準となり、現在の C の基盤をなしているからである。

だが、1989 年の規格に一切の改良が行われなかったわけではない。とりわけ、1999 年の C++ を意識した規格の変更は大きなものであった。本書を書いている 2012 年現在、1995 年と 1999 年、2011 年に、C 言語は三度の改訂を経ている。1995 年の改訂は 89 年制定の C に対して、各国の文字を扱えるようにする改訂であった。これは、それまでの C に対して軽い追加が行われた程度のものであり、大きな変動ではなかった。

一方、1999 年制定の C は C99 とも呼ばれるかなり大幅な拡張であった。89 年の C に対して、C++ を意識した拡張を行ったのがこの改訂である。1989 年以降の C で書かれたコードであれば、大抵は現在のこの C99 規格でも動くと考えて差し支えない。だが、この規格は意外と知られておらず、書店に並んでいる本を見ても、この事まで書いている本は多くなく、むしろごく一部に C99 の概念が取り入れられている程度に過ぎない本が多い。特によく目にするのが、// によるコメントを説明していながら (このコメントは C99 規格で取り入れられたもので、C89 の規格では通用しない。コメントについてはあとのページを参照されたい。)、これがあたかも昔からの C に存在するかのように書いている本である。明確にされていないが、背景で C99 を使っているのである。

2011 年に C++11 という C++ の新規格と共に出された C の新規格は C11 と呼ばれている。ここでの改訂は、C 言語をより現代的な視点に対応させるための改訂と言える。しかしながら、この規格は、出してから時間が経過していないという事もあり、対応しているコンパイラが少ない。また、本書の段階で C11 で新たにできた文法が必要になることは少ない。このことを踏まえ、本書では C99 の規格を基に、旧来の C 規格にも通用する内容を記述することにする。C11 については脚注などで触れるに留める。

³B.W.Kernighan(1942-) と (先に名の出してきた)Ritchie の C という意味である。この二人は「プログラミング言語 C(The programming language C)」という C の原点とも言うべき書籍を著した。惜しいことに、Ritchie は 2011 年 10 月に亡くなってしまったが、Kernighan は存命である。「プログラミング言語 C」は ANSI 規格 (C89) に対応した第 2 版が出版されており、通称「K&R 本」として広く知られている。なお、日本語版は石田晴久氏の訳が共立出版より出版されている。

1.3 プログラム作成の一連の流れ

プログラムの作成は概ね、次の順で行われる。

1. 作成したいものを決める。(プログラムの計画)
2. 作成したいものをどのようにしてプログラムにするかを決める (アルゴリズム (algorithm) を立てる)。
3. 立てたアルゴリズムにしたがってソースコードを打つ (コーディング (coding))。
4. 作成したソースコードをコンパイルして実行する。
5. コンパイル中ないしプログラムの実行中にエラーが出た場合は、そのエラーを取り除く (デバッグ (debug))。
6. ソースコードの無駄な部分を取り除く (プログラムの改良)。

これを実際に体験してみよう。

1.3.1 コーディングまでの体験

はじめなので、あまり難しいプログラムを書くことはできない。今回は最も単純に、標準出力に

```
hello, world
```

と出力するプログラムを作成してみよう。なお、第0講で記したとおり、以降のプログラムは (特段の記述がなければ)Linux に gcc をインストールしていることを前提に行う。

まず、端末を開き、適当なフォルダを作成し、そこに移動する。例えば、

```
mkdir Clang
cd Clang
mkdir class2
cd class2
```

とすれば、ホーム以下に Clang というフォルダができ、その中に class2 というディレクトリを入れて、そこで作業することになる。

フォルダを移動したら、テキストエディタを開こう⁴。

```
emacs Hello.c
```

⁴第0講にも記したが、emacs はテキストエディタに応じて tea,editra,gedit 等に読み替えること。

このように書けば、Hello.c という名前のソース (Source code) を作ることになる。ここで、ソースというのはプログラムの内容を記述したファイルである。

それでは、次の問題を読んで、書いてある解答例 (リスト 1.1) を打ち込んでみよう。この時、ソースには、適度に空行やタブ (インデント) が入っているが、これはソースを読みやすくするための工夫であるので、その部分も真似してみたい。但し、誤って全角空白を入れないこと (エラーになる)。なお、以降のソースでは、紙面節約のためスペースや空行を省くが、単語等を分割しない範囲で、読みやすいように空白等を入れたほうが良い。

【hello world プログラム】

”hello, world”と出力するプログラムを作成する。

【解説】

hello, world と出力するプログラムは、一般に「ハローワールド」と呼ばれており、多くの言語の入門書の最初に挙げられている「世界一有名なプログラム」である。初出は先に記した「プログラミング言語 C」であるとされる。

その名の通り、ただ hello, world と出力されるだけのシンプルなプログラムであるが、言語の書式を学ぶ上で、とても重要なプログラムでもある。

リスト 1.1: ハローワールド

```
1 /* hello, world
2    puts Version */
3
4 #include<stdio.h>
5
6 // main function
7 int main(void){
8
9     //output for stdout.
10    puts("hello, world");
11
12    return 0; //val=0
13 }
```

以上を打ち込み終わったら、保存して (emacs の場合、ctrl+x を押した後、ctrl+s を押せば保存できる)、テキストエディタを閉じよう。

1.3.2 コンパイル・実行

さて、それではコンパイルして実行ファイルを生じよう。コンパイルのコマンドは次の通りである。

プログラムのコンパイル

あるソースを gcc でコンパイルして、実行ファイルを得るためのコマンドは

gcc (ソースファイル名) -o (出力実行ファイル名)

である。この時、-o 以下を省略すると、a.out という実行ファイルができる^a。

^aWindows の場合は、a.exe という実行ファイルができる。

したがって、

```
gcc Hello.c -o Hello.out
```

とすれば、実行ファイル Hello.out が生成されるはずである。もし何らかのエラーが出たら、どこかに打ち間違いがあると考えられるので、再度テキストエディタを開いて間違っている場所を調べてみよう。このように、誤り (バグ (bug)⁵) を取り除くことをデバッグ (debug) と呼ぶ。

続いて、できたファイルを実行する。これは、

```
./Hello.out
```

などとすれば良い。正しくできていれば、コンソール上に

```
hello, world
```

と出力されるはずである。

以上のようにして、プログラムを作成することができる。今後、この手続きは何度もやっていくことになるので、今取り立てて覚えずとも、そのうち自然と身につくだろう。

1.3.3 「はじめてのプログラム」の解説

さて、それではリスト 1.1 のプログラムがどのような動作をしているのか、コードを追いつつ順に説明していこう。

【コメント】

このプログラムの 1,2,6,9 行目はコメント行であり、12 行目にもコメントがある。コメント (comment) とは、ソースコードのうち、我々人間のために覚え書きとして挿入される注釈のことである。この部分は、コンパイルの際に何も書いていないものと同様に扱われるので、好きなことを書くことができる。この例のようにタイトルを書いたり、文法解説やアルゴリズム解説の際に横に入れたり、デバッグの際に一時的にコードを無効化したりと、色々な用途に使う。なお、本書では紙面節約のため、掲載ソースリストのコメントは省く。だが、一般には別人⁶が読んでもわかるようにコメントを書くべきである。

C 言語では、コメントは 2 種類ある。

- // から行末まで。
- /* から始まり、最初に出てくる */ まで。

⁵原義は「虫」であり、トーマス・エジソンも使っていたことが知られているが、「世界最初のバグ」は昔の大型コンピュータに実際に虫が入ったものであるとされている。

⁶たとえば自分自身しか使わないソースコードでも、コメントを書くに越したことはない。というのは「1 か月後の自分は別人と思え」と言われるほど、ソースコードの細部は覚えにくいものだからである。

前者は1行のコメントまたは命令のすぐ横に使う場合が多い⁷。このように、行末までをコメントとみなすコメントの形式をインラインコメント (in-line comment) と呼ぶ。一方で、後者は、タイトルで使っているように、何かまとまったコメントを書きたい場合に使うことが多い。このように、ある記号から始まりある記号で終わる (この場合、/*から始まり*/で終わる) コメントの形式をブロックコメント (block comment) と呼ぶ。

コメントは冗長すぎても良くないし、少なすぎるのも後々困ることになる。コメントの基準は人によるが、概ね処理の要約を基本に、難解な処理に解説をつけていく形でコメントを付すと、読みやすいソースになる。

【include 文】

コメントを抜いて、実質的に意味があるのは4行目である。この行は **include 文** と呼ばれる文で、< >で囲まれたヘッダファイル (header file) を読み込む⁸ という意味である。ヘッダファイルというのは、C の様々な命令 (関数) の入っている辞典のような物だと思えば良い。(実際には、辞典そのものはライブラリ (library) と呼び、ヘッダファイルはこれを引くための道具である。) これらの辞典を読み込むことにより、コンパイラは命令を解釈することができる。このことからわかるとおり、この一文はコンパイラへの命令である。このようなコンパイラへの命令のことを前処理命令またはプリプロセッサ命令 (preprocessor command) と呼び、#から始める⁹。include はヘッダファイル等の読み込みを行う前処理命令である。

今回読み込んだ stdio.h は standard input and output (標準入出力) のライブラリで、その名前通り、入出力に関する命令が収められているライブラリである。このように、C 言語が仕様として標準で用意しているライブラリのことを標準ライブラリと呼ぶ。

以上を踏まえると、リスト 1.1 の第4行は、「以降で標準入出力の命令を使えるようにするため、stdio.h のライブラリを読み込め」という意味になる。

【main 関数】

プログラムのソースコードは原則上から下に向けて解釈される。だが、実際にはプログラムの主幹を司る部分を各プログラムに一つ作り、その部分を実行することによってプログラムが動く。つまり、プログラムというのは、主部より前に準備のための文があり (先の include など) はまさにそれである)、主部があり、その後ろに (必要に応じて) 主部を詳述する部分がある、という構成になる。このうち、準備のための文や主部を詳述する部分は必要ない場合もあるが、主部はプログラム毎に必ず1つ必要である。この主部を示すのが **main 関数** である (リスト 1.1 の、第7行から第13行の部分)。当面の間、main 関数は次のように記せばよい。

⁷実は工夫次第で2行以上のコメントも書けるのだが、わかりにくいので使わないほうが良い。

⁸実際にはダブルクォーテーション (") で囲む場合もある。これについては、後の講で説明する。

⁹この記号 (#) は「シャープ」ではなく「ハッシュ」である。二つの記号は線の角度が違う。

main 関数の書き方

main 関数は、次のような形式で記す。(但し、後に学ぶコマンドライン引数を用いる場合はこの限りではない。)

```
int main(void){
    処理を記す
}
```

main 関数の定義にある int,void については後の自作関数の講で詳しく解説するので、ここでは「このように書くものである」とっておいて良い。

【puts 関数】

リスト 1.1 の第 10 行目は今後 C プログラムを書いていく上で必要なことを多く含む、示唆に富む 1 行である。動作は”hello, world”と標準出力 (コンソール) に出力するだけのものだが、それ以外の、ルールなどの面でこの一文から学べるものは多い。

puts は stdio.h に入っている関数で、put string の意味である。この関数は引数 (ひきすう, argument) の文字列を標準出力に出力し、最後に改行する関数である。ここまでさらに説明したが、新たな用語が連続しているので、その用語を見ていくこととする。

関数というのは後ろに () を伴い、何らかの値を与えると、それに対して処理を行い、何かの値を返すものと言う。ここであげた puts 関数は、文字列を引数にとり、その引数の文字列を標準出力に出力し (処理)、無事に出力できたかどうかを (内部的に) 返す。この時に返す値のことを返却値¹⁰(return value) と呼ぶ。

少しわかりづらいかもしれないので、出前を例に考えてみよう。まず、出前して欲しい店に、電話などで注文を行う。これを受けて、店では注文されたメニューを料理する。その後、店から自分のところに注文した料理が届く。この、注文が引数であり、店での調理が処理、自分のところに届いた料理が返却値である。つまり、出前という関数は「注文」を引数にとって、それに対応した「料理」を返却すると言える。

ここまでで puts の意味はわかったと思うが、後 2 点、ダブルクォーテーション (") の意味と、セミコロン (;) の意味について見ていこう。

ダブルクォーテーションは二つセットで、その間が文字列であることを示す¹¹。

セミコロンは、C の各文の最後に付ける記号で、日本語の句点 (。) や英語のピリオド (.) に相当するものである。

【return 文】

先に書いたとおり、関数は返却値を持つ¹²。そして、main 関数は名前が示唆する通り、関数である。したがって、main 関数も返却値を持つ (ようにしている)。この返却値を指

¹⁰返し値、戻り値などとも言う。「返却値」は JIS 規格での呼び方である。

¹¹厳密には文字列リテラルというものであり、これについては後に詳述する。

¹²持たない関数もあるが、これは「持たない」ようにしているから持たないのであって、例えば標準ライブラリでは持つ関数のほうが多い。

定するのが return 文である。

また、関数は返却値が定まった時点で終了するので、関数の終了を示すために return 文を用いる場合もある (後述)。

return 文で返却値を定める場合は、次のように記す。

return 文

関数の返却値には return 文を用い

```
return (返却値);
```

と記すことによって返却値を指定できる。(返却値を持たない関数の終了のみを示す場合、返却値を指定しないこともある (後述)。)

main 関数は、正常終了した場合 0 を返すように作る (つまり、return 0 を処理の終端に書く) のが慣例である。

1.4 標準出力への出力

簡単なプログラムを一つ作ったところで、もうひとつ、出力するだけのプログラムを作ってみよう。

【簡単な表の出力】

簡単な表を出力してみる。

【解説】

水平タブ文字を用いて場所を揃えることにより、簡単な表を作ることができる。今回は、数学の記号とその正式名称を出力するプログラムを書いてみよう。なお、8 行目が 2 行に渡っているが、実際のプログラムでは 1 行に続けて書くこと。

リスト 1.2: 数学の記号表出力

```
1 #include <stdio.h>
2
3 int main(void){
4     printf("Math words.\n");
5     putchar('\n');
6
7     puts("_sin_ \t _lim_ \t _sup");
8     puts("_sine_ \t _limit_ \t _
9         supremum");
10    return 0;
11 }
```

【エスケープシーケンス】

リスト 1.2 には `\n` や `\t` といった文字が登場する。これらは各々、改行や水平タブを表している。このように、コード上で意味を持っているなどの理由で書き表すことができない文字に対しては、代替となる書き方が用意されている。このような文字のことをエスケープシーケンス (escape sequence) とよび、`\` 記号¹³に文字を付けることで実現される。エ

¹³ バックスラッシュ。日本の円マークも同じ役割であり、バックスラッシュが出ない代わりに円マークが用いられる場合もある。

スケープシーケンスは2文字(以上)に見えるが、実際には`\n`や`\t`で1文字である。よく使うエスケープシーケンスを表 1.1 に示す。

表 1.1: よく使うエスケープシーケンス

文字	意味	文字	意味
<code>\0</code>	NULL 文字	<code>\'</code>	シングルクォーテーション
<code>\t</code>	水平タブ文字	<code>\"</code>	ダブルクォーテーション
<code>\r</code>	復帰文字	<code>\\</code>	バックスラッシュ
<code>\n</code>	改行文字	<code>\?</code>	クエスチョン

表 1.1 に載っている`\r`はCR(キャリッジリターン、carriage return)で、Macでの改行コードである。一方`\n`はLF(ラインフィールド、line field)で、Linuxなどでの改行コードである。Windowsの改行コードはCR+LF(`\r\n`)であり、メールの通信などでもCR+LFが使われている。

エスケープシーケンスは、リスト 1.2 のように、文字列の間や、あるいはそれ単独で用いて、表 1.1 に示したような意味になる。

【printf関数と putchar 関数】

リスト 1.2 において新しく出てきた関数に `printf` 関数と `putchar` 関数がある。これらについて説明する。

`printf` 関数は、後からより詳しく説明するが、書式指定文字列という文字列を出力する関数である。現状での動作は、`puts` 同様、引数の文字列を標準出力に出力するだけだが、`printf` 関数は出力した後最後に改行しない関数である。そのため、リスト 1.2 の 4 行目では、最後に`\n`を書いて意図的に改行させている。

`putchar` 関数は文字を一文字だけ、標準出力に出力する関数である。C 言語ではシングルクォーテーション記号で囲むことで、それを文字として扱う(囲まなければ、命令や後に記す変数などとして認識してしまう)。ここでは、`\n`という一文字がシングルクォーテーションで囲まれているので、5 行目は改行文字の出力、すなわち「改行せよ」という意味になる。

`printf` 関数や `putchar` 関数を使うと、プログラムの終了直前の出力の最後に改行されないようなプログラムができてしまう。だが、これではコンソールが見難くなる。そのため、出力の終端では必ず改行を入れるようにしておくのがマナーである。

本講の要点

本講では、まずプログラミングとはどのようなものか考え、C言語の特徴について学んだ後、実際に簡単なプログラムを書いてプログラミングを体験的に学んだ。

C言語とプログラミングの概観

- プログラミングは、コンピュータの動作手順を記し、そこから実行ファイルを生成する行為である。
- C言語はシステム向けのプログラミング言語で、高い汎用性を持つ。
- プログラミングを行う際には、作るものを設計し、アルゴリズムを定め、コーディングした後コンパイルし、最後にデバッグを行って完成という流れが一般的である。

C言語の基礎的な文法と標準出力

- ソースを書く際には、必要に応じてインデントを入れたり、理解を助けるための注釈であるコメントを入れたりすることで、読みやすく書くのが大切である。
- コンパイラに対する命令を前処理命令と呼び、ハッシュ(#)の後に記す。
- include文は必要なライブラリを読み込む前処理命令である。
- プログラムには主部であるmain関数が必要である。
- 各文の終わりにはセミコロン(;)を付す。
- main関数では、慣例としてreturn 0;により、返却値として0を返す(正常終了の場合)。
- 文字で表すことのできない改行やタブといった記号を表すために、エスケープシーケンスがある。
- " "で囲まれた部分は文字列である。
- ' 'で囲まれた部分は、プログラム上で文字として扱われる。
- puts関数やprintf関数、putchar関数を用いて出力を行うことができる。

第2講 変数の概念

前講では、パソコンが何かを出すプログラムを作成した。一方、通常のソフトウェアは何らかの指令——入力 (input)——を受けて、それをプログラム中で保持して活用する。ここでは、標準出力に対応した入力である、標準入力 (standard input) から入力して、それを保持・活用する為の手法である変数について学ぶ。

2.1 変数とデータ型

まずは、「変数とは何か」から掴んでいくことにしよう。

2.1.1 変数とは

変数 (variable) とは、プログラムのソースコードにおいて、扱われるデータを一定期間記憶し必要なときに利用できるようにするために、データに固有の名前を与えたものである。一人一人の人間が異なる名前によって区別されるように、一つ一つの変数も名前によって区別される。これにより、複数のデータを容易に識別することができる。また一般に、変数が表しているデータをその変数の値という。最も簡単に例えれば、データを入れる箱が「変数」であり、その箱の名称が変数名 (識別子)、その箱の中身が変数の値ということになる。

【変数使用の3ステップ】

変数を扱う際には、宣言 (declare) ・ 初期化 (initialize) ・ 参照 (refer) の3ステップが行われる。これは、次のような順番である。

1. まず、どのような名前で、何を格納するための変数なのかを定める (宣言)。
2. 次に、その変数に初期値が代入される (初期化)。
3. その後、必要に応じて代入されたり計算に用いられったりする (参照)。

これらについての例は次節以後に譲る。このような3ステップを経て利用されることを知っていれば良い。

【データ型】

先に何を格納するための変数なのか定めると書いたが、一般の容器に用途があるように、変数にも用途がある。例えば、整数用の変数、小数も入れられる変数、文字を入れる変数などである。このような、変数が何をを入れるためのものかというのを、その変数のデータ型 (data type) あるいは単に型と呼ぶ。より正確に言えば、メモリ上のビット列をどのように解釈して値を保持するか定めるのがデータ型である。そのため、変数を使えるように宣言するとは、変数の型と名前を定めるということである。

1999年に定まったC言語の規格における、基本的なデータ型 (基本型) を表 2.1 に示す。なお、C99以前の規格から定まっている型には何も記さないが、それ以降から定まった型には (C99) と記した。

表 2.1: C 言語におけるデータ型 (基本型)

型名	扱えるデータ
short	整数
int	整数 (範囲が short 以上)
long	整数 (範囲が int 以上)
long long	整数 (範囲が long 以上)(C99)
_Bool	ブール代数 (整数)(C99)
float	浮動小数点数
double	浮動小数点数 (範囲が float 以上)
long double	浮動小数点数 (範囲が double 以上)
_Complex	複素数型 (C99)
_Imaginary	虚数型 (C99)
char	文字 (整数)

各変数の型についての説明はここには記さず、別で丁寧に説明する。ここでは、各々に何が入るか、範囲がどれぐらいか (何より何が大きい) かということを知っておけば良い。

2.1.2 識別子の命名

変数の名前には規則があり、その規則を守らない名前は許されない。また、規則以外にもマナーもある。我々が変数はじめ様々なものを命名するときには、規則とマナーの両方を守らねばならない。これらの違いについて考えよう。

規則はコンパイラがソースファイルを解釈する際に必要とするものである。したがって、規則は変数を明確に識別できるものである必要がある。識別の為に使う言葉を識別子 (identifier) と呼び、C では概ね変数等の名前と同義と考えて良い。

マナーは、仮に守らなかったとしてもプログラムは動作するであろうが、可読性その他の観点において問題が生じるというものである。規則は実生活での法律に対応し、マナーはそのまま日常生活のマナーに対応する。スーパーマーケットのレジで並んでいるとき、それに割り込むのは、違法ではないかもしれないが、マナーに反する。同じことで、識別子を定める際にも、マナーというものがあるのである。

一般的な、C 言語における識別子命名のマナーと規則についてまとめておく。

【命名規則】

- 識別子の第1文字目は半角英字でなければならない。
- 識別子に含まれる文字は、半角英数字及びアンダースコアのみである。
- 識別子は前もってCの命令として用意されている単語である予約語 (reserved words) と同じであってはならない^a。
- 識別子は他の識別子と重複してはならない。^b

【命名のマナー】

- 識別子はその用途が明確になるように命名すべきである。
- 識別子はあまり長くするべきではない。
- 慣例的に定まっている識別子に反する名称をつけるべきではない^c。
- 大文字と小文字が違うだけのような間違いを誘発する名前は避ける。

^a予約語については付録で述べる。

^b厳密には、後で学習する名前空間が違っていれば同一識別子でも規約上問題ないのだが、マナーとして書かないほうが良い。名前空間が同一の場合はそもそも(規約上)付けられない。

^c例えば、後日学習する反復処理においてカウンタとして用いる変数には一般にはi,j,kを用いる。特別な理由なくこれら以外の識別子を使うとソースが読みにくくなる。

2.1.3 整数型の仕組み

それでは、実際に型の内部を見ていくことにしよう。ここでは、整数型と浮動小数点数型、文字型について解説する。

整数型の特徴は、次のとおりである。

整数型の特徴

- 符号付き (signed) 整数と符号なし (unsigned) 整数があり、宣言時に指定できる。ただし、signed は通常省略される。
- 符号付き整数は、定められた範囲における整数を扱うことができる。
- 符号なし整数は、定められた範囲における非負整数を扱うことができる。

ここに書いた、「定められた範囲」とは何だろうか。

整数型に限らず、変数はいずれも、メモリ上¹にある一定サイズの領域を確保する。よくある環境では、short 型は2Byte,int 型と long 型は4Byte,long long 型は8Byteである²。

¹ここで学ぶ変数については、メモリ上のスタック領域と呼ばれる部分である。

²整数型変数の型名は元々short int,int,long int,long long intであった。だが、長いのでintを略記し、こ

ここで、例えば short 型のビットパターン数は 65536 通り³である。これを全て 0 以上の数に当てはめると、0 から 65535 までの値を扱うことができ、これが unsigned short である。一方、先頭ビットが 0 であるようなビットパターンを 0 及び正に、先頭ビットが 1 であるようなビットパターンを負に当てはめると -32768 から 32767 までの値を扱うことができる。これが (signed) short である。

今見たように、変数には扱える値に範囲がある。この範囲を超えてしまうことをオーバーフロー (over flow) と呼ぶ。オーバーフローするとプログラムがおかしくなるので、変数の範囲については概ね覚えておくことが便利である。例えば、32bit 整数は約 21 億まで、64bit 整数は 20 桁の数の途中まで扱える。

【2 の補数表現】

先に、ビットパターンを割り当てると書いたが、その仕組みについてもう少し掘り下げることとしよう。なお、この内容は後で解説するビット演算に大きく関わるので、よく理解されたい。

まず、unsigned 整数型のビットパターンであるが、これは単純である。すなわち、表したい 10 進数をそのまま 2 進数に変換して、それをビットパターンに割り当てれば良いのである (勿論、オーバーフローはおきていないものとする)。

難しいのは符号付き整数型の場合である。このビットパターンには **2 の補数表現** と呼ばれる表現が用いられる。この表現は、次のようなものである。

まず、0 を基準とする。10 進法における 0 を、2 進法における 0 と対応させるのは自然な考えであろう。さて、これから 1 を引いた数、すなわち -1 をどう表すべきであろうか??

4 ビットで考えることとしよう。-1 を 1001 とすれば、先頭ビットが符号を示し、残りが絶対値を示すという風に解釈できる。しかし、この方法では、先頭ビットに応じて引き算や足し算を使い分けなければならない。そのため、内部的な実装が面倒になってしまう。

そこで、使い分けなくていいようにビットを定めることにしたのが 2 の補数表現である。1=0001 から 1 を引いて 0 にするには、0001 から 1 を引いて 0000 にすることができる。では、それより 1 小さいものはどうすればいいか。これは、1111 とすれば良いのである。ビットがあふれる部分への繰り上がりを無視して考えれば、 $1111 + 0001 = 0000$, $0000 + 0001 = 0001$ となり、確かに $-1 + 1 = 0$ となる。さらに、この方法を用いた場合でも、先頭ビット 0 は + または符号なしを、1 は - を示す。このように、 $-a$ のビット表現を、0 から a を引いたものとして表現するのが 2 の補数表現である。

参考として、表 2.2 に、2 の補数表現を用いた 4 ビットの場合の符号なし整数・符号あり整数一覧を示す。

ここに書いたように short, int, long, long long と書くことが一般化している。実際のプログラムでは、short int などと省略せずに書いても問題ない。同じことが unsigned int にも言えて、コードを書く際には unsigned とだけ書いても良い。

³2Byte=2×8bit なので、 $2^{16} = 65536$ と求められる。

表 2.2: 4bit 符号なし整数・符号あり整数の一覧

ビット列	符号なし整数	符号あり整数	ビット列	符号なし整数	符号あり整数
0000	0	0	1000	8	-8
0001	1	1	1001	9	-7
0010	2	2	1010	10	-6
0011	3	3	1011	11	-5
0100	4	4	1100	12	-4
0101	5	5	1101	13	-3
0110	6	6	1110	14	-2
0111	7	7	1111	15	-1

表 2.2 からわかるように、 n ビットの整数が 2 の補数表現により表されているならば、その数値の範囲は $-2^{n-1} \sim 2^{n-1} - 1$ である。

2.1.4 浮動小数点数型の仕組み

float, double, long double は浮動小数点数型といい、実数を格納するための変数である。この型は整数型と違い、符号が必ず存在し、誤差が生じうるという特徴を持つ。ここでは、これらが実数を格納する仕組みについて説明する。

【固定小数点数】

ビットパターンを小数に当てはめる方法として、もっとも考え易いだろうものが**固定小数点数** (fixed point number)⁴である。この方式は一般に、定められた範囲において誤差が出ず、計算も高速である代わりに、表現できる数の範囲が小さくなる。そのため、細かな値は不要だが誤差が生ずると困るような経済計算などで用いられている。C には固定小数点数型はないが、後述の浮動小数点数型の理解のために簡単な解説を行う。

固定小数点による数値表現は、ある定まったビットまでを整数部として、そこから下の部を小数部としてみなす方法である。例えば、16 ビットある時に、上位 8 ビットがその数の整数部を示し、下位 8 ビットが小数部を示すようにすれば良い。この時、 $2^0 = 1$ を示すビットのすぐ下のビットは 2^{-1} を、その下は 2^{-2} を... というように、桁が連続的に定まっていくなのが普通である。

【浮動小数点数】

浮動小数点数 (floating point number) は、小数点の場所を上手く変えながら (浮動) 表す方法で、C 言語で実数を扱う型はこの方式である。この方式の場合、表現できる数の範囲は広いが、誤差が出る上に、計算も固定小数点型に比べて遅い。そのため、経済計算等には向かず、科学計算などで使われることが多い。以下、これについて説明する。

⁴英単語からもわかるとおり、これは固定-小数点-数という事になる。したがって、「固定小数」という言い方は間違いである。ただし、慣用的に固定小数と使われている場合も少なくない。

浮動小数点方式では、変数のビットを符号部 (1bit)、指数部 (符号付き整数)、仮数部 (符号なし整数部) に分割して、これを用いて数値を表現する。すなわち

$$(\text{符号})(\text{仮数}) \times (\text{基数})^{(\text{指数})}$$

という表現である。一般に、基数には2が用いられ、仮数部は1以上2未満の数の小数点以下を示す⁵。この説明だけでは分かりにくいと思うので、実際の例を見ることにしよう。

-4.8 という数字を浮動小数点数で表現することを考えよう。 $-4.8 = -1.2 \times 2^2$ である。したがって、符号部には-が、仮数部には0.2が、指数部には2が格納されることになる。

このように、指数部の値によって、絶対値が非常に小さい値や非常に大きい値を計算できるようにしたのが浮動小数点数型である。

【例外演算】

先のような形式で定義された数では表現できないような計算もある。このような計算を例外演算 (arithmetic exception) と呼ぶ。例外演算は通常不要なものあるいはバグを示すものであるため、例外が起こっている場合にはその部分を直さなければならない。

整数の項で記したオーバーフローは例外演算の一つである。これは浮動小数点数でも起こりうる。すなわち、絶対値が大きすぎて表現できないような数になることである。

一方で、浮動小数点数の場合、絶対値が小さすぎて表現できない数になることもありうる。このように、非常に小さな絶対値となって表せなくなる現象をアンダーフロー (under flow) と呼ぶ。浮動小数点数におけるオーバーフロー/アンダーフローは、浮動小数点数の指数部の上へのオーバーフロー/下へのオーバーフローと対応している。

この他、0による除算や負の値の平方根も例外である。前者を行った場合、inf という特殊な値になり、後者の場合、NaN という特殊な値になる⁶。

また、浮動小数点型には通常の0に加え、符号部がマイナスになった-0がある。これも特殊な値であり、アンダーフロー等を示す場合もあるが、例外演算でないこともある。

これらについて詳しく覚える必要はないが、初心者のうちは、これらの見慣れない表現が出てきたら「何かおかしい」と思えば良い。

2.1.5 文字型の仕組み

文字型 char は1バイトの整数型でもあり、内部的には整数を保存している。これらの整数を、文字コードと呼ばれる規則に従って表示することにより文字を表現している⁷。例えば、非常によく使われる文字コードである ASCII⁸の場合、数字の9は57であり、文字

⁵IEEE754 方式と呼ばれる方式である。

⁶inf は infinity(無限) の略であり、NaN は Not a Number(非数) の略である。

⁷文字コードには様々な種類があり、それによって同じ数字でも違う文字に割り当てられることがある。

⁸American Standard Code for Information Interchange の略である。

A は 65 である。出力する際に、65 を ASCII にしたがって文字として出力すれば A になり、数値とすれば 65 になるわけである。

1 バイトで表される文字は少なく、256 通りしかない。このように 1 バイトで表されるのが、一般に言う半角文字である。半角文字を扱うパソコンは大抵 ASCII に従うため、パソコンでしか動作しないようなプログラム (競技プログラミングなど) は ASCII であることを仮定して作って良い。ASCII には「0~9」「A~Z」「a~z」の連続性が保証されているという特徴がある。このうち、0 から 9 の連続性については C の規格で保証されているが、それ以外は ASCII 固有の特徴であるので、ASCII であることが保証されない場合には使わないほうが良い。

文字は、プログラム中ではシングルクォーテーション (') で囲み '1' や 'A' のように記す。これはそれぞれ、文字 1, A の文字コードを示している。文字コードは内部的に整数として表されるので、例えば '1'+4 は '5' である。

2.2 変数の入出力と演算

長い説明で疲れただろうが、いよいよ、実際のプログラムを見ていく。

2.2.1 変数の出力

【変数の使用】

自分で定めた変数を出力するプログラムを作成する。

【解法】

変数の使い方は

1. 宣言する。
2. 初期化する。
3. 参照する。

である。これらを順次行えば、リスト 2.1 のようなプログラムができる。

宣言と初期化はリスト 2.1 の 4-6 行目に、参照は 8-10 行目に当たる。

リスト 2.1: 変数の出力

```
1 #include<stdio.h>
2
3 int main(void){
4     int n = 5;
5     const double pi = 3.14;
6     char c1 = 'p', c2 = 'i';
7
8     printf( "n:%d\n", n );
9     printf( "%c%c=", c1, c2 );
10    printf( "%f\n", pi );
11
12    return 0;
13 }
```

新出文法がまだまだ多いので、ソースコードを追って順番に説明していくこととする。

【変数の利用】

まず、リスト 2.1 の 4-6 行目、宣言と初期化についてみていくことにしよう。

変数の宣言と初期化

変数の宣言と初期化は

(型名) (識別子)=(初期値);

によって行われる。

実際は、

(型名) (識別子);

が宣言であり、これにイコールと初期値を付与することで初期化を行っている。たとえば、この 4,6 行目の意味を記せば

1.4 int 型 (整数型) の変数 (箱) である n を使うことを宣言し、n を 5 で初期化する。

1.6 char 型 (文字型) の変数 (箱) である c1,c2 を使うことを宣言し、c1 を文字 p で、c2 を文字 i で初期化する。

となる。

5 行目についている **const** 修飾子は Read only を表す修飾子で、その変数が読み込み専用で用いられることを示す。これは、後述する文字列リテラルなどの際によく用いられるが、通常の変数に用いる場合は定数の宣言であると思って差し支えない。例えばここでは円周率を定義しているが、円周率を他で書き換えてしまうと計算結果がおかしなことになりかねないため、この部分に集約し、容易に書き換えられないようにしている。

const 修飾子を用いた宣言の際には、宣言の後に=(初期値)を記し、必ず初期化しなければならない。一方で、通常の変数については、宣言と同時に初期化をしなくても構わない。例えば、先のリスト 2.1 のソースの 4-6 行目は、次のように書き換えても支障ない。

```
4  int n;  
5  const double pi = 3.14;  
6  char c1 , c2;  
7  n=5;  
8  c1='p';  
9  c2='i';
```

この時の初期化は、厳密に言えば代入であるが、このような宣言後最初の代入は初期化であると言える。変数を用いる場合、その変数に最初何が入っているかは特別な場合を除いてわからないので、必ず初期化 (代入や入力受け取り等の、初期化に代わる処理を含む) せねばならない。

【書式指定出力】

続いて、8-10 行目である。ここに出てくる、printf 関数は書式を指定した文字列 (書式文字列) を標準出力 (コンソール) に出力する関数である。第 1 引数に書式文字列を指定し、

以後の可変引数に式の引数リスト (変数名、定数、 $2*a$ などの式) を与えることで、その式の値を出力する。例えば、9 行目では、`%c` が文字型出力書式指定子であるので、それに対応した文字が出力される。1 つ目の `%c` に対しては 1 つ目の可変引数である `c1` が、2 つ目の `%c` に対しては 2 つ目の可変引数である `c2` が対応するので、`pi=` という出力になる (= はそのまま出力される)。

さて、書式文字列とは一体何なのかについて説明していこう。書式文字列は、特定の書式を用いて変数などを出力できるようにした文字列であり、表 2.3 に示す書式指定子を含む。これらに対して、先に説明したように可変引数に適当な変数を取れば、その値が出力されるのが `printf` 関数である。なお、書式指定子は `printf` 関数以外にも、後述する `scanf` 関数などでも用いられる。

表 2.3: 書式指定子一覧

書式指定子	データ型
<code>%d</code>	int 型
<code>%u</code>	unsigned int 型
<code>%o</code>	int 型, unsigned int 型 (8 進数)
<code>%x, %X</code>	int 型, unsigned int 型 (16 進数/X:大文字, x:小文字出力)
<code>%h</code>	(後ろに d/u/o/x を伴い)short 型
<code>%ll</code>	(後ろに d/u/o/x を伴い)long long 型
<code>%f</code>	float 型, double 型
<code>%lf</code>	double 型 [入力時のみ]
<code>%e</code>	double 型 (指数表示) [出力時のみ]
<code>%g</code>	double 型 (<code>%f</code> と <code>%e</code> を自動選択)
<code>%L</code>	(後ろに f/e/g を伴い)long double 型
<code>%c</code>	char 型 (文字)
<code>%s</code>	char *型 (文字列)
<code>%p</code>	ポインタ全般

それでは、書式指定子について簡単に解説を行っておく。概ね必要なことは表に書いておいたが、補足すべき点や、オプションについて述べる。

後ろに何かを伴う書式指定子は、`%llu` や `%Le` のように使い、その当該の型を後ろにつけた指定の書式で出力する。この例でいうと、`%llu` は unsigned long long 型の出力であり、`%Le` は long double 型の指数出力である。但し、元々 `%lf` は後に記す `scanf` 系関数の際のみ用いるもので、`printf` 関数では使わないことになっていた⁹。

整数の桁数を調整して出力したい場合がある。このようなときには、`%3d` のように、`%` の後に桁数を入れれば良い。さらに、3 桁でゼロうめをしたい場合 `%03d` のように、桁指定の前に 0 をつければ良い。同様に、小数についても、`%3.4f` とすれば、整数桁 3 桁、小数桁 4 桁での出力になる。整数桁の桁数字を省けば、小数桁のみの指定ができる。

⁹元々は、出力に書式指定子 `%lf` を用いるのは誤りとされていた。しかし、入力と同じであったほうがわかりやすいこと、あまりに間違える人が多かったことなどから、各コンパイラが独自に `%lf` を使えるように拡張していった。これを受けて 1999 年の改訂により、出力時に `%lf` を使えるという仕様が実際に規格に追加された。そのため、現在では `%lf` を出力の書式指定子として利用しても問題なくなった。

書式指定子には%を用いるが、これだと記号の%を出力したい時と書式指定子の区別がつかない。そのため、記号の%を出力したい時には%%と書くことになっている。¹⁰

【リテラルの書き方】

変数の初期化について話したので、関連事項として数値リテラルについても解説する。リテラル (literal) とはプログラムのソースコード中で用いられる定数のことで、その種別に応じて「文字列リテラル」「数値リテラル」などがある。

文字列リテラルはダブルクォーテーションで囲んだ文字列のことであり、これについて、詳細は後の講で説明する。

数値リテラルには、指数表記、8進表記、16進表記など、いくつかの書き方がある。

まず指数表記であるが、 $1e3$ ($1e+3$ とも)や $0.2E-2$ のように数値 k の後に E/e を置き、その後に指数 m を置くことで $k \times 10^m$ を表すことができる。先の例で言えば、前者は $1 \times 10^3 = 1000$ に、後者は $0.2 \times 10^{-2} = 0.002$ になる。

8進表記は 034 のように数字の前に 0 を、16進表記は $0x3a$ のように $0x/0X$ をおくことで実現することができる。8進の例は28に、16進の例は58になる。これらの表記は通常整数に用いられる。だが、C99において、浮動小数点数についても16進表記が可能になった。16進指数表現で $a.b \times 2^c$ と表される数は $0xa.bpc$ のように、 $0x$ のあと $a.b$ を記し、その後に p をつけて c を書くことで実現することができる。

2.2.2 変数の入力と簡単な計算

前節の解説で変数の宣言と出力は理解できたことだろう。今度は、変数の入力と変数を用いた計算を行おう。

【変数と四則演算】

2つの整数が入力された時、和・差・積・商・剰余の演算を行うプログラムを作る。

【解法】

まず変数を入力した後、各計算を行ってその値を出力すれば良い。ここでは、計算結果を変数に保持せずに、直接出力させた。

入力の後計算して出力するプログラムを簡単に書くと、リスト 2.2 のようなソースになる。

¹⁰同じことが'や'、\ のエスケープシーケンスなどにも言える。

このソースをコンパイルした後、実行すると入力待ち状態になる。そこで、 a, b をスペース区切りで 5 4 などと入力すると、値が出力される。printf1 行に対して出力の 1 行が対応するので、各々の対応付けを確認しておこう。

リスト 2.2: 変数の四則演算

```
1 #include<stdio.h>
2
3 int main(void){
4     int a , b;
5
6     scanf( "%d %d" , &a , &b );
7
8     printf( "%d+%d=%d\n" , a , b , a + b );
9     printf( "%d-%d=%d\n" , a , b , a - b );
10    printf( "%d*%d=%d\n" , a , b , a * b );
11    printf( "%d/%d=%d\n" , a , b , a / b );
12    printf( "%d+%d=%f\n" , a , b , (float) a / b );
13    printf( "%dmod%d=%d\n" , a , b , a % b );
14
15    return 0;
16 }
```

必要な解説は入力と演算であろう。以下、それらについて説明する。

【scanf による入力】

6 行目の scanf 関数は書式指定文字列に応じた入力を行う関数である。読み方は printf と同様に、各書式指定子に可変引数が対応している。各変数の前に&(アンパサンド)が付いているが、これは現状、約束事と思って受け取ってもらえれば良い¹¹。この例の場合、空白文字区切りで a と b を入力してもらうということになる¹²。他に、例えばコンマ区切りにしたい場合

```
scanf( "%d,%d" , &a , &b );
```

のように書式指定文字列を書けば良い。

文字を入力する場合、例えば

```
scanf( "%d" , &a );
scanf( "%c" , &c );
```

¹¹実際にはアドレスを取得する演算子であり、scanf 関数は書式指定子に対応した型の値を標準入力ストリームより入力してもらって可変引数のアドレスから参照される変数に代入する関数である。詳細は後で丁寧に記す。

¹²実際には、空白・タブ・改行は自動的に区切り文字として検出されるので、空白を入れなくても大丈夫である。

のようにすると、cに区切り文字の改行が入力されてしまう。これは、改行の段階で区切りが判定され、入力(入力ストリーム)に改行が残ってしまうのが原因である。このような場合に役立つのが読み飛ばしを示す*である。書式指定子の%の後に*を入れることにより、読み飛ばしを行うことができる。したがって、先に書いたコード断片について、整数を読んだ後1文字読み込みたい場合

```
scanf( "%d%c" , &a );
scanf( "%c" , &c );
```

のように書きなおせば、意図通りに動作する。

scanf 関数についても、各種書式指定子へのオプションを利用できる。とりわけよく利用されるのは文字列の読み込みの時で、文字列の読み込みを 30 文字にしたい場合は、"%30s"という書式指定文字列を用いればよい¹³。

【簡単な計算と演算子】

リスト 2.2 では、いくつかの演算子を用いた。これらを含め以下の表 2.4 に、C 言語における基本的な演算子を示す。

表 2.4: 算術・代入演算子

記法	意味	種類
+	左側の値と右側の値を足す	2 項演算子
-	左側の値から右側の値を引く	2 項演算子
*	左側の値と右側の値をかける	2 項演算子
/	左側の値を右側の値で割る	2 項演算子
%	左側の値を右側の値で割った剰余をとる	2 項演算子
=	左辺値に右辺値を代入する	代入演算子
+=	左辺値に右辺値を加えて新たな左辺値とする	複合代入演算子
--	左辺値から右辺値を引いて新たな左辺値とする	複合代入演算子
*=	左辺値に右辺値を乗じて新たな左辺値とする	複合代入演算子
/=	左辺値を右辺値で割って新たな左辺値とする	複合代入演算子
%=	左辺値の右辺値による剰余を新たな左辺値とする	複合代入演算子
++	このついた変数に 1 を加える (インクリメント)	単項演算子
--	このついた変数に -1 を加える (デクリメント)	単項演算子
+	プラス符号	単項演算子
-	マイナス符号 (符号を反転する)	単項演算子
sizeof	変数やデータ型のサイズ (バイト) を取得する	単項演算子

以下、補足を記す。

¹³書籍あるいはサイトによっては、「scanf 関数ではバッファオーバーランを防げない」として、文字列の読み込みを scanf で行うべきでないとしていることがある。これについては後に記すが、ここで書いた書式指定子の技法を用いることで防ぐことができるため、見落としのある意見である。

演算子には評価順序というものがあり、通常、数学と同じ順番で評価される。例えば $3+2*4$ は $2*4$ が先に計算され 11 になる。式中で、数学における括弧は全て $()$ によって表現される。すなわち、 $(2+4)*((1+3)*2)$ は 48 になる。

演算子の種類であるが、**2項演算子**は、その名前のとおり2つの項を結合して値を計算する演算子である。

代入演算子は左辺への操作を示す。例えば、

```
a = 3;
a += 5;
```

というソースは、第1行が「aに3を代入する」となり、第2行が「aに5を加える」という意味になる。これらは演算の評価順序では通常最後になる。つまり、右辺の値がひと通り計算された後で適用される。

単項演算子は一つの項への作用を示すものである。これは少し説明が必要であるので、演算子別に見ていこう。

1. インクリメント/デクリメント

演算子を付した変数に1を加え/減じ、以下それを定数として扱うという演算子である。どういうことかというと、 $a++$ と書けば、aに1を加えるという意味であるが、これは定数になるので、 $(a++)--$ などとも書いても、正常に動作しない、ということである。これらの演算子は、その項の前後どちらにつけるかによって意味合いが変わる。基本的に、後ろにつけた場合は、ひと通りの評価が終わった後、最後に1を加える/減ずる処理になる。前につけた場合は、最初にこの演算子を評価して、その後他の評価を行うことになる。例えば、 $a=3, b=5$ の時、

```
c = (a++) + (b++) ;
```

は、先にcに $3+5(=8)$ が代入されたあと、a,bに1が加えられるが、

```
c = (++a) + (++b) ;
```

はa,bに1が加えられた後、cに $(3+1)+(5+1)(=10)$ が代入されることになる。

2. 符号を示すプラスとマイナス

単項演算子のプラスとマイナスは、数学における $-x$ や $+2$ の $+$ 、 $-$ と同じである。

3. sizeof 演算子¹⁴

sizeof 演算子は、`sizeof var`のように、変数の前におくことで、その変数の大きさ(バイト)を取得することができる演算子であり、後で解説する派生型の利用などの際に便利である。また、sizeof 演算子は後ろに括弧を伴い、その中に変数や型を入れることで、その変数/型の大きさ(バイト)を取得することができる。`sizeof(int)`とすればこれはint型の大きさ、ということになる。基本的には括弧を後ろに伴って用いるが、括弧を必ず伴わなければならないわけではなく、また、括弧の中に変数の型を入れても良い¹⁵点が他の演算子と違う。

¹⁴sizeof 演算子に関数と混同している初心者がいるが、演算子である。

¹⁵これが、sizeof が関数でないことを明確に示す特徴である。

【型変換】

リスト 2.2 の 11 行目と 12 行目は同じ計算結果を出力しているが、5 / 4 と入力したとすると、前者は 1 になり、後者は 1.25 になったはずである。これは変数の型が違うことによる。

通常、変数の演算は同じ型の中で行われる。すなわち、int 型同士の演算は int 型で為される。したがって、5/4 は (int 型の 5)/(int 型の 4) という計算になり、結果も int 型となる。このとき、整数型では、計算結果が通常絶対値の小さい側の整数に丸められる (したがって、-5/4 は -1 となる。)。

ところが、実際の計算では違う型同士の演算や、同じ型同士の演算でも違う型の結果が欲しい場合がある。このような場合に考慮するのが型変換 (type conversion) である。型変換には、プログラムが計算内容に応じて自動的に行う暗黙の型変換 (implicit type conversion) と、プログラマーが変換する型を明示して行うキャスト (type casting) がある。

暗黙の型変換は、例えば float の値に int の値を足した時などに、float が int の値をカバーするため、int の値を float に変換する等の処理である。違う型同士の計算を行う場合に、カバー範囲の大きい方に自動的に変換してくれる (但し、代入演算子の場合は、左辺の型に合わせて型変換を行う) のが暗黙の型変換である。整数同士や浮動小数点同士でも、float 型と double 型の計算の場合、float 型を double 型に変換するなどの例もある。

一方、キャストは、次のように行う。

キャスト

キャストは

(型名) 変換したい値や式

によって行われる。

この時 (型) は単項演算子として扱われ、sizeof などと同様の優先順位で評価される。

以上から、リスト 2.2 第 12 行で何が行われているかを解釈してみると

1. (float) a が評価され、a がこの式中において float 型として扱われる (キャストされた) ことになる。
2. 割り算が評価される。このとき、float 型と int 型の計算であるので、暗黙の型変換により float 型の計算結果が出る。

となる。

型変換も一つの演算であるので、演算に評価順位がある。暗黙の型変換を行う場合、必要などところで初めて行われるのであり、「式中に float と int を混在させておけば大丈夫だ」などと考えるはいけない¹⁶。

¹⁶ 試しに、7/3/2.0 と、7/2.0/3 の 2 種類を計算してみよう。前者は 1.0 になるが、後者は 1.1 以上になるはずである。

本講の要点

本講では、変数について学習した。

変数について

- 変数は、まず型と識別子を指定して宣言を行い、その後初期化を行って、後は変数名を書くことにより参照できる。
- 変数の名前 (識別子) には命名規則があり、それに従わねばならない
- 符号付き整数型は、通常2の補数表現によって表される。そのため、全てのビットが1であるような数は、int 型の場合-1 である。
- Cにおいて、小数を格納する型には浮動小数点型が採用されている。
- 演算の際には演算例外に注意しなければならない。
- 文字型は、内部的には整数を格納しており、入出力の際にその整数を文字コードに従って文字に変換して出力している。
- 文字を'(シングルクォーテーション)で囲むことにより、その文字の文字コードを得ることができる。

変数の入出力と演算

- 変数を宣言するのは、通常関数の冒頭で、型名の後に変数名を列挙することで行われる。
- 宣言の際に代入を行うことで変数を初期化することができる。これは、後のコード中において入力や代入で代替することもできる。
- const 修飾子は Read Only(読み取り専用)を示す型修飾子で、これを付して宣言された変数は、以降のコード中において書き換えることができない。
- scanf 関数の引数に指定する変数には、&マークを付ける。
- 演算子は数学と同様に使うことができ、その演算順序も数学と同様である。
- 型変換には、暗黙の型変換とキャストとがあり、どちらも一種の演算として扱われる。

第3講 変数の活用

第2講に引き続き、変数の扱い方を学んでいくことにする。前講では簡単な演算のみを扱ったが、今回はより多くの演算や理論を学び、変数を活用できるようにしよう。

3.1 計算の誤差

浮動小数点数演算には誤差が出ることが知られている。実際どのような感じか見てみよう。

【誤差の確認】

リスト 3.1 に示したプログラムの出力結果を予想し、また実際に実行して、どのような結果になるか確認せよ。

人間の頭で計算すれば a は 0.1, b は 100, c は 1000000.00000001 となるはずである。前回までの文法で学んだ内容を用いているので、特に読めない箇所もないと思う。だが、物は試しなので、必ず打ち込んで実行してみることはして、考えた通りの動作をするだろうか？しないとすれば、何が原因だろうか？

リスト 3.1: 様々な誤差

```
1 #include<stdio.h>
2
3 int main(void){
4     float a , b , c;
5
6     a = 0.1 / 3.0;
7     a *= 3.0;
8     b = 1000000.000001;
9     b -= 1000000;
10    b *= 1E7;
11    c = 1000000 + 1E-7;
12
13    printf( "a=%.8f\n" , a );
14    printf( "b=%.8f\n" , b );
15    printf( "c=%.8f\n" , c );
16    return 0;
17 }
```

筆者の環境で実行してみると

```
a=0.10000001
b=99.99959564
c=1000000.00000000
```

となってしまった。何れも、予想した結果とは違うことがわかる。これが誤差であり、場合によっては大きなバグに発展してしまうこともある厄介な問題である。

このように、数値の計算を行う場合にパソコンの性質その他によって起こる誤差を計算

誤差という。前講の計算例外が「パソコンの性質その他により起こる、計算できない値」であるのに対して、計算誤差は「計算することはできるが、実際に期待される値と食い違いが生じる」というものである。以下、各種の計算誤差について見ていこう。

3.1.1 丸め誤差

リスト 3.1 の変数 a の出力のずれを起こした誤差が丸め誤差 (round-off error) である。

丸め誤差について考える前に、100 円均一ショップなどで売っている普通の電卓を用いて、 $1/3 \times 3$ を計算してみよう。0.9999... となるだろう。これは、電卓の表示桁に限界があるため、それよりも下の値が切り捨てられて起こる現象である。

似たような現象として、やはり電卓で、ルートを繰り返しとった後、逆に同じ回数だけ 2 乗を繰り返ししても、元の数に戻らない場合がある¹。これも計算中にあふれた桁が捨てられることによって起こると考えられる。

パソコンは浮動小数点などを用いて様々な数を扱えるようにしているが、内部で使われているビットは有限であり、それ故、無限の桁を保持することができない。そのため、やはりあふれた桁が捨てられ (あるいは切り上げられ) 真の値との誤差が出てくる。これが丸め誤差である。具体的には、10 進数の 0.1 を 2 進数で表現すると無限小数 (循環小数) になるため、丸め誤差が生じている (パソコンでは内部的に 2 進数として保存されていることに注意!)。これが、3 での除算・乗算過程を経ることによって見える値になったのが今回の事例ということである。

3.1.2 桁落ち

リスト 3.1 の変数 b の出力のずれは桁落ち (cancellation of significant digits あるいは単に cancellation や cancellation error と) によるものである。

この例では、絶対値の近い二つの大きな数の引き算を行っている。浮動小数点の原理を考えると、計算結果の小数点以下の値の精度があまり良いとは思えない。ところが、引き算によって有効桁数が減少し、この「精度が良くない部分」が表に出ることになってしまった。こうして誤差が顕在化したものが桁落ちである (このソースでは見やすくなるように増幅させている)。単純には、絶対値の大きな 2 数があり、これらの値が近い時に、0 方向に近づく加減算を行うと有効桁数が減少すること、と言える。

桁落ちはその原理を考えれば、絶対値の近い 2 数の引き算を上手に避けることで回避できる。アルゴリズムの変更は勿論、混合計算の場合等、うまい変形で避けられることもある。例えば、二次方程式の解の計算の際など、 $\frac{-b+\sqrt{b^2-4ac}}{2a} = \frac{-2c}{b+\sqrt{b^2-4ac}}$ と変形することで桁落ちを避けられる。

¹手持ちの電卓で試した所、元の数 x を 2 として、16 回ルートをとった後、2 乗を 16 回繰り返すと、1.999997708 という値になった。

3.1.3 情報落ち

リスト 3.1 の変数 c に関する誤差は情報落ち (loss of trailing digits あるいは information loss) とよばれるものである。情報落ちはアンダーフローと似た原理で起こる誤差である。

絶対値の大きい数に対し、絶対値のごく小さい数の加減を行う。計算後の値を浮動小数点表現すると、足された部分が元の数に比べて小さすぎて、仮数部への代入時に無視されてしまうことがある。これが情報落ちである。この処理は、例えば総和を行う場合などにしやすい (特に級数和の計算に出る情報落ちを積み残しと呼ぶ。)。

原理から、情報落ちを回避するに法は絶対値の違いすぎる数の加減を避ければ良い。また、情報落ちの典型例である積み残しに対しては、誤差をうまく評価して適切に総和を計算する **Kahan の加算アルゴリズム** (Kahan summation algorithm) が知られている。

3.1.4 計算機イプシロン

情報落ちに関連して、計算機イプシロン (machine epsilon) についても紹介しておく。1 より大きい最小の浮動小数点数 s に対し、 $\varepsilon = s - 1$ を計算機イプシロンという²。

計算機イプシロンは環境及び型によって異なるが、limits.h にある値を用いることで調べられる。limits.h 等の利用については付録 (リファレンス) に譲る。

3.1.5 打ち切り誤差

リスト 3.1 のソースでは見られないが、打ち切り誤差 (truncation error) と呼ばれる誤差も存在する。これは、コンピュータで数学の計算を行う際の近似のために無視された部分による誤差である。例えば、級数の和を用いた計算を行うとき、実際に無限の項を足し続けるわけには行かないので、途中何処かでやめることとなる。このとき、無視された部分は当然誤差となって効いてくる。これが打ち切り誤差である。これはパソコンの性質というより、アルゴリズムに依存するものであり、アルゴリズムの解析を行うことでその大きさを評価できる場合が多い。

3.2 ビット演算

コンピュータの数値表現にはビットが用いられていると述べた。場合によっては、そのビットを直接扱うことができると便利であろう。整数型についてこれを実現したのが、ビットを直接扱うビット演算である (浮動小数点型には適用できない)。早速見てみよう。

² $1 + \varepsilon$ が 1 と見なされなくなる最小の ε と説明される場合もある (本書でも最初そう書いていた) が、丸め処理によって値が変わってしまうことがあるため、厳密にはこの説明は誤りである。

【IPv4 address の解析】

IP アドレスは、32bit からなり、それを 8 桁ずつに区切って 10 進数に直され表現される。このうち、上位何ビットかはそのアドレスがどのようなネットワークに属するかを示す「ネットワークアドレス」である。これが何ビットかは「サブネットマスク」と呼ばれる数値で表される。例えば、サブネットマスクが 12bit=255.240.0.0 であるようなネットワークは、上位 12 ビットがネットワークアドレスである。これを用いて、IP アドレスは、a.b.c.d/n の形で表される。a,b,c,d は 0 以上 256 未満の整数で、n(サブネットマスクのビット数) は 0 以上 32 以下の自然数である。この形式で入力される IP アドレスのネットワークアドレスを出力するプログラムを書く。

リスト 3.2: IPv4 からネットワークアドレスを求める

```
1 #include<stdio.h>
2
3 int main(void){
4     unsigned int ip;
5     unsigned int a,b,c,d,n;
6     unsigned int mask,tmp;
7
8     scanf("%u.%u.%u.%u/%u",&a,&b,&c,&d,&n);
9     ip=(a<<24)+(b<<16)+(c<<8)+d;
10    mask=(-1<<(32-n));
11    ip&=mask;
12    tmp=(1<<8)-1;
13    a=(ip>>24)&tmp;
14    b=(ip>>16)&tmp;
15    c=(ip>>8)&tmp;
16    d=ip&tmp;
17    printf("%u.%u.%u.%u\n",a,b,c,d);
18    return 0;
19 }
```

解法で少し難易度が高いと思われる発想もあるのだが、何はともあれビット処理について理解しないことには話が進まないなので、ビット処理について学んでいこう。

【各種のビット処理】

先に述べた整数のビット演算には、表 3.1 のような種類の演算がある。

各演算子は=を伴って代入演算子として用いることができる(リスト 3.2 の第 11 行等。)。なお、ビット演算子の優先順位は低く、掛け算や足し算を行う時は勿論、第 5 講で紹介する比較演算よりも後なので、順番には十分注意しなければならない。

それでは、各演算について見ていくことにする。以下、簡単のため、 $a = (1100)_2$, $b =$

表 3.1: ビット演算子

記法	意味	形式	種類
&	ビット毎論理積	$a \& b$	2項演算子
	ビット毎論理和	$a b$	2項演算子
^	ビット毎排他的論理和	$a \wedge b$	2項演算子
~	ビット毎否定	$\sim a$	単項演算子
<<	左ビットシフト	$a \ll b$	2項演算子
>>	右ビットシフト	$a \gg b$	2項演算子

$(1010)_2$ とする。

論理積は、**AND 演算**とも呼ばれ、各ビットについて、ともに1ならば1を、それ以外の時ならば0を返す。例えば、 $a \& b$ は $(1000)_2$ となる。

論理和は、**OR 演算**とも呼ばれ、各ビットについて双方が0でなければ1を、双方が0ならば0を返す。 $a | b$ は $(1110)_2$ である。

否定は各ビットを反転する演算で、**NOT 演算**とも呼ばれる。 $\sim a = (0011)_2, \sim b = (0101)_2$ である。

排他的論理和は **XOR 演算**³とも呼ばれ、各ビットを見比べ、双方が違うビットならば1を、双方が同じビットならば0を返す。 $a \wedge b = (0110)_2$ となる。

左ビットシフト、右ビットシフトは、2進数において指定した桁数だけ左/右にずらすことを言う。 a を左1ビットシフトすると $(11000)_2$ と、右1ビットシフトすると $(110)_2$ となる。なお、実際には、はみ出たビットは捨てられる。右ビットシフトには、符号ビットを含めて行われる論理シフトと、含めずに行われる算術シフトとがある。C言語では、右ビットシフトにどちらのシフトを採用するかは処理系定義である。

これらの演算を用いると、例えば 2^n などの計算を高速化できたり、コンピュータでよく用いられる2進数関連の計算をわかりやすく行うことができる。ビット演算は一般に四則演算よりも速いため、2倍するよりは1ビット左シフトする、偶数かどうか判定するためには剰余を利用するより1とAND演算する、といった等価な操作をビット演算によって記すことで、プログラムを高速化できることがある(ただし、最近是最適化の技術向上のため、必ずしもビット演算で書いたほうが速いとは限らない)。

【リスト 3.2 の解説】

ここまでの内容を踏まえて、リスト 3.2 が要求されたプログラムになっているかどうかを見ていこう。

l.9 においては、まず、32bit としてひとつの変数にまとめる処理を行っている。これは、後の処理を楽にする目的であり、この処理により、32bit の int 型変数一つで全て扱うことができるようになる。変数が順番に 8bit ずつ並ぶことがわかるだろう。

l.10 は mask の作成である。これは、上位 n ビットが1、それ以下のビットが0であるような数である。整数型の-1のビットは全て1であり、0は全て0であることから、意図

³XOR は eXclusive OR(排他的論理和) の略である。

的にオーバーフローを起こして適切なビットに調整している。式について検証すると、これが正しくマスクを生成することがわかるだろう。

l.11 で目的の処理を行い、これによってネットワークアドレスを取得することができる。後は、それを分けるため、tmp を作成し、順次分けていき、出力している。

このように、ビット処理は IPv4 アドレスの解釈などに利用できる。ビット処理は人間の視点からだと、使うことは稀のように思える機能であるが、コンピュータに根ざしている処理を行う上で非常に便利なものである。有効利用してほしい。

3.3 数学関数と複素数演算

パソコンは「計算機」であるから、様々な計算ができる。先までの演算の話とはうってかわって、今度は C 言語による数学計算について見ていこう。

3.3.1 数学関数の利用

まず、実数 (浮動小数点数) に関して、数学関数がどのように適用できるかを見ていくことにする。なお、この節には多数の新出関数があるが、その解説は付録 A 等を適宜参照されたい。

【対数関数の使用例】

視等級 m 等の恒星が地球より d (ly, 光年) 離れた場所にあるとき、その絶対等級 M は $M = m + 5 - 5 \log_{10} \frac{d}{3.26}$ によって与えられる。 m, d が入力されるとき、その絶対等級を出力するプログラムを作成する。

【解法】

単純な計算問題であるが、対数関数が必要になる。指数対数関数や三角関数といった、演算子では用意されていない数学演算は `math.h` というヘッダに入っている。

なお、このコンパイルにあたっては、`gcc filename -lm` と、コンパイル時に末尾に `-lm` オプションをつけなければならない。

リスト 3.3: 絶対等級の計算

```
1 #include <stdio.h>
2 #include <math.h>
3
4 int main(void){
5     double z,m,d;
6     scanf("%lf",&m);
7     scanf("%lf",&d);
8     z=m+5-5*log10(d/3.26);
9     printf("%f\n",z);
10    return 0;
11 }
```

【数学関数利用の上で】

数学関数は、リスト 3.3 に見られるように、`func(args)` 等の形式で書けば、その値を返してくれる。そのため、数学関数を利用する場合は、通常の数学の式を書くのと同様に `2*sin(x)` などと記せば良い。

数学関数は `math.h` に入っているため、利用する場合はこれをインクルードしなければならない。更に、`math.h` 内の関数を用いた C 言語のソースを `gcc` でコンパイルする場合、先に記したように `-lm` オプション⁴を末尾に⁵付さねばならない。

数学関数を利用するにあたっての注意点を記しておく。

math.h の関数の一般の注意点

- 大抵の関数は一般性を失わない範囲で適切に設定されているため、自作の関数のほうが速かったり誤差が小さかったりする場合は稀である。但し、扱うデータの特長性によっては、より適切にできる場合も少なくない。
- 浮動小数点数を扱う以上、丸め誤差を始めとして各種の計算誤差は避けられない。したがって、誤差に関する処理を施したり、最悪の場合、目的に応じて(速度等を落としてでも)自作のルーティンを用いたほうが良い場合もある。(さすがにそんなケースは稀なように思えるが)

また、個々の関数において、自作関数のほうが良い場合が(`math.h`に限らず)存在する。ここでは、数学関数における代表例を示そう。

⁴lm は link to `math.h` の略である。

⁵`gcc` のバージョンによっては、`gcc -lm source -o output` のように、中間に `-lm` オプションを付しても大丈夫な場合もある。ただ、必ず通るわけではないので、通常末尾にしておくほうが安全である。

この問題は、`gcc` の引数解析及びリンクとスタティックリンク・ダイナミックリンクの差異などから起こっているものである。以下、簡単に解説を書いておく(ここまでの内容を逸脱している部分もある)が、詳しく知りたい場合は「エキスパート C プログラミング」(P.Linden 著, 梅原系訳, ASCII, 1996)などを参照されたい。なお、以下がわからなくとも、末尾においてさえいれば実害はない。

スタティックリンクは、ライブラリそのものが実行ファイルに組み込まれる方法である。一方で、ダイナミックリンクは実行時に都度引いてくるリンク方法である。ダイナミックリンクの場合、実行速度に劣るが、サイズが小さい実行ファイルが生成される。数学関数のライブラリ `math.h`(リンク先の実体は `libm` というライブラリ)は、実行速度を優先する場合が多いため、スタティックライブラリとして提供されている。

スタティックリンクとダイナミックリンクの違いは、コンパイル時にも及ぶ。ダイナミックリンクの場合は、一旦メモリにライブラリを持ってきて、それをひと通り関連付ける。一方で、スタティックリンクは、ライブラリの検索が始まる段階で「未定義である」となっているシンボル(関数、マクロなど)に対してのみ、ライブラリのオブジェクトが関連付けられる。つまり、スタティックリンクの場合、リンクを行う前の段階で、ソース中にある全未定義シンボルを洗い出しておかないといけない。

ここで、`-lm` をソースファイルの前においた場合、前から順に引数解析を行うと、“`math.h` にリンクをする”→“ソース中を解釈する”となってしまう、未定義シンボルが残ってしまう。後ろに置けば、この問題は解決する。前に付して上手くいっているのは、引数解析の際にスタティックリンク処理を最後にまわすように考えられているか、あるいは `math.h` に対して自動でリンクが張られるように設定されているかの何れかである。これらの設定は前提にできるほどのものではないため、リンクオプションは末尾につけたほうが良いのである。

なお、この説明からわかるとおり、この問題は `math.h` に限った話ではなく、一般のライブラリにも起こりうるものである。本書で扱う中では、`math.h` 関連で最初に出てきた問題であるので、ここに記述した。

- pow 関数は実数用に作られているので、整数乗を計算する場合は、後に学ぶ文法と繰り返し自乗法を適切に用いて自作で関数を用意したほうが速い。
- fabs 関数は実数向きであり、整数向きの関数は stdlib.h に abs 関数あるいは labs 関数として用意されている。また、これらの関数は、後に学ぶ関数マクロを用いて定義し、型依存性をなくして用いる場合も多い。
- 正の数に対する floor 関数や、負の数に対する ceil 関数は、「浮動小数点数型を整数型にキャストした場合 0 方向に丸められる」という性質のため、整数型へのキャストによっても実装できる。したがって、結果を整数型に代入する場合は、キャストを用いることもある。

ここに述べた以外にも、速度やメモリなど、何らかに特化させる必要がある場合は、自作の関数に置き換えたほうが良い⁶場合もある点、注意しておこう。但し、一般の用途においては、通常はライブラリ関数のほうが良い。これは、ライブラリ関数が広い汎用性のもとでできる限り最適化されているためである。

なお、数学関数の実装は、当該関数を McLaurin 展開した級数の計算である場合が多い。そのため、計算にはある程度時間がかかるという事も知っておくと良い。

【数学関数の型】

数学関数の引数の型は、通常 double である。C 言語では float 型や char 型、short 型を用いても、内部的には double 型や int 型として扱われる場合が多いため、float に double 型の関数を用いても多分問題はない(暗黙の型変換が行われるため、精度が失われることもないと考えられる)。だが、long double を用いる場合には、問題が出てくる可能性もある。

そこで、各々の関数の名前の後ろに f や l を付し、名前の後ろに f を付した関数は float 型として、l を付した関数は long double 型として用いるようにした⁷。したがって、先のプログラムを long double に書き直したい場合は、log10l 関数を用いれば良い⁸。

数学関数に限らず、C 言語の浮動小数点演算は double 型が基本であるため、特別な理由がなければ、浮動小数点型には double 型を用いると良い。同様に、整数には int 型を、文字には char 型を用いるのが一般的である。

3.3.2 複素数演算と型総称数学関数

C 言語は 1999 年の改訂により、数学関連の機能が大幅に強化された。この強化された機能を見ていこう。

⁶なお、標準関数を自作の同名の関数で置き換え、再定義することを、インターポジショニング (interpositioning) と呼ぶ。これは実に強力であるが、意図しない場合に悲惨な結果をもたらすこともある危険な機能である。それゆえ、ここで言う置き換えも、同名ではなく、別名の関数で置き換えたほうが良い(つまり、標準関数との衝突を避けて、別名で定義しておくほうが良い)。このインターポジショニングという機能は、本書レベルでは必要ないが、「自作関数での置き換え」について述べたため、ここに記した。

⁷この機能ができたのは C99 であるため、一部のコンパイラでは機能しない場合がある。例えば、Visual Studio やバージョンが古い gcc でコンパイルできない場合がある。

⁸この「分けて書かないといけない」現象を解決するのがこのあと取り上げる tgmath.h である。

【ド・モアブルの定理の確認】

ド・モアブルの定理は、次のような定理である。

$$(\cos \theta + i \sin \theta)^n = \cos n\theta + i \sin n\theta$$

但し、 θ, n は実数、 $i = \sqrt{-1}$ とする。この成立を確認する。

リスト 3.4: ド・モアブルの定理

```
1 #include<stdio.h>
2 #include<math.h>
3 #include<complex.h>
4 #include<tgmath.h>
5
6 int main(void){
7     double theta,n;
8     double complex a,b;
9     scanf("%lf%lf",theta,n);
10    a=cos(theta)+I*sin(theta);
11    a=pow(a,n);
12    b=cos(n*theta)+I*sin(n*theta);
13    printf("a=%f+%fi\n",creal(a),cimag(a));
14    printf("b=%f+%fi\n",creal(b),cimag(b));
15    return 0;
16 }
```

ド・モアブルの定理は複素数に関する定理であるので、当然ながら複素数演算を利用している。このソースを読み解いていこう。

【複素数型と虚数型】

通常、複素数型の宣言には、`_Complex` 型を、虚数を用いる際には `_Imaginary` 型を用いる。だが、`complex.h` のインクルードによってこれらの型を `complex` ないし `imaginary` と記述できるようになる。この方がわかりやすいため、`complex.h` を用いて宣言するケースが多い。これを踏まえて、リスト 3.4 の 8 行目を見てみると、`complex` の前に `double` がついていて、よくわからない。

複素数型は、内部的には 2 つの浮動小数点数型により実現されている (虚数型は 1 つ)。先に学んだ通り、浮動小数点型には精度によって 3 つの型がある。これが複素数や虚数にも適用される。`double complex` は実部および虚部が各々 `double` の精度を持った複素数型変数であり、`long double imaginary` は実部が 0 で虚部が `long double` の精度を持つ純虚数を表す型である。このように、複素数/虚数の宣言の際には、その精度を指定しなければならない。

【複素数型/虚数型の型変換】

複素数型、虚数型も通常の浮動小数点型などと同じように扱うことができるが、型変換関連では注意する必要がある為、ここに述べる。なお、以下では区別のために、float や double を総称して実数型と呼ぶことにする。

複素数型は実数型、虚数型の双方よりも広い範囲を表す型である。このため、二項オペランド⁹の一方が複素数型である場合、その計算結果は複素数型になる(暗黙の型変換である)。なお、complex.h をインクルードしない場合、四則演算において演算例外が起きる場合があるので、インクルードすることを推奨する。

虚数型はやや複雑な型変換体系を持つ。虚数型同士の積/商は実数型になる。一方、実数型と虚数型の積/商は虚数型である。和・差については、虚数型同士の場合虚数型のままであるが、実数型と虚数型との間で行った場合、その結果は複素数型になる。具体的に、リスト 3.4 の 10 行目について、型変換の手順を見てみよう。なお、リスト 3.4 中に出てくる I は虚数単位 ($\sqrt{-1}$) を表す定数(定数マクロ)であり、虚数型である。

1. `I*sin(theta)` は虚数型と実数型の積なので、虚数型の結果になる。
2. `cos(theta)+I...` は、実数型と虚数型の和なので、複素数型の結果になる。

上記のように、暗黙の型変換は比較的数学に近い形で行われるようになってきているが、キャストはやや面倒である。表 3.2 に、実数型、虚数型、複素数型の各キャスト結果をまとめた。

表 3.2: 実数型・虚数型・複素数型の間のキャスト

キャスト元	キャスト先	結果
実数型	虚数型	値が 0 の虚数型
実数型	複素数型	実部が元の値、虚部が 0 の複素数型
虚数型	実数型	値が 0 の実数型 (<code>_Bool</code> のみ例外(後述))
虚数型	複素数型	実部が 0、虚部が元の値の複素数型
複素数型	実数型	元の数の実部を値とする実数型
複素数型	虚数型	元の数の虚部を値とする虚数型

【複素数関数】

複素数の演算にも、それ専用の関数があり、complex.h に収められている。このヘッダの詳細は付録を見てもらえばよいが、以下のような法則だけを紹介しておく。

⁹オペランド (operand) はプログラミングにおいて演算の対象となる値や変数のことを指す。和訳される場合は通常被演算子とされる。これは、演算子 (operator) に対して、その効果を受けるという意味からの訳である。

complex.h 内の関数の命名法則

- 実数型に同様の機能の関数がある場合 (sin 等)、その複素数型のものは関数名の先頭に c をつけたもの (csin 等) になる。
- 複素数型関数にも精度による別があり、数学関数と同様の命名規則になっている (csinf, csin, csinl 等)。

【型総称数学関数ヘッダ】

ここまでで、実数にも複素数にも同じ機能のある関数は、その型に応じて 6 種類を使い分ける必要があることがわかる (例えば、正弦関数の場合、実数型の `sinf`, `sin`, `sinl` があり、複素数型の `csinf`, `csin`, `csinl` がある)。だが、これらをわざわざ分けて書くのは、読みづらい上に書く側としても面倒である。そこで、こういった例を統一的に扱うことができるように、Java や C++ では同名ながら引数の型/個数が違う関数を定義できるようにした。これを多重定義ないしオーバーロード (overloading) と呼ぶ。だが、残念ながら、C にはこの機能が存在しない。

とはいえ、マクロ/プリプロセッサ文を上手く使うことで、C 言語のソースでも似たような機能を実現できる (また、C11 ではそのような機能が充実した)。C99 まででは、言語としてのオーバーロード機能はないものの、コンパイラの機能に頼ることでそれに近い効果を得られる。C 言語の標準関数のうち、ここで紹介した `math.h` および、次に紹介する `complex.h` については、`tgmath.h`¹⁰ でこの機能が提供されているため、これをインクルードすればいちいち使い分けなくてすむようになる。

例えば、リスト 3.4 の 11 行目であるが、これは引数・返却値とも `double complex` 型であるので、本来は `cpow` 関数であるはずである。だが、ここでは `pow` と書いて通用している。これが `tgmath.h` の効用である。

`tgmath.h` を利用する際に置き換える名前は、`math.h` にも `complex.h` にも共通する機能の関数ないし、`math.h` のみの関数である場合は、`math.h` の `double` 型関数の形で書く (`sin`, `pow` 等)。`complex.h` にしか関わらない関数は、c をつけた名称で書く (`creal`, `cimag` 等)。

ここまで長々と書いてきた数学関連の関数の機能をまとめておく。

数学関連機能のまとめ

- `math.h` を用いている際には `-lm` オプションを末尾につけてコンパイルすること。
- 複素数演算を用いる際には `complex.h` をインクルードすること。
- 必要に応じて `tgmath.h` をインクルードし、可読性を向上させること。極論、`math.h`、`complex.h`、`tgmath.h` を常にセットでインクルードしても良い。

¹⁰ これは、Type-generic math の略である。

本講の要点

本講では、前講で学んだ変数を活用し、接続構造で更に多くの計算処理を行う手法を学んだ。

計算誤差

- コンピュータで浮動小数点数の演算を行う場合、その内部表現やアルゴリズムなど故に、様々な計算誤差が出る。
- 丸め誤差は不可避であるが、桁落ちや情報落ちはその原因となる計算をなくすことで避けることができる。
- 打ち切り誤差はアルゴリズムに依存する誤差で、アルゴリズムからある程度見積もることができ、アルゴリズムの改良によって小さくすることができる。
- 計算機イプシロンとは1以上の最小の浮動小数点数から1を引いた値のことである。

ビット演算

- ビット演算は整数型のみに適用される演算で、AND,OR,XOR,NOT, ビットシフトなどの演算がある。
- ビット演算は通常の四則演算よりも負荷が少ないため、等価な四則演算をビット処理に置き換えることで、プログラムの高速化が期待できる。

数学関連機能

- 数学的な演算を行う関数は `math.h` に収められている。
- `math.h` を用いている際には `-lm` オプションを末尾につけてコンパイルする。
- 複素数演算を用いる際には `complex.h` をインクルードする。
- `math.h`、`complex.h`、`tgmath.h` を常にセットでインクルードしておくと、似たような機能の関数を型に応じて書き分ける必要がなくなり、読みやすいソースになる。
- これらのヘッダで定義される関数は `func(args)` 等の形式で呼び出すことで、その計算結果をそのまま値として返却するので、式中に直接書けば良い。

第4講 分岐構造

構造化定理において挙げられる3構造のうち、分岐構造について学ぶ。

4.1 分岐構造とは

実際に使い方に入る前に、分岐構造とは何なのか、説明しておこう。

分岐構造というのは、何らかの条件を与え、それによって処理を変えるという構造である。例えば、RPGにおいて、「はい」「いいえ」の選択肢がある時、選択肢に応じて会話が変わる。平方根を計算するプログラムを組む時、もし、与えられる数が負の数だったらエラーを出力する。このように「条件の真偽に応じて行う処理を変える」のが分岐構造の役目である。

4.2 if文とelse文

4.2.1 if文

まず、最も単純な分岐を考えてみよう。

【0 割回避機能付き四則演算】

二つの整数の四則演算を行うプログラムを作成する。この時、割り算については、除数が0でない場合のみ実行するようにする。

【解説】

四則演算そのものは前回までに習ったものと同じである。これに、もしも2つ目の数が0でないならば割り算を計算して出力するという命令を書き加える。

リスト 4.1: 0 割回避四則演算

```
1 #include<stdio.h>
2
3 int main(void){
4     int a,b;
5     scanf("%d%d",&a,&b);
6     printf("Sum:%d\n",a+b);
7     printf("Dif:%d\n",a-b);
8     printf("Prod:%d\n",a*b);
9     if(b!=0){
10         printf("Div:%d\n",a/b);
11     }
12     return 0;
13 }
```

【if文の使い方】

今回目新しいのは、1.9のif文であろう。これは、次のような形式で用いられる。

if文

if文は

```
if(条件式){
    処理
}
```

の形式で記し、条件式が真の場合に中括弧内の処理が行われる。

なお、処理が一文の場合

```
if(条件式) 処理;
```

の形でも書ける。(但し、書き直しの際に中括弧を入れる必要が出てくる場合も多いので、多用はされない。)

if文を用いることにより、単純な分岐を作ることができる。また、if文の中に更にif文を入れることもできる。このようにあるブロック内に更に別のブロックを入れることをネスト(nest)あるいは入れ子と呼ぶ。後に習う各ブロック(if,else,switch,for,while,doなど)は任意に組み合わせてネスト構造にすることができる。

【条件式とその値】

if文の引数は条件式である。この条件式は、表4.1に示す比較演算子あるいは関係演算子(relational operator)を用いて書くことができる。

表 4.1: 比較演算子の一覧

記号	意味
==	左辺と右辺が等しい
!=	左辺と右辺が等しくない
<	左辺が右辺より小さい
<=	左辺が右辺以下である
>	左辺が右辺より大きい
>=	左辺が右辺以上である

これらによって比較が行われた時、条件式が真であれば1、偽であれば0と評価され、その値が利用できる。例えば、 $((a==1)+(b==-1))==2$ などすると、これは、aが1でbが-1の時だけ真になる条件式になる。この条件式の値を利用すれば、ここまでの知識だけでも様々な条件を書くことができる。

逆に、一般の式の値を真/偽の値としても利用することができる。一般の式の値について、0は偽を示し、それ以外の値(非0)は真を示す。例えば、

```
if(n%2){  
    処理  
}
```

とすると、この処理は n が奇数の場合のみに処理が行われることになる。一般に、条件式のあるべきところに単なる値が書かれているときには、後ろに $!=0$ を補えば良い。このことから、 $\text{if}(1)$ は必ず実行されるし、 $\text{if}(0)$ は実行されないとわかる。

条件式の値は if 文だけにとどまらない重要事項であるので、以下にまとめておく。

条件式と値

- 条件式は真の時に 1 と、偽の時に 0 と評価される。
- 一般の式が条件式として用いられた場合、その値が非 0 の場合は真として、0 の場合は偽として扱われる。

最後に、比較演算子の注意点を記しておく。

比較演算子の注意点

- 比較演算子は二項演算であるので、左辺と右辺の比較しかできない。つまり、 $a==b==c$ や $a<=b<c$ のような書き方は思ったのと違う評価になる^a。
- 等しいことを示す $==$ は必ず二つ書かねばならない。これを $=$ とするバグは非常に多く、またコンパイルエラーにならないことが多いため、見つけづらい^b。
- $<=$ と $>=$ 、 $!=$ は $=$ を後ろ側に書かなければならない。「小なりイコール」などと読む順番に書くと覚えれば良い。
- 比較演算子はビットシフトより遅く評価されるが、他のビット演算とは同等で、左側から評価される。

^aこのような論理式を実現するには、後述の論理演算子を用いるか、if 文のネストを組めば良い。

^bこれを回避する方法として、変数 $==$ 値と書いているものを、逆に値 $==$ 変数と書けば良いと言われることもある。だが、ソースが見づらくなる割に、特定の場合にしか効果を発揮せず、推奨しがたい。似非イディオムと言われることさえあるので、極力使わないほうが良い。

4.2.2 else 文

ここまでは単なる「〇〇の時に～する」形の分岐だけを扱ったが、「そうでなければ～する」という分岐を加えることができる。例えば、「リストに登録されたメールアドレスであれば受信し、それ以外であれば遮断する」の「それ以外」に当たる部分である。

ここでは、偶数か奇数かを判別して出力するプログラムを考えてみよう。

【偶奇の判定】

入力される自然数が偶数か奇数かを判定するプログラムを作成する。

【解説】

単純な条件式は $a\%2$ の形であるが、2 による剰余は 2 進法の一番下の位が 0 か 1 かを判定することで実装できる。そのため、ここでは $a\&1$ という形で実装してみた (条件式の値を思い出そう)。

リスト 4.2: 偶奇の判定

```
1 #include<stdio.h>
2
3 int main(void){
4     unsigned int a;
5     scanf("%u",&a);
6     if(a&1){
7         puts("Odd_number.");
8     }else{
9         puts("Even_number.");
10    }
11    return 0;
12 }
```

【else 文】

ここで新しく出てきた else 文こそが、「そうでなければ～～する」を実行するための文である。

else 文

else 文は、前述の if に続けて

```
if(条件式){
    処理
}else{
    処理
}
```

の形で書き、対応する if 節の条件式が偽である場合のみに実行される。なお、対応する if 節とは、同じ階層 (ネストの深さが同じ) である if のうち、直前の if を指す。例えば、

```
if(...){ //if-1
    if(...) a=b; //if-2
    else b=a; //if-2 に対応
}else(...){ //if-1 に対応
    ...
}
```

といった具合である。

この else 文を用いると「A ならば〇〇、B ならば～～する」と書けるが、これにとどま

らず、else 文は更にその後ろに if 節を取ることができる。

else if 文

else の直後に if を記し、

```
if(条件式 1){
    処理 1
}else if(条件式 2){
    処理 2
    :
}else if(条件式 n){
    処理 n
}else{
    処理 else
}
```

のような形で書けば、「条件式 k-1 までに合致せず条件式 k に合致する」場合に、処理 k が行われる。

これにより、多岐分岐を実装することができる。多岐分岐には後述の switch～case が使われることもあるが、こちらのほうが活用される範囲が広いので、有効活用されたい。

なお、else 文は if 文に対するオプションのようなものであり、if に対して必ず対応する else を書かなければならないというわけではない。したがって、else if を連続した末尾に else 節が必ず来るとは限らない。逆に、else には必ずそれに対応する if 節が必要である。

4.3 論理演算子

ここまでで単一の条件による判定と、条件式による判定を学んできた。だが、 a, b, c が全て 1 であると表現するのに、 $(a==1)+(b==1)+(c==1)==3$ のような式で書くのは見づらい。このような複数条件の条件式に用いるのが論理演算子 (Logical operator) である。

【閏年の判定】

入力される年が閏年かどうかを判定するプログラムを作成する。

【解説】

グレゴリオ暦で考える場合、閏年は 4 で割りきれて 100 で割り切れないか、あるいは 400 で割り切れる年である。

リスト 4.3: 閏年の判定

```

1 #include<stdio.h>
2
3 int main(void){
4     unsigned int y;
5     scanf("%u",&y);
6     if(y%4==0 && y%100!=0 || !(y%400)){
7         puts("Leap_year");
8     }else{
9         puts("Not_Leap_year");
10    }
11    return 0;
12 }

```

【論理演算子の意味】

なんと言っても条件式に出てくる、&&、||、!であろう。これは、表 4.2 のような意味を持つ。これらによって条件式を複数組み合わせることができる (複合条件文)。

表 4.2: 論理演算子

記号	意味
&&	かつ (連言)
	または (選言)
!	でない (否定)

これらの演算子はビット演算と似ているが、ビット演算はビットごとに演算するのに対し、論理演算は論理値のみに対して演算を行う点が違う。論理値は先に述べたとおり、非 0 を真、0 を偽として扱い、逆に条件式から論理値が値として出る場合は真が 1、偽が 0 となる。

論理演算子とその値

- (条件式 A) && (条件式 B) は、条件式 A, B とも真 (非 0) である場合には真 (1) を返し、それ以外は偽 (0) を返す。
- (条件式 A) || (条件式 B) は、条件式 A, B の一方が真 (非 0) である場合には真 (1) を返し、両方が偽 (0) の場合は偽 (0) を返す。
- !(条件式) は、条件式が真 (非 0) の場合には偽 (0) を返し、偽 (0) の場合には真 (1) を返す。

【論理演算子の計算順序と評価】

論理演算子の評価順序および評価の仕組みについて説明する。

論理演算子の評価順序

- !演算子は単項演算子として扱われ、評価順序も他の単項演算子と同じである。
- &&演算子は比較演算、ビット演算の後に評価される。但し、結合前後の論理値によって評価に影響を与える。
- ||演算子は、&&演算の後に評価される。但し、結合前後の論理値によって評価に影響を与える。
- どの論理演算子も、左側にあるものを先に評価する。

この説明のうち、気になるのが「結合前後の論理値によって評価に影響を与える。」という文面であろう。これについて、&&を例にとって説明する。

&&演算子について、例えば、`(a==5)&&(b%2)` という式があったとしよう。これは左側から展開されるので、まず、`(a==5)` が評価される。そして、仮にこれが偽の場合、&&演算子の値はその段階で定まってしまうので、以降の `(b%2)` が評価されない。

このように、論理演算子は、その複合条件文の値が定まった場合、その段階で評価を打ち切ってしまう。この性質は、有効利用すればプログラムの高速化につながる反面、例えば `(a==2)&&(b++)` などと式を書いた場合、`a` が 2 でなければ `b` のインクリメントが行われないために意図した結果と違う結果をもたらす場合もある¹。評価順序は原則的に上に記したとおりだが、論理演算を行う場合はこの打ち切り処理に注意しなければならない。慣れてくるまではこの処理を有効利用して高速化するなどは考えず (実際、体感できるほど速くなることは、初心者のうちはないと言って良い)、後に影響を残す可能性のある演算 (代入演算、インクリメント/デクリメント) を複合論理式中に組み込まないように注意を払ったほうが良い。

なお、論理式が 1 または 0 で評価されることを考えると、ビット演算子でも似たようなことができると思うだろう。だが、ビット演算には、ここで書いた値決定時の評価打ち切りがないという違いがある。XOR 等を使えるなどの利点もあるので、使うべきではないとは言わないが、使う際には注意されたい。

4.4 switch～case 文

ここまで、条件式を用いた条件判定を行って来たが、C 言語にはもうひとつ、値の列挙による条件判定機能がある。値の列挙とは、現実世界でいうところの「唐揚げ弁当ならば 500 円、トンカツ弁当ならば 550 円、エビフライ弁当は 580 円…」のような、多数の選択肢から 1 つを選ぶ場合どうするか、というものである。C 言語の場合は、この値の列挙に

¹—方で、コードの短さを競う競技である“Code Golf”では、この性質を用いた書き換えを行うのが定石になっている。これは、`if(a==0) b=1;` などの文について、`a||b=1;` などとして、文字数をカットできるためである。勿論この書き方は、“Code Golf”競技以外では通常使わない。

は「それ以外」が加わり、例えば「定食のご飯が白飯なら 500 円、玄米なら 520 円、五穀米なら 550 円、それ以外なら 580 円」といった形の選択になる。この実装を見ていこう。

【計算式を打ち込む四則演算】

整数と演算子 (+, -, *, /) からなる式を打ち込んでもらい、その値を計算するプログラムを作成する。

【解説】

- 演算子の判別が必要になる。演算子は範囲などでは書けないので、ここではじめて、先に書いた「値の列挙」、ここでは文字の列挙による判定が行われる。
- データの入力は、整数, 文字, 整数なので、scanf の書式文字列を "%d%c%d" としてやれば良い。
- 途中の else 節 (l.20-22) の中にあ
る `return 0;` は main 関数の終了を示し、仮にこの else のところの処理が行われることになったら、ここで main 関数の処理が打ち切られるということを示す。このように、main 関数の途中に `return` をおいて main 関数を終了させる手法はよく用いられる。(特に、0 以外の値を指定して、異常終了という形にすることが多い。)

リスト 4.4: 計算式を打ち込む四則演算

```
1 #include<stdio.h>
2
3 int main(void){
4     int a,b,ans,flg=0;
5     char c;
6     scanf("%d%c%d",&a,&c,&b);
7     switch(c){
8         case '+':
9             ans=a+b;
10            break;
11        case '-':
12            ans=a-b;
13            break;
14        case '*':
15            ans=a*b;
16            break;
17        case '/':
18            if(b!=0){
19                ans=a/b;
20            }else{
21                return 0;
22            }
23            break;
24        default:
25            flg=1;
26            break;
27    }
28    if(flg){
29        puts("error");
30    }else{
31        printf("%d\n",ans);
32    }
33    return 0;
34 }
```

【switch case による分岐】

今回用いた値の分岐は、switch～case 文による。この switch～case 文は先に書いたとおり、定数値による分岐を行う。公式の前に、流れを解説しておく。

switch 部分に来ると、switch の引数の式を評価し、それと同じ定数式をブロック中の case から探し求める。見つかったならば、該当する case まで飛び、見つからなければ default

まで飛んで、その後の処理を実行する²。途中で `break` という命令が見つかった場合は、その段階で `switch` を抜ける。なお、この `case` 文を多数書くとコードが長くなるが、一応 256 個までであればコンパイルを通ることが保証されている³。

通常、`switch`～`case` 文は `break` を用いて、次のように書く。

`switch`～`case` 文

`switch`～`case` 文は

```
switch(判定値){
    case 定数式:
        処理
        break;
    :
    default:
        処理
        break;
}
```

の形で書く。

`switch`～`case` において `default` は省略してもよい。省略して `case` の中で該当する値が見つからなかった場合は何も処理が実行されないことになる。また、同様に `default` の終端の `break` も省略することができるが、ラベルを書き換えたときに忘れがちなので、書くようにしている人も多い。

なお、`switch`～`case` もネストすることはできるが、ひどく見づらいため、めったに使わない。`switch`～`case` の中で更に条件分岐が必要な場合は、`if` を使うほうが良いだろう。

【`break` なしの `switch`～`case`】

続いて、`break` を使わない `switch`～`case` について見ていこう。

`break` を書かないと、その後の処理が引き続き行われる。例えば、

```
switch(a){
    case 1:
        puts("a=1");
    case 2:
        puts("a=2");
```

²実は、この `case` ないし `default` というのは、ラベルという文である。ラベルは、ラベル名: という形で書き、次講で解説する `goto` 文 (構造化プログラミングの提唱論文において、流れを見難くするので乱用すべきでないとされた命令) の引数にとることができる。`switch` 内のラベルは `case` を用いるか `default` を用いるかしかなく、これに `goto` を使ってアクセスすると流れが見づらいスパゲッティソースコードがあつという間にでき上がる。これは可読性を著しく下げるので、`switch` 中に飛ぶような `goto` を作ってはならない。この理由のため、一部の書籍では、`switch` そのものを用いるべきでないとするものまである。

³256 個というのは、文字コードに応じた文字を返す処理を作るときに必要なからである。通常、そんなに沢山の `case` を使うことはない。なお、処理系によっては 256 個より多くの `case` 文が許される場合もある。

```

    default:
        puts("a>=3 or a<=0");
}

```

というコードがあった時、もしも a が 1 だったならば、出力は

```

a=1
a=2
a>=3 or a<=0

```

となる。

これは、break を忘れた場合に処理がおかしくなるということであるが、逆に利用することもできる。最後に、その例を紹介しておく。

【月の日数】

月が入力される時、その月の日数を出
力する。

【解説】

ここでは、break なしの case 文を一部
(4月、6月、9月の場合の処理) に用い
て、処理を簡単化した。break なしの場
合、そのまま下にうつって 30days. と
出力される。

なお、閏年の処理は施しておらず、1 か
ら 12 以外の数が入れた時の処理
も行っていない。これらについては、
switch の前後に if 文を設ければ、比較
的に簡単に処理ができる。

リスト 4.5: 月の日数

```

1 #include<stdio.h>
2
3 int main(void){
4     short month;
5     scanf("%hd",&month);
6     switch(month){
7         case 4:
8         case 6:
9         case 9:
10        case 11:
11            puts("30days.");
12            break;
13        case 2:
14            puts("28days.");
15            break;
16        default:
17            puts("31days.");
18            break;
19    }
20    return 0;
21 }

```

なお、break なし switch 構造では、default ラベルが switch 文の末尾にこないこともある。

4.5 条件演算子

「ある条件の時にはこの計算式を用い、そうでない場合にはこの計算式を用いる」のよ
うな、ごく単純な (計算だけの) 場合分けをいちいち if 文, else 文で書いていると非常に面
倒である。そこで、式中で使えるような分岐を作る演算子が用意されている。この分岐を

行う演算子が条件演算子 (conditional operator) である。この演算子は、C 言語で唯一の三項演算子であり、C 言語中ではしばしば三項演算子と条件演算子が同じ意味で用いられる。では、二数の比較を例に、場合分けを見てみよう。

【二数の比較】

入力される二つの整数のうち、大きい側の値を出力する。

【解説】

if 文を用いる代わりに、条件演算子を用いて作成してみる。なお、条件演算子は、後に学ぶマクロと相性がよく、主にマクロの定義に用いられる。

リスト 4.6: 二数の比較

```
1 #include<stdio.h>
2
3 int main(void){
4     int m,n,max;
5     scanf("%d%d",&m,&n);
6     max=(m>n)?m:n;
7     printf("%d\n",max);
8     return 0;
9 }
```

このように、条件演算子は、?と:からなり以下の形式で用いる。

条件演算子の形式

(条件式)?(真の場合の式):(偽の場合の式)

この時、条件式の値に応じて真の場合/偽の場合のいずれか一方が評価される。この、式を評価する性質のため、リスト 4.6 のように、代入の右辺値等として使われる。

4.6 _Bool 型の利用

C99 では条件式の値を保存するための型として、_Bool 型が新たに作られた。これは、条件値の保持や返却に有効である。以下に、その特徴を示す。

_Bool 型の特徴

- _Bool 型は符号なし型として扱われ、signed, unsigned はつけられない。
- _Bool 型は 0 と 1 を格納できれば十分なサイズとして定義されている。
- _Bool 型に型変換する際は、元の値が 0 なら 0、非 0 なら 1 に変換される。
- _Bool 型は最もサイズの小さい整数型として扱われ、他の型との演算の際には優先度が一番低い。
- _Bool 型は整数型であるのでビット演算を行うことができる。とりわけ、XOR を表現するのに利便性が高い。

【_Bool 型を扱うヘッダ】

この _Bool 型を利用するために、C99 において `stdbool.h`⁴ というヘッダが追加された。実際に _Bool 型を利用する際には、このヘッダを使う場合が多いだろう。以下、利用上の手引きを示す。

stdbool.h の利用方法

- このヘッダは C99 から導入されたヘッダであるので、対応していないコンパイラでは利用できない。
- このヘッダをインクルードした場合、型名を `_Bool` と書かず `bool` と書いて良いようになる。(complex.h をインクルードした時の `complex` や `imaginary` と同じことである)
- このヘッダをインクルードした場合、`true/false` を用いて真値 (1)/偽値 (0) を表現することができるようになる。プログラムを書く際に、条件式の値をあてにする場合、この `true/false` を利用するとソースが読み易くなる。
- (参考) このヘッダには、`__bool_true_false_are_defined` というマクロが定義されている。このマクロを利用することにより、後に解説する列挙型などを用いて _Bool 型と同等の機能を定義した場合の判別ができるようになっている。

これらとビット演算子ないし論理演算子を組み合わせることで、論理代数 (ブール代数) の計算を行うことができる。条件式の値に関する法則と共に有効活用すると良い。

⁴STanDard BOOL から。

本講の要点

ここまで、条件分岐について学んできた。以下、解説の流れとは別でまとめた。

条件式

- 条件式は比較演算子や論理演算子を用いて書くことができる。
- 論理演算子は式の値が決まった段階で評価を打ち切る。
- 条件式は真値として1を、偽値として0を取る。これらのための型として `_Bool` 型が、それに対応したヘッダとして `stdbool.h` が用意されている。
- 一般の式を条件として用いた場合、非0を真値、0を偽値として扱う。

様々な分岐

- 条件分岐をするには、`if～else`, `switch～case`, 条件演算子などを用いる。
- `if～else` 文は条件分岐に以下の形式で用いられ、最も多用される分岐である。

```
if(条件式){  
    処理  
}  
else{  
    処理  
}
```

- `switch～case` 文は定数値を列挙する形での分岐に用いられ、次の形式で記す。

```
switch(値){  
case 定数式:  
    処理  
    break;  
:  
default:  
    処理  
    break;  
}
```

- `switch` 節中での `break` は、直上の `switch` 節からの抜け出しを意味する。
- 条件演算子は、条件に応じて変化する式を書く際などの値を返すだけの分岐に用い、次の形式で記す：`(条件式)?(真の場合の値):(偽の場合の値)`
- ここにあげた分岐 (及び次節で学ぶ反復) はいずれもネストすることができる。

第5講 反復構造

構造化定理の第3の構造、反復構造について学ぶ。

5.1 反復構造とは

反復構造は、ある条件の下、一定の処理を繰り返す構造である。例えば、「1以上1000未満の自然数の内、3ないし5の倍数である数の総和を求めよ」¹という問題を考える場合、1から1000までの数について、それが3ないし5で割り切れるかを判定し、割り切れるならばそれを加える、ということになる。換言すれば「ある自然数 n が3か5の倍数であるか判定し、真ならば足す」という処理を、 n を1から順に1ずつ増やし、 $n = 1000$ となるまで繰り返すということになる。

5.2 while 文

不定回数のループには、while 文を使うことが多い。

【角谷予想】

自然数がある時、それが偶数ならば2で割り、奇数ならば3倍して1を足す。この操作を繰り返して、最終的に1になるかどうか(全ての自然数は最終的に1になるという予想が角谷予想/Collatz 予想である。)を確認したい。これは、次のプログラムが終了するかどうかで確かめられる。
なお、unsigned int の範囲の全ての数について、この予想は成立することが知られている。

リスト 5.1: 角谷予想

```
1 #include<stdio.h>
2
3 int main(void){
4     unsigned int num;
5     scanf("%u",&num);
6     while(num!=1){
7         if(num%2==0){
8             num>>=1;
9         }else{
10             num*=3;
11             num++;
12         }
13     }
14     return 0;
15 }
```

¹これは Project Euler の第1問からの引用である。

【while 文による反復】

今回のテーマである while 文の公式を早速示そう。

while 文

while 文は

```
while(条件式){
    処理;
}
```

の形式で書き、条件式が真である限り {} 内の処理を繰り返すという意味になる。

while 文の動作について、もう少し詳しく説明しておこう。

まず、while 文に達すると、条件式が評価される。条件式が真であれば、{} 内の一連の処理が実行される。実行された後、再度 while の場所に戻り、条件式が真か偽かを判定する。これを、条件式が偽になるまで繰り返し、偽になったら while(...) {...} を全て飛ばして次の処理にうつる。以上が、while 文の動作である。

while 文は、通常不定回数のループに用いられる。例えば、リスト 5.1 のプログラムのように、数そのものに対して繰り返し処理を行ったり、ある漸化式が収束するまで計算したりする場合である。また、後に紹介する break 文と組み合わせて while(1) の形式で使い、無限ループにする場合もある。

条件式の値ということを利用した、少し高度なテクニックの例を紹介しておく。

【ターミネータ型総和】

整数が次々入力される時、その総和を出力するプログラムを作成する。なお、入力の終端は 0 とする^a。

【解説】

ここでは少し難しいテクニックを利用した。scanf は読み込みを行った場合に値を返し、その値は入力に成功したバイト数 (正) である。これと、入力された数が 0 でないかどうかをチェックし、都度計算を行っている。

^aこのように、入力などの終端を示す値のことをターミネータ (terminator) と呼ぶ。

リスト 5.2: ターミネータによる総和計算

```
1 #include<stdio.h>
2
3 int main(void){
4     int num,sum=0;
5     while(scanf("%d",&num) &&
6           num){
7         sum+=num;
8     }
9     printf("%d\n",sum);
10    return 0;
11 }
```

5.3 do-while 文

最低でも一回はやってほしいというとき、条件文の判定をしてから処理を繰り返す前置反復ではなく処理をした後にもう一度やるかどうか判定する後置反復という方法もある。これを実装するのが do-while 文である。では、今回はゲームを作成してみよう。

【数当てゲーム】

コンピュータが 0 以上 10000 以下の数のうち、一つを内部で決定する (これには擬似乱数を用いる。この節では、擬似乱数についても解説する。)。

プレイヤーはこれに対して、適当に自然数を入力していく。

各入力に対して、コンピュータの決定した数と異なっているならば、その数が決定した数より大きい小さいかを出力し、そうでなければ正解として終了するプログラムを作成する。

リスト 5.3: 数当てゲーム

```
1 #include<stdio.h>
2 #include<stdlib.h>
3 #include<time.h>
4
5 int main(void){
6     int num,input;
7     srand(time(NULL));
8     num=rand()%10001;
9     do{
10         scanf("%d",&input);
11         if(num<input)
12             puts("It's_Greater!");
13         else if(num>input)
14             puts("It's_Lower!");
15         else
16             puts("Congraturation!");
17     }while(num!=input);
18     return 0;
19 }
```

【擬似乱数】

コンピュータは完全な意味での乱数を作ることができない。そこで、乱数のように見える値がでる規則性 (漸化式ないし操作) に従って順次計算を行い、それを乱数列としている。このように、ある一定の規則のもと生み出される、乱数のように見える数のことを擬似乱数 (pseudorandom number) と呼ぶ。

先に書いたとおり、擬似乱数は数列であるので、その計算には初項が必要である。その初項のことを乱数の種 (seed of rand) といい、これを設定するのが srand 関数である。この、srand 関数の引数を初項とし、その後、rand 関数を呼び出す毎に擬似乱数列²の次の項が計算される。これが擬似乱数利用の仕組みである。なお、rand 関数及び srand 関数は stdlib.h に入っている。

ここでは、srand の引数を time(NULL) としている。この time 関数は、1970 年 1 月 1 日 00:00 からの経過時間 (sec) を返す関数である。本来、引数にはその時間を格納する変

²単に擬似乱数といった場合、実数の乱数もありうるが、rand 関数で計算されるのは整数の乱数である。

数のポインタを取るのだが、ここでは格納するつもりがないので NULL とした (詳細は後ほど、ポインタを学んだ後に理解されたい。)。したがって、プログラムの起動した時刻に応じて乱数の種が変わり、それによって計算される乱数が違うということになる。

以上がここで用いた擬似乱数の説明であり、一般にも比較的よく見かけるものである。まとめておこう。

(擬似) 乱数の使い方

(擬似) 乱数を用いるには、まず

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

と、二つのヘッダファイルをインクルードした後、

```
srand(time(NULL));
```

で、現在の時刻を乱数の種に設定し、必要毎に

```
rand();
```

で計算を行う。なお、乱数の種の設定は一度だけで良い。(rand 関数は呼び出しの際に前の乱数の値を覚えており、その値を用いて次の乱数を計算するため。)

【do-while 文】

ここで新しく出てきた構造である、do-while 文は後置反復を行うものである。後置反復とは、条件判断が処理の後に来る反復を言う。つまり、「処理を行う」→「その処理を再度実行するか判定する」という形式の反復構造である。反復の判定を後に持ってくることにより、よりプログラムが書きやすい場合 (例えば、初期値を一度以上改良してから、適切な値になるまで計算を行う方程式の反復解法など) に用いられる。

do-while の書式は次のとおりである。

do-while 文

do-while 文は

```
do{
    処理
}while(条件式);
```

の形式で書き、do 以後処理を一度行った後、条件式を判定し、再度反復を行うかどうか判定する。

while(条件式) の後にセミコロン (;) が必要なのを忘れがちなので注意。

5.4 for 文

while 文が不定回数のループによく用いられるのに対し、for 文は定回数のループに用いられることが多い構文である。では、具体例を見てみよう。

【FizzBuzz】

FizzBuzz は次のようなゲームである。

- 1 から順に整数を言っていく。
- 3 の倍数であるときには、整数を言う代わりに”Fizz”という。
- 5 の倍数であるときには、整数を言う代わりに”Buzz”という。
- 3 の倍数でも 5 の倍数でもあるときには、整数を言う代わりに”FizzBuzz”という。

入力される自然数 n まで、上記のルールに従って出力するプログラムを作成する。

リスト 5.4: FizzBuzz

```
1 #include<stdio.h>
2
3 int main(void){
4     unsigned int n,i;
5     scanf("%u",&n);
6     for(i=1;i<=n;i++){
7         if(i%15==0){
8             puts("FizzBuzz");
9         }else if(i%5==0){
10            puts("Buzz");
11        }else if(i%3==0){
12            puts("Fizz");
13        }else{
14            printf("%u",i);
15        }
16    }
17    return 0;
18 }
```

【for 文】

今回用いた for 文は、少しつかみにくい形の文かもしれないが、多用される文である。

for 文

for 文は

```
for(初期処理; 条件文; 終端処理){
    処理
}
```

の形式で記す。

この for 文は、次のような動作を行う。

1. まず、for 文のところに来たら、初期処理を行う。
2. 次に、条件文を判定し、真ならばブロックの処理を行う (偽ならば for の後に飛ぶ)。
3. ブロック終端まで来たら、終端処理を実行し、再度条件文判定部に戻る。

したがって、リスト 5.4 の for は

1. まず、 i に 1 を代入する。
2. i が n 以下であれば、ブロック中の処理 (if 節) を実行する。そうでなければ for 節を飛ばす。
3. i をインクリメントし、前項に戻る。

という処理になる。

この for 文において回している変数をカウンタ (counter) と呼び、通常 i, j, k をこの順で用いる。また、後述する配列などとの関連から、単に n 回繰り返したい場合には

```
for(i=0;i<n;i++)
```

の形で用いられることが多い。

もうひとつ、多重ループの例を見ておこう。

【九九表の出力】

入力される自然数 n に対して、九九表と同様の形で積の一覧表を出力するプログラムを作成する。

【解説】

やや難しいのは l.8 の printf (2 行にわかれているが、続けて書くこと) だろうか。%c 書式指定子が第 3 引数の条件演算子を用いて書かれた式に対応している。ここから、この %c は j が $n-1$ の時に限り改行になり、それ以外の時は水平タブとなる。

なお、多重ループは (書いてあるとおりに) そのまま読めばわかるが、内側のループが先に回る。つまり、 $(i,j)=(0,0),(0,1),\dots,(0,n-1),(1,0),\dots$ という順でループが行われる。

リスト 5.5: 掛け算表の出力

```

1 #include<stdio.h>
2
3 int main(void){
4     unsigned int i,j,n;
5     scanf("%u",&n);
6     for(i=0;i<n;i++){
7         for(j=0;j<n;j++){
8             printf("%4u%c",(i+1)*(j+1),
9                 j==n-1?'\\n':'\\t');
10        }
11    }
12    return 0;
13 }
```

リスト 5.5 の l.6~l.10 は、

```
for(i=0;i<n;i++) for(j=0;j<n;j++) printf("%4u%c", (引数略));
```

と、中括弧を用いずに書くこともできる。一つの処理だけを二重ループする場合にはこのような書き方をしても良いが、処理が複雑な場合は無理にまとめず中括弧を用いてブロック化して書いたほうが良い。

ここでブロック化したほうが良いと述べたのは、例えば

```
for(i=0;i<n;i++) for(j=0;j<n;j++){
    処理
}
```

のような記述や、

```
for(i=0;i<n;i++) if(i==0) if(n==0) n++; else printf("%d\n",i)
```

のような記述が文法上許されてしまうためである。これらの記述は複数の解釈が可能な「曖昧な文法」であり、誤解のもとである。そのため、構造を作る命令に対しては、ごく簡単な場合を除き、面倒であっても中括弧をつけたほうが良いだろう。

5.5 反復制御文と goto 文

反復を途中で抜けだしたり、処理を打ち切ったりするのが反復制御文である。また、流れを変える最も強力な手段が goto 文である。これらを用いたゲームの例を見ていこう。

【ハイ&ローゲーム】

現在出ている数字に比べて、次に出てくる数字が小さいか大きいかを当てるゲーム「ハイ&ロー」を作成する。

【解説】

様々な反復制御文を入れて、エラー処理を行った。ここまでで学んだ文法の多くが出ている、総復習になるソースである。

リスト 5.6: ハイ&ロー

```
1 #include<stdio.h>
2 #include<stdlib.h>
3 #include<time.h>
4
5 int main(void){
6     int num,next,money=50,bet,flag,stat;
7     char rep;
8     srand(time(NULL));
9     while(1){
10         printf("now your money is %d.\n",money);
11         printf("How much do you bet? [1-%d]>>>",money);
```

リスト 5.6: ハイ&ロー (続き)

```

12  scanf("%d%c",&bet);
13  if(bet<=0 || bet>money){
14      puts("Bet_error");
15      continue;
16  }
17  money-=bet;
18  stat=bet;
19  num=rand()%10;
20  while(1){
21      printf("now_num=%d, High&Low? [H/L]>>>",num);
22      scanf("%c%c",&rep);
23      if(rep!='H' && rep!='h' && rep!='L' && rep!='l'){
24          puts("Input_error");
25          continue;
26      }
27      next=rand()%10;
28      if(num==next){
29          puts("It's same! One more...");
30          continue;
31      }
32      if((rep=='H' || rep=='h') && num>next) flg=1;
33      else if((rep=='L' || rep=='l') && num<next) flg=1;
34      else flg=0;
35      if(flg==1){
36          printf("The number is %d! You lose...\n",next);
37          break;
38      }else{
39          printf("The number is %d! You win!\n",next);
40          challenge:
41          printf("You can bet continuously. Challenge? [Y/N]>>>");
42          scanf("%c%c",&rep);
43          if(rep=='N' || rep=='n'){
44              printf("Okay, you get %d money!\n",bet);
45              money+=bet+stat;
46              break;
47          }else if(rep=='Y' || rep=='y'){
48              puts("Okay, you challenge next!");
49              bet*=2;
50          }else{
51              puts("Input_error");
52              goto challenge;
53          }
54      }
55      num=next;
56  }

```

リスト 5.6: ハイ&ロー (続き)

```
57     if(money==0){
58         puts("you can't bet!");
59         break;
60     }
61     more:
62     printf("One more?>>[y/n] ");
63     scanf("%c",&rep);
64     if(rep=='N' || rep=='n'){
65         break;
66     }else if(rep=='Y' || rep=='y'){
67         puts("Okay, you bet next!");
68     }else{
69         puts("input error");
70         goto more;
71     }
72 }
73 printf("your last money is %d.\n",money);
74 puts("See you again!");
75 return 0;
76 }
```

非常に長いソースであるが、新出文法は少ない。

【break 文と continue 文】

l.15 や l.25 に見られる continue 文と l.37 や l.46 に見られる break 文がここで説明する重要な新出文法である。

continue 文は、それが含まれる直近階層のループ終端の}の直前に飛ぶという命令である。例えば、l.15 は l.9 の while 終端 (=l.72) に、l.25 や l.30 は l.20 の while 終端 (=l.56) に飛ぶことになり、while の判定が再度行われることになる。なお、continue 文はブロック終端に飛ぶ命令であるため、for 文中で continue を用いた場合、その終端処理は忘れず行われる。

break 文は、それが含まれる直近階層のループの終端直後に飛ぶ (=ループから抜け出す) 命令である。l.37 の break は l.20 の while 終端 (=l.56) から、l.65 の break は l.9 の while 終端 (=l.72) から抜け出すことになる。break 文はその性質上、条件文と併用して用いられることが多いが、switch case 文と併用すると switch に対する break 文と解釈されてしまい、ループ抜け出しにならない点に注意されたい。なお、break は抜け出しであるので、for 文の終端処理は実行されない。

【goto 文とラベル】

goto 文は通常使わないほうがいいとされる文である。この文は、任意のラベルまで飛ぶという命令である。ここで、ラベルとは *l.40* や *l.61* のように、

任意の名前:

によってつけられるものである。これに対して goto は

```
goto label;
```

の形で記し、label まで飛ぶという意味になる。この goto 文を無闇に用いると流れがわかりづらいスパゲッティソースコードになるため、使用する際には十分注意を払い、多重ループからの抜け出し程度のみとすべきである。基本的に全く使わないとしておいても良い。

【リスト 5.6 の簡単な解釈】

ソースの大雑把な説明だけを記しておく。

l.6～8 初期設定。

l.10～*l.16* 賭け金の入力とそのエラー処理。

l.17～*l.19* 賭けの初期設定。

l.21～*l.26* 高いか低いかの入力とそのエラー処理。

l.27～*l.39* 次の番号の抽選と高い/低いの正誤判定。

l.40～*l.56* 勝った場合に、賞金を次の賭け金にしてそのまま連続チャレンジするかの処理。

l.57～*l.60* 所持金が尽きた場合の処理。

l.61～*l.72* もう一度プレイするかどうかの処理。

l.73～*l.76* 最後の出力。

本講の要点

本講では、反復構造の記法と乱数について学習した。

反復構造と反復制御文

- 反復構造は次のように作ることができる。

- while 文。

```
while (条件式) {  
    処理  
}
```

といった形で用い条件式が真である場合にその処理を実行し続ける。

- for 文。

```
for (初期処理; 条件式; 終端処理) {  
    処理  
}
```

の形式で記す。この文にきた時に初期処理を実行したのち、条件式を評価し、非0ならばブロックの処理を実行する。その後、終端処理を施して、再度条件式の評価を行う。これを繰り返す。

- do while 文。

```
do {  
    処理  
} while (条件式);
```

といった形で用い、まず一度ブロック内の処理を行った後 while と同じ動作を行う。

- 反復構造で途中から抜きたい場合には `break` 文を、途中でそのブロック終端まで移動したい場合は `continue` 文を用いる。これらを反復制御文という。
- `goto` 文とラベルを用いて流れを制御することもできるが、可読性を下げることにもなりかねないので無闇に使うべきではない。

擬似乱数

- コンピュータは本当の乱数を作ることができないので、乱数に見えるようなある一定の規則に従った数列を乱数の代わりに用いる。これを擬似乱数という。
- 擬似乱数の乱数列の初項を乱数の種といい、`srand` 関数で設定する。
- `rand` 関数は内部状態として乱数列の前項の状態を保存しており、呼び出すことで前項の乱数から別の乱数を計算して返却する。

第6講 自作関数とプリプロセッサ

本講では関数について学ぶ。関数 (function) とは、引数をとって、ある一定の処理を行い、返却値を返すものである (引数が無かったり返却値が無かったりするものもある)。実際、printf や rand など引数を取り、返却値を返す関数である。だが、これら C 言語に標準搭載されている関数 (標準関数) だけでは物足りないこともある。例えば、暗号化や数論でよく使われる素数の判定関数は標準関数としては提供されていない。だが、素数の判定は多くの目的で用いられるため、関数として準備し、使い回したほうが効率的にプログラミングできる。このような、自作の関数を作る方法の基礎を見ていく。とはいえ、main 関数も自作関数であるので、この講で扱う自作関数とは、厳密には「main 関数以外の自作関数」である。

また、関数の作成に必要な「スコープ」「寿命」などについても学習し、それと関連の深いプリプロセッサについても説明する。

6.1 関数の呼び出しと利用

実際の関数を作る前に、ざっと復習しておこう。

既存の関数を呼び出す際には、次のように行った。

関数の呼び出し

関数を呼び出す際には、関数を呼び出す場所で

関数名 (引数)

のように記す。

これにより、既存の関数の値 (ないし処理) を利用することができた。このように、名前を書いて呼び出すためには、関数そのものの定義が必要であるが、これはヘッダファイル及びその機能を提供するライブラリによって行われている。

したがって、自作関数を利用する際には、ヘッダファイルやライブラリで行われているようなことをソースに埋め込み (ないし、ヘッダやライブラリを自作し)、後は上記の関数名 (引数) の形式で呼び出せばよい。

6.1.1 関数化の意義

ここで、改めて関数化する理由を考えてみよう。

我々は既に複数の関数を用いている。仮に、これらがなかったとしたらどうなるだろうか。例えば、math.h の ceil 関数を考えてみよう。

誤差がなく、絶対値が極端に大きくないとすれば、ceil 関数は、

```
(double)(int)x+(x-(int)x>0)?1:0;
```

のような一文で実装することができる。だが、この一文を見てすぐさま”これは ceil 関数の処理だ!”とわかるだろうか。ceil という名前をつけておいたほうがわかりやすいのではないだろうか。また、誤差や絶対値が大きい場合にはこの方法は通用しないので、別の実装が必要になる。これをいちいち組むのは面倒であるばかりでなく、腕による違いも出てきてしまう。そもそも、そのような場合にどう実装すればよいか、すぐに思いつくだろうか。これらの実装が ceil の一言ですむのであれば、そちらのほうが余程わかりやすいだろう。

このように、関数化をすることは

- 可読性を上げる。(どんな処理かが明確になりやすい)
- 似たような処理を書くときなどに面倒でない。(使い回しが可能である)
- 内部が厳密にわからなくても使うことができる (処理のブラックボックス化)

等、多くの利点がある。この内、我々が自作関数を使う場合は、主に前の 2 つの理由をもって、自作関数を作ることになる。

熟達したプログラマは、自分がよく使う関数をライブラリとしてまとめて持っている場合も多い。これはまさに、前の 2 点の利点のためのものといえよう。

6.1.2 関数の要素

冒頭で説明したとおり、関数には大きく分けて 3 つの要素がある。

引数 関数に渡すべき値。

処理 関数の中でどのようなことが行われているかという動作。

返却値 関数において処理が行われた後に返される値。

この内、引数には関数の定義の際に用いられる**仮引数** (parameter) と、実際の呼び出しの際に用いられる**実引数** (argument) とがある。これらについては、後で自作関数を作る際に実例を伴って理解して欲しい (その際に再度紹介する)。

一方で、関数の処理には、返却値以外にも後に影響を与えるものが存在する。例えば、srand 関数は返却値を持たないが、rand の呼び出しに影響を与える。このように、返却値以外で関数が処理によってコンピュータの状態等に与える影響を**副作用** (side effect) と呼ぶ。関数によっては、返却値より副作用に重きを置かれることもあり (printf 関数や scanf 関数など)、そもそも返却値がなく副作用目的の関数も存在する (先にあげた srand 関数など)。このような副作用目的の関数は「引数をとって何らかの処理を行い、その返却値を返す」という関数本来の説明よりも、「引数をとって、それに応じて動作する」という、ある種の「操作」と捉えたほうが理解しやすいかもしれない¹。

¹実際、言語によっては返却値を返すようなものを関数と、操作を目的とするものをサブルーチンとして分けているものもある (Fortran 等)。

6.2 自作関数の利用

では、ここまで前置きが長かったが、実際に関数を作成してみることにしよう。ここでは、比較的汎用性が高い「素数の判定関数」を考える。

【素数の全列挙】

入力される自然数 n までの素数をすべて出力するプログラムを作成する。

【解説】

素数を判定する関数は `isPrime` として作成した。この関数の内部の実装は「2以上の自然数 p が素数かどうか判定するには、 \sqrt{p} 未満の全ての素数で割り切れるかどうか判定すればいい」を弱め、「 \sqrt{p} 未満の全ての奇数及び2で割り切れなければいい」を用いた。

一方、`main` 関数においても、簡単のため偶数を無視して、奇数だけを判定するように書いてある。すなわち、`isPrime` 関数は一般化されたものであるが、その呼び出し回数を減らし、無駄な計算を避けるために、`main` 関数でも工夫を凝らしているということである。なお、紙面の都合上、`for` や `if` で `{}` を省いた書き方を多用している。

リスト 6.1: 素数の判定

```
1 #include<stdio.h>
2 #include<stdbool.h>
3
4 typedef unsigned int u_int;
5
6 bool isPrime(u_int p);
7
8 int main(void){
9     u_int n,i;
10    scanf("%u",&n);
11    if(n>=2) puts("2");
12    for(i=3;i<=n;i+=2)
13        if(isPrime(i)) printf("%d\n",i);
14    return 0;
15 }
16
17 bool isPrime(u_int p){
18     u_int i;
19     if(p==2)
20         return true;
21     else if(!(p%2) || p<=1)
22         return false;
23     for(i=3;i*i<=p;i+=2)
24         if(!(p%i)) return false;
25     return true;
26 }
```

新出文法が多いソースであるので、新出文法事項毎に説明していこう。

6.2.1 自作関数の作成方法

まず、ここまで議論してきた自作関数の実例を見ていくことにしよう。リスト 6.1 において、自作関数として作られているのは、`l.17` から `l.26` の `isPrime` 関数である²。これを見てもらえばわかるとおり、関数は `main` と同じように、`{}` で囲まれており、その前に引数リストが、その前に名前があって、その前に型が書かれている。`main` 関数も同様の形式で書かれており、関数の実装の際にはこの形式を守らなければならないことがわかる。以上をきちんとまとめると、次のようになる。

²冒頭にも書いたが、厳密には `main` 関数も自作関数である。

自作関数の実装

自作関数を実装するには

```
返却値の型 関数名 (引数リスト){  
    処理  
}
```

の形で記す。

この、「処理」の部分の書き方は main 関数と同じであるので、省略してよいだろう。残りの「返却値」「関数名」「引数リスト」について説明していこう。

【自作関数の返却値】

返却値について押さえておくべきことは、その型と決定方法である。

返却値の型は変数の型と同じように書くことができるが、変数の型には存在しない void というキーワードがある。これは、「返却値なし」を意味する。すなわち、先に説明した「処理のみを目的とする関数」などで使われる型である。たとえば、isPrime 関数では、返却値の型は bool 型である。

なお、返却値の型を省略した場合、勝手に型を定めるコンパイラもある³が、とりわけ自作関数の場合は型がないと分かりづらいため、必ず型を書く癖をつけておくこと。また、返却値の型は関数を代表する型であるため、一般には関数の型と呼ばれる。「関数の型」という場合「関数の返却値の型」と補って理解すればよい。

これらによって、返却値の型が定まったら、後は具体的に返却値を定めればよい。

返却値の決定

返却値は、

```
return (返却値);
```

の形式で定める。この返却値には変数や計算式を指定してもよい。

関数は、返却値が定まった時点 (=return が実行された時点) で、その処理を終了し、呼び出し元に戻る。すなわち、return は返却値を定めるだけでなく、関数の終了を示す役割も果たしているわけである。isPrime の中には沢山の return が書かれているが、このうちのどれか一つが実行された段階で関数の処理は打ち切れ、呼び出し元に返却値が返されることになる。

この「関数を終了させる」という役割のため、void 型関数でも return が用いられることがある。この場合は単に

```
return;
```

³試しに、適当なプログラムについて main 関数の前の int を省略してみよう。問題なく動作するはずである。(とはいえ、main 関数は特別扱いという処理系もあるが)。

とだけ書かれ、それが実行された段階で関数の処理が打ち切られる。

なお、返却値はどのような関数についても1つまでしか持てない。2つ以上の値を返したい場合、後に学ぶ構造体を用いて2つ以上の値を変数1つに見せかけたり、副作用を利用して返却したりといった工夫が必要になる。

【関数名について】

自作関数の名前は識別子 (identifier) の一種であるので、変数名の命名規則の時に説明したようなルールを守らなければならない。また、似たような処理を行う関数であっても、関数名は明確に違うものをつけなければならない⁴。これは、math.h 中の sin 関数が、sinf, sin, sinl の3つに分かれているということからもわかるだろう。

【引数について】

関数の引数は、関数名の後の () に、コンマ区切りで記す。一つ一つの引数には型をつけ、変数の宣言と同様の形式で記す。

引数リスト

関数の引数は

関数型 関数名 (引数1の型 引数1の識別子, 引数2の型 引数2の識別子, …)

の形式で記す。

実際、引数リストの各引数は、その関数の実装において用いられる変数の宣言でもある。つまり、引数として () 内に記された各変数は、呼び出しの際に呼び出し元の値で自動的に初期化される、関数内のみで通用する変数であるということである。このように、関数の定義において用いられる引数を仮引数 (parameter) と呼び、呼び出しの際に呼び出し元の値が代入される点を除いては変数と同じ扱いができる。一方で、呼び出し元の関数の引数のことを実引数 (argument) と呼ぶ。

先のリスト 6.1 において、仮引数は l.17 の p であり、実引数は l.13 の i である。なお、これらの型の u_int は見たことがない型であろうが、これについては後で説明する。同様に、l.6 についても後述する。

仮引数と実引数において重要なのは、「仮引数はあくまでも実引数とは別の変数であって、値が代入されるだけである」ということである。別の変数であるため、関数中で仮引数をインクリメントしたりしても、実引数 (= 元の呼び出し側の引数) には影響しない。このように、仮引数に値をコピーするような引数の渡し方を値渡しと呼ぶ。C 言語では、一

⁴一方、Java や C++ では、引数の型や個数が違えば、同じ名前の別の関数を作ることができ、呼び出し側の与えた引数に応じて自動的にどの実装を呼び出すかを決定して実行してくれる。これを関数の多重定義ないしオーバーロード (overload) と呼ぶ。C 言語でこれと似たような機能を実装する場合には、後述するマクロを用いるか、C11 の新機能を用いるかといった手段がある。似た機能を提供する tgmath.h でも、その内部実装にはマクロを用いている。

部の例外を除き、関数への引数の渡し方は値渡しである⁵⁶。この仕ようにより、関数内で気軽に仮引数をいじることができるが、反面、実引数に影響を及ぼすような関数は別の方法を用いて作ることが出てくる。

6.2.2 関数プロトタイプ宣言

ここまでで、自作関数の作成方法について説明したが、リスト 6.1 の新出文法はこれだけではない。今度は *l.6* の関数の宣言らしきものについて見ていこう。

この *l.6* の宣言は関数プロトタイプ (function prototype, 関数原型とも) ないし、プロトタイプ宣言 (prototype declaration) という宣言である。

通常、ソースは上から読まれる。これは、コンパイル時も同様で、変数や関数は、それが使われるより上で宣言しなければならない(後でより詳細に説明する)。ところが、main 関数より上で、必要な関数ばかり書いていると、最も重要である main 関数が下の方に来てしまい、可読性を損なうことになりかねない。そのため、main 関数より上で「このような引数を取り、型が〜の関数***を使います」と宣言のみ行い、定義は main 関数より下に記す、という方法を取るのが一般的である。

プロトタイプ宣言

自作関数を用いる際には、main 関数より前に

関数型 関数名 (引数 1 の型 引数 1 の識別子, 引数 2 の型 引数 2 の識別子, …);

という形で宣言のみしておき、これと同じ形で main 関数より後に実態を定義する部分を書く。(引数の識別子は付さなくても良い)。

ここで出された *l.6* の宣言が isPrime 関数のプロトタイプ宣言であり、この宣言があるために main 関数の中にある isPrime がコンパイルエラーにならないのである(試しに、コメントアウトしてコンパイルすると、コンパイルエラーになるだろう)。

6.2.3 型の別名定義

名前が長い型を何度も使わなければならない場合、煩わしいと感じる場合が出てくる。例えば、

```
unsigned long long func(unsigned long long par1, unsigned long long par2);
```

などという関数を打てと言われたらどうだろうか。打つのも面倒であるし、また、読みづらくもある。

そこで、型の別名を定義できるようにしたのが typedef という修飾子である。

⁵⁶後に習うポインタを用いるとポインタ渡し関数も作ることができるが、これは単にポインタの値を渡しているだけ=「ちょっと変わった型の」値渡しであるので、本質的にはこれだけといってしまうかもしれない。配列渡し、関数ポインタ渡しなど少し怪しい(値渡しのように見えない)渡し方もあるが、これらも「特殊な型の」値渡しと考えられる。それ故、C 言語には原則値渡ししかないと思っていいだろう。

⁶C++などでは、値渡しだけでなく、変数の実体そのものを渡すような参照渡しもある。C 言語で参照渡しをするためには、後に学ぶポインタを用いたポインタ渡し(=ポインタの値渡し)を用いる。

型の別名定義

ある型に別の名前を付けたい場合

```
typedef 元の型 別名;
```

の形式で記す。

先に出した例において、プログラム冒頭で

```
typedef unsigned long long ull_int;
```

などとしておけば、関数は

```
ull_int func(ull_int par1, ull_int par2);
```

とずいぶん短くなり、読みやすくなる。これが型の別名定義で、通常はインクルードなどが終わった後(あるいはヘッダファイル)に記し、プログラム全体で使えるようにする。

リスト 6.1 では、`unsigned int` が長いので `u_int` として、読みやすくしたわけである。

6.3 スコープと寿命

リスト 6.1 には、2ヶ所も `i` という変数が登場している。これはどこで通用するのだろうか。また、`isPrime` 中の `i` は何度も呼び出されるわけだが、その都度前の値を保持していたりはしないのだろうか。本節ではこの問題について議論する。

6.3.1 オブジェクトのスコープ(有効範囲)

何らかのオブジェクト(変数、関数、後述のマクロなど)には、スコープ(scope)ないし有効範囲というものが定まっている。これは、そのオブジェクトを(ソース上の)どこから呼び出せるか定まっているという事である。リスト 6.1 の `l.9` の `i` は `main` 関数の中でしか通用しないし、`l.18` の `i` は `isPrime` 関数の中でしか通用しない(つまり別物であるという事)。このように、オブジェクト毎に定まっている「ソースのどこからアクセスできるか」の範囲のことをスコープと呼んでいる。

オブジェクトのスコープは一般に次のとおりである。

一般オブジェクトのスコープの性質

- オブジェクトのスコープは、それが宣言されたより後の部分である。宣言前のオブジェクトは使えない。
- 何らかの節(`{}` で囲まれた部分及び、`if`, `for` 等の構造を作る節)の中で宣言されたオブジェクトは、その節及びそれよりも深い階層のネストでしか通用しない。
- 同一名のオブジェクトのスコープが重複した場合(通常避けるべきである)、よりスコープが狭い側のオブジェクトが優先される。

簡単に言えば、ある{}内で宣言されたとして、その宣言以降で、宣言した階層の{}の終わりが来るまで、というのがスコープである。スコープの外では、そのオブジェクトにアクセスしようとしても、そのようなオブジェクトは存在しないという意味のコンパイルエラーが出る。

【宣言とスコープ】

ここまでの解説から、変数の宣言や関数のプロトタイプ宣言の意味を再考してみよう。

変数は宣言することにより、それを使うことができるようになった。これは、変数にスコープを付与することに他ならない。一方で、関数のプロトタイプ宣言は、実体定義を呼び出しよりも後で行う際に「そんな関数はない」と言われるのを防ぐため行うと書いた。しかし、先のスコープの性質に照らし合わせてみれば、何のことはない、関数プロトタイプ宣言は「定義とは別の宣言」であり、スコープを決めるための通常の宣言と何ら変わらない役目を果たしているのである。

定義と宣言の違いは、ここまでの内容ではあまり良くわからなかっただろうが、スコープの概念を取り入れれば明解になる。

定義と宣言

宣言 (declaration) あるオブジェクトのスコープを定める動作。

定義 (definition) あるオブジェクトがどのような動作をするか定めてメモリ上に確保する動作。

最後に、スコープの違いがよくわかるソースをひとつ試して、1ソースの場合のスコープの解説を終わろう。

【グローバル変数とローカル変数】

ソースの全ての関数で通用する変数(グローバル変数ないし大域変数)と、ある関数の中でしか通用しない変数(ローカル変数ないし局所変数)の違いを見てみる。スコープを考えて動作結果を見てみよ。

リスト 6.2: 変数のスコープの違い

```
1 #include<stdio.h>
2
3 int n;
4 void func(int x);
5
```

```
6 int main(void){
7     int x;
8     scanf("%d",&n,&x);
9     printf("1:%d\n",n,x);
10    func(x);
11    printf("4:%d\n",n,x);
12    return 0;
13 }
14
15 void func(int x){
16     printf("2:%d\n",n,x);
17     n++;
18     x++;
19     printf("3:%d\n",n,x);
20 }
```

6.3.2 変数の寿命

先に解説したスコープは「リスト 6.1 には、2ヶ所も `i` という変数が登場している。これはどこで通用するのだろうか。」という問いに対する答えであった。もうひとつの疑問「`isPrime` 中の `i` は何度も呼び出されるわけだが、その都度前の値を保持していたりしないのだろうか。」の答えを説明するのが変数の寿命である。

変数は通常、宣言されると同時にメモリ上に確保されるわけだが、これはいつ取り除かれるのだろうか。この、「メモリ上に変数が確保されている期間」のことを**変数の寿命** (variable extent または variable lifetime) という。変数の寿命とスコープは誤りやすいので、以下に違いをまとめておこう。

スコープと寿命の差異

スコープ ソースないしプログラムにおいて、どの部分からそのオブジェクトを呼び出すことができるかという「ソース上での位置」の意味での有効範囲。

寿命 変数がいつからいつまでメモリ上に置かれているかという、「メモリ上に置かれている期間」を示す指標。

通常の変数の場合、定義された後スコープが尽きるまでが寿命である。例えばリスト 6.1 の `isPrime` 中の `i` は、`isPrime` が呼び出される毎にメモリ上に配置され、`isPrime` が返却値を返すたびにメモリから取り除かれる。実際には、このような変数は `auto` という指定の寿命にであり、**自動変数** (auto variable) と呼ばれる。この `auto` のように、寿命その他の変数の性質を決めるための指定子が**記憶クラス指定子** (storage class specifier) である。

【記憶クラス指定子】

記憶クラス指定子は寿命 (及びスコープ等の変数の記憶に関する性質) を定めるための修飾子で、表 6.1 に示すような種類がある。

表 6.1: 記憶クラス指定子の一覧

修飾子	名称	意味
<code>auto</code>	自動変数	(通常の設定)
<code>register</code>	レジスタ変数	CPU レジスタに確保される変数
<code>static</code>	静的変数	寿命が宣言後プログラム終了までとなる変数
<code>extern</code>	外部参照変数	別ファイルにおいて定義された変数

これらのうち、自動変数については、先に書いたとおり、標準通り「定義された後スコープが尽きるまでが寿命の変数」のことである。また、外部参照変数の指定子 `extern` はスコープに関連する指定子で、1 ファイルの場合には使わないので後回しにする。

`register` で指定される**レジスタ変数** (register variable) は、通常のメモリではなく、CPU レジスタと呼ばれる「CPU に最も近いメモリのような場所」に格納される変数である。この指定子をつけた変数はアドレスを取得できないなどの制限を受けるものの、優先的に

レジスタに格納され、計算速度の向上に役立つ。但し、レジスタにおかれることが保証されるわけではなく、必ずしも速くなるとは限らない。

重要なのは static で指定される静的変数 (static variable) である。これについて、簡単なプログラムを見て見ることにしよう。

【呼び出し回数を数える関数】

呼び出し回数を数える関数を作成する。入力 is 自然数とする。

```
リスト 6.3: 呼び出し回数
1 #include<stdio.h>
2
3 int func(void);
4
5 int main(void){
6     int n,k;
7     scanf("%d",&n);
8     while(n-->0) k=func();
9     printf("%d\n",k);
10    return 0;
11 }
12
13 int func(void){
14     static int num=0;
15     return ++num;
16 }
```

リスト 6.3 では、入力された数と同じ値が出力されるはずである。これは、関数 func が入力されたと同じ回数だけ呼び出されたという事であり、確かにプログラムに合致する。このことから、++num は func() の呼び出し回数だけ実行されていることになり、呼び出し毎の初期化が行われていないこともわかる。このように、静的変数は

- 初呼び出し時に定義・初期化され
- 以降プログラム終了まで初期化されることなくメモリ上に配置されている

という 2 つの性質をもつ変数である (試しに、static を外すと適切に動作しないことを確認してみよう)。静的変数は、回数を数える他、関数が内部状態を持つ時⁷などに用いられる。

なお、大域変数は何もしなくとも静的変数の性質を持つため、static はローカル変数にのみ用いられる指定子であることを付記しておく。

6.4 前処理命令と分割コンパイル

自作関数と、それに関連した 1 ファイルでの寿命やスコープについてここまで解説した。この節では、これまで決まり文句としてかいていた # 始まりの文——前処理命令 (preprocessing directive)、コンパイラに対する命令——の活用法について学ぶ。とりわけ、もう一つ重要なオブジェクトであるマクロと、毎度使っているインクルードに焦点を当てることにする。この説明によりインクルードの意味が詳らかになると、複数ファイルを用いた

⁷内部状態を持つ関数とは、引数等で指定する必要はないが、前回の値を利用する場合などにその前回の値を保持しているような関数のことである。例えば、rand 関数は前回の計算結果を内部で保持していて、それを利用して計算する、内部状態を持つ関数である。回数を数える関数も、回数という内部状態を持つ関数である。

プログラムという観念が自然に意識されることだろう。ここでは、これらの複数ファイルからなるプログラムのコンパイルやその場合のスコープについても説明する⁸。

6.4.1 マクロ

マクロ (macro) とはソースコード中において「このように書いたら置き換えてください」という指示である。例えば、NUM と書いたら 512 に置き換えると設定しておき、プログラム中で NUM と書いてソースを書いておけば、後からこの NUM を書き換えたいくなった時に置き換えのルールを変更するだけですむ。このような「置き換え」が C プログラムにおけるマクロである。

【マクロの利用】

2 種類のマクロを宣言して、その動作を見てみる。

リスト 6.4: マクロの利用

```
1 #include<stdio.h>
2 #define NUM 1024
3 #define ABS(x) (((x)<(0))?(-(x)):(x))
4
5 int main(void){
6     printf("%d\n",ABS(NUM));
7     return 0;
8 }
```

では、マクロについて、学んでいくことにしよう。なお、ここではマクロの基本的な扱い方及び注意点を述べるに留める。より発展的な使い方については付録を参照すること。

【オブジェクト形式マクロ】

リスト 6.4 のプログラムの 1.2 で使われているような、単に数字や文字列を置き換えるだけのマクロをオブジェクト形式マクロ (object-like macro) と呼ぶ⁹。この文は、「以降 NUM と書かれていた場合、これを 1024 に置き換えてくれ」という意味である。では、これを確認するため、マクロの記述方法を述べよう。

⁸ここで説明する複数ファイルのコンパイルは、主に原理の部分であり、各種ツール (make や IDE など) の利用については触れない。

⁹オブジェクト形式マクロという名前であるが、オブジェクトのみを置き換えられるわけではなく、何でも置き換えられる。これは、後に紹介する関数形式マクロも同様である。極論、ソースコードを全てマクロにして、マクロを乱発するだけでソースコードを構成することもできる。例えば、

```
for(i=a;i<n;i++)
```

を、REP(i,n,a) などというマクロにする人が少なからず見られる。

マクロの記述

マクロを利用する場合には

```
#define 置き換え名称 元の名称
```

の形式で記述する。

このことから、1.2のマクロはたしかに、「以降 NUM と書かれていた場合、これを 1024 に置き換えてくれ」という意味になっていることがわかる。

【関数形式マクロ】

リスト 6.4 のプログラムの 1.3 には、関数のように (引数) を伴ったマクロが定義されている。このようなマクロを関数形式マクロ (function-like macro) と呼ぶ。

関数形式マクロ

関数形式マクロを利用する場合には

```
#define 関数名(引数リスト) 置き換え文字列
```

の形式で記述する。

関数形式マクロも関数と同様に引数リストを準備し、それを用いて置き換え文字列を書く。例えば、先の ABS(x) というマクロが、実際のプログラム中 (main 関数の中など) で、

```
ABS(-5)
```

と書かれていれば、これは、

```
(((-5)<(0))?( -(-5)):(-5))
```

と展開される。このように、引数を用いたマクロが関数形式マクロであり、簡単な処理であれば関数を使うよりも速い場合が多い。

【マクロのスコープとマクロに関する命令】

マクロにもスコープがあり、これは定義されて以降定義が終了するか、ファイル終端かまで続く。しかし、マクロには階層がないため、新たに定義されてしまうと、元のマクロが置きかわってしまう。これは、コンパイラの Warning としては検出されるが、Error では無いので、再定義しないように注意する必要がある。

だが、大きなプロジェクトになると、マクロを定義するたびにいちいち他のマクロ全てをみて「定義されていない」ことを確認するのは煩わしいか、場合によっては不可能である。そこで、マクロに関する各種のプリプロセッサ文が役に立つ。マクロの多重定義を防いだり、コードを状況に応じて書き換える等の操作を行うためのプリプロセッサ文をひと通り紹介しておこう。とりわけ、大規模なプロジェクトや、コンパイル環境に応じたソースコードの変更などに用いられることが多い。

マクロ関連の前処理命令

#define マクロを定義する。

#undef マクロの定義を解除する。

#if 引数に示される定数式が真ならば**#endif** までのテキストを挿入する。

#ifdef 引数に示すマクロが定義されていれば**#endif** までのテキストを挿入する。

#ifndef 引数に示すマクロが定義されていなければ**#endif** までのテキストを挿入する。

#elif 上記の**#if(n)def** や**#if** と組み合わせて用い、プリプロセッサ文において **else if** と同じ役割を果たす。

#else 上記の**#if(n)def** や**#if** と組み合わせて用い、プリプロセッサ文において **else** と同じ役割を果たす。

#endif 上記の**#if(n)def** や**#if** と組み合わせて用い、これらの節の終端を示す。

defined 通常**#if** や**#elif** と組み合わせて用い、「この引数のマクロが定義されていれば」という意味を持つ。

ここにあげた全ての命令を逐一解説するつもりはないが、簡単な書き換えの仕方だけ紹介することにしよう。例えば

```
#ifndef NUM
    #define NUM 512
    (NUMを使ったソース (ソース A))
    #undef NUM
#else
    #define NUMBER 512
    (NUMBERを使ったソース (ソース B))
    #undef NUMBER
#endif
```

というコードを考えよう。これは、NUMをそれまでに定義していなければソース A の部分を採用してコンパイルし、定義していればソース B の部分を採用してコンパイルするようにしたコードである。このようにして、状況に応じてコンパイルするコードを自動的に選択させる機能として、上記の命令を用いることが多く、これによってマクロの多重定義を防ぐことができる。

なお、このようにして条件をつけてコンパイルすることを、文字通り条件付コンパイル (conditional compilation) と呼ぶ。

【マクロに関する諸注意】

ここまで、マクロを色々を見てきたわけであるが、

- マクロの名称がすべて大文字であること
- 関数マクロはやたら () が多いこと

が気にならなかっただろうか。これは、マクロを使う上で注意すべき点である。

一般に、マクロは他のオブジェクトと区別するために、英大文字と数字、アンダースコアからなる名称を付ける。これは慣例的に決まっているものであるので、必ずしもこれに従う必要はないが、どれがマクロで、どれが変数なのか一目に区別がつくほうが読みやすいため、できる限り従うことを推奨する。

関数マクロにやたら () が多いのは、次のような例について考えてみればわかるだろう。

```
#define SQUARE(x) x*x
```

見た目には問題が無いように思える。しかし、ここで `SQUARE(a+b)` という呼び出しは

```
a+b*a+b
```

と展開されてしまい、 $(a+b)^2$ を返してほしいはずが、 $ab + a + b$ という、全く違った値を返す動作になってしまう。これは、関数と違って評価後に処理されるわけではなく、コンパイル時の単なる置き換えであるために起きる問題である。実際にこのような局面に出会ったとしたら、マクロによってエラーが隠蔽されてしまうため、人間の目で見つけるのが困難になってしまう。同様の演算子の計算順序や演算子の結合などから出るエラーを防ぐために、関数形式マクロでは逐一 () を付けるのが安全である。また、インクリメント等と組み合わせると、一層意図しない動作になることがある (2度評価されてしまう等) ので、注意深く扱わねばならない。

なお、マクロによるエラーを発見するためには、マクロの展開結果を見たほうがわかりやすい。gcc では、マクロを展開した後 (厳密にはプリプロセッサの行動を実行した後) のプログラムを見ることができる。

プリプロセッサ実行結果の確認

あるソースに対して、プリプロセッサの実行結果を確認するためには

```
cpp ソース名
```

を実行する。

なお、リダイレクトすることで、展開後のソースそのものをファイルとして保存できる。プリプロセッサの動作を理解したい場合に使うと良い。

関数マクロについて、もう少し注意点がある。これは利点でもあるが、関数マクロには型チェックがない。そのため、先の `ABS` 関数マクロのように型を気にせず使える反面、整

数同士の割り算によるバグなどを招く場合もある。そのため、関数マクロを使う場合には、通常の変数だけを使う場合に比べ、プログラマが尚更型を意識しなければならない。

また、関数マクロの引数は見かけ上の引数であって、値がコピーされている関数の引数とは違うものである。マクロの引数は単に置き換えられるものに過ぎず、それ故マクロ中で変数いじりを行うとそれがモロに置き換え後の変数にも影響する。逆に言えば、変数を直接扱うこともできるという事でもある。

先の多重定義も含め、マクロの利用で注意すべき点をまとめておこう。

マクロ利用上の注意

- マクロは多重定義してはならない。できれば`#ifndef`などを用いて、多重定義を防止するコードを書くように心がけよ。
- マクロは他オブジェクトと区別するため、英小文字を使わない識別子とする。
- マクロ定義時には、展開後の構文解釈の齟齬を防ぐため、項毎に `()` を付すこと。
- プリプロセッサ展開結果を見るには `cpp` コマンドを用いる。
- 関数マクロには型チェックがない。
- 関数マクロの引数は見かけ上の引数であり、値がコピーされて動作が行われるわけではない。

6.4.2 ヘッダファイルの実態とインクルード

ここまで天下りに書いてきたインクルードについてもう少し掘り下げよう。

【インクルード文の動作】

サブを `sub.c` としてメインと同じディレクトリに保存した後、メイン側をコンパイルしてみよ。

リスト 6.5: インクルード確認 (サブ)

```
1 #include <stdio.h>
2 #define INC_STDIO
3
4 int func(void){
5     return 5;
6 }
```

リスト 6.6: インクルード確認 (メイン)

```
1 #include "./sub.c"
2 #ifndef INC_STDIO
3     #include<stdio.h>
4 #endif
5
6 int main(void){
7     int i;
8     for(i=0;i<func();i++){
9         puts("Hello World!");
10    }
11    return 0;
12 }
```

【インクルード文】

インクルード文は、引数に示されたファイルをそのままソースに埋め込む命令である。試しにリスト 6.6 のファイルのコンパイルを `cpp` に変えてみれば、埋め込みである証拠に、2 つのソースが結合されて出力されるだろう。

インクルード文を使う場合、システムが標準で用意してくれているライブラリは `<>` 中にヘッダファイルを記す。一方、自前のソースを埋め込む場合は `"` で囲み、その中に埋め込みたいソースのパスを示す。したがって、普段よく書く `#include <stdio.h>` は `stdio.h` をその部分に埋め込む、という命令である。

【ヘッダファイルの実態】

`stdio.h` を埋め込んだソースを `cpp` で見るなり、Linux の場合 `/usr/include` にある `stdio.h` を見るなりして、その中身を確認してみよう。すると、例えば `puts` の周辺は

```
extern int puts (__const char *__s);
```

となっている。これはプロトタイプ宣言である。他の関数もプロトタイプ宣言が行われている他、必要に応じてマクロや型が定義されていることがわかる。

このように、ヘッダファイルは、マクロや型の定義及びプロトタイプ宣言が行われているファイルであり、関数の実体そのものは別の場所にある¹⁰。実際、自作ヘッダファイルを作る場合も、これを踏襲して、

- ヘッダファイルには共通定義すべきマクロや型と関数プロトタイプのみを記し
- 実体は別のファイルに書いて分割コンパイルの手法を用いて利用する

のが一般的である。

【多重インクルードの防止】

先に出てきた `include` 等を使っていると、同じソースを複数回インクルードしてしまうことがあり、バグの原因となりうる。これを防ぐため、適当なマクロを定義して、それが定義されていなければインクルードする、というように書くのが一般的である。このように、適当なマクロを定義して多重インクルードを防止する手法をインクルードガード (include guard) と呼ぶ。リスト 6.5 およびリスト 6.6 では `INC_STDIO` というマクロを用意し、これが定義されているかどうかでインクルードガードを行なっている。通常、インクルードガードに利用するマクロはヘッダファイルに書かれている (自作する場合書くようにする)。例えば `stdio.h` の場合は `_STDIO_H` というマクロが定義されている。

¹⁰ この、別の場所にある関数の実体そのものを引っ張ってくるのが「リンク」という作業である。ちなみに、リンクで呼び出される側のファイルはバイナリファイルであり、その実装元のソースは通常見ることができない。

6.4.3 分割コンパイルと複数ファイルでのスコープ

インクルードを用いて「はめこみ合成」を行えば、ソースを複数に分けて分割することも可能である。だが、それは単にひとつのソースを切っただけに過ぎない。巨大なプログラム開発だと

- コードを機能別に保守したい。
- 少しコードを書き換えたからと言って全てコンパイルし直すのではなく、コードを書き換えた部分だけを更新したい。

などの要求が出てくるだろう。これらの要求に答えるのが分割コンパイルの手法と、それを支援するツール群¹¹である。ここでは、分割コンパイルの手法と、その利用に必要な複数ファイルのスコープに関して簡潔に述べることにする¹²。

【分割コンパイルの手法】

分割コンパイルは、先に挙げたような要求に答えるための機能であり、分割の単位は人間がわかりやすいよう、関数単位などとなることが多い。ここでは、先にインクルードの解説で用いたプログラムを、分割コンパイル用書きなおして見ることにしよう。

【分割コンパイル】

サブとメインを同じフォルダに入れて、適当な名前をつけて保存せよ。以降の解説では、Sub.c と Main.c として記す。また、共有ヘッダも header.h という名前で、同じフォルダにおいておくこと。

リスト 6.7: 共有ヘッダ (header.h)

```
1 #include<stdio.h>
2 int func(void);
```

リスト 6.8: 分割コンパイル確認 (サブ)

```
1 #include "../header.h"
2
3 int func(void){
4     return 5;
5 }
```

リスト 6.9: 分割コンパイル確認 (メイン)

```
1 #include "../header.h"
2
3 int main(void){
4     int i;
5     for(i=0;i<func();i++){
6         puts("Hello World!");
7     }
8     return 0;
9 }
```

このように、分割されたソースからなる一つのプログラムをコンパイルするときには、

```
gcc Sub.c Main.c
```

¹¹分割コンパイルに用いられるツールとしては、IDE(統合開発環境)は勿論のこと、make や Auto-toolset, Cmake などが知られている。

¹²より細かいことを知りたい場合は「C 言語によるスーパー Linux プログラミング」(飯尾 淳 著, 2011, ソフトバンククリエイティブ) など参照。

のように、全てのソースファイルを指定すれば、通常と同じ結果が得られる。だが、これだけでは、いちいちすべてコンパイルしなければいけない煩わしさは変わっていない。

そこで、次のようにコンパイルを実行する。

```
gcc Sub.c -c
gcc Main.c -c
gcc Main.o Sub.o
```

この、最初の2行の-c オプションをつけたコンパイルにより、結合直前までの作業を終わらせておき、最後の1行のコードで結合する、というような意味になる。このことから、例えばSub.cを書き換えた場合は第1行と第3行のみを実行すれば良い(実際に書き換えて試してみよ)。

また、ここでは共通ヘッダファイルを用いている。これは、分割ファイルをコンパイルする際に面倒なエラーなどが起こらないようにする手法の一つである。例えば、関数funcは、メインプログラムには書かれていない。そのため、単に#include<stdio.h>とただけでは、定義されていないとしてエラーが起こってしまう。そこで、必要なプロトタイプ宣言を記すわけだが、多数のファイルの先頭にいちいちプロトタイプ宣言を書くのは間違いの元なので、ヘッダファイルを作成してインクルードすることにより対処しているのである。

【複数ファイルでのスコープについて】

最後に、複数ファイルでのスコープについて述べておく。このような、複数ファイルでのスコープのことをリンケージ(linkage)と呼ぶ。この内、あるファイル内限定で使えるものは内部リンケージ(internal linkage)を持つといい、他のファイルでも使えるものは外部リンケージ(external linkage)を持つという。

リンケージは関数と変数とで取り扱いが違う(マクロは通常、複数ファイル間のスコープは持たず、必要ならばインクルードを利用する必要がある)ため、個別に述べていく。

関数の場合、通常の宣言では外部リンケージを持つ。したがって、プロトタイプ宣言さえしておけば、他のファイルに書かれている関数を自在に使うことができる。この「自在に使われる」現象を防ぐためには、関数の宣言前にstaticを付ける。このstaticは静的変数のstaticと全く意味が違うので注意せねばならない。

staticの意味

- 変数の前のstaticは静的変数の意味。
- 関数の前のstaticは内部リンケージを持つ関数の意味。

このように、C言語には綴りが同じであるのに意味が違う記号や言葉があるので、注意

せねばならない¹³。

関数はプロトタイプ宣言によって宣言を行い、実体定義を読みに行くが、変数にはプロトタイプ宣言などというものはない。そこで、別ファイルで定義された変数を読みに行く際には、同名同型の変数の宣言の前に `extern` 記憶クラス指定子を付す。なお、この方法で読めるのは大域変数のみであり、局所変数は (そもそもスコープがその関数内に制限されているため) 読むことができない。つまりファイル A でグローバルに `int a=5` などと定義しておき、ファイル B で `extern int a` とすれば、ファイル A で定義した `a` と同じ `a` をファイル B 内で使うことができるが、`a` の定義がローカルだったならば、これは使えないという事である。

変数と関数の複数ファイルにおけるスコープをまとめると、表 6.2 のようになる。

表 6.2: 複数ファイルにおける変数/関数のスコープ

	変数	関数
内部リンケージ	普通に宣言	<code>static</code> 指定子を付ける
外部リンケージ	呼ぶ側で <code>extern</code> 指定子を付ける	プロトタイプ宣言して別で定義

なお、スコープが重複した場合の処理については、1 ファイルの場合と同様、より狭い側が優先される。

¹³これについては「エキスパート C プログラミング」(P.Linden 著, 梅原 系 訳, 1996, ASCII) が詳しい。本講で出てきた `void` というシンボルも引数リストでは「引数がない」、返却値の型では「返却値を返さない」と複数の意味を持つし、`extern` 指定子は関数につけた場合 (冗長な宣言であり、通常付ける必要はないが) 外部リンケージを持つ関数であるという意味になる (変数は別ファイルで定義しているのを呼ぶという意味で、違う意味であることに注意)。

しかし、`extern` や `void` といった例も、`static` ほどの意味の違いはない。`static` については、同書においてわざわざ脚注がつけられ、「なぜこれほどまでに違った意味が持たされたのか？」とまで書かれているほど意味が違う。

本講の要点

本講では、自作関数を導入し、オブジェクトのスコープや寿命について学んだ後、プリプロセッサについても触れた。

自作関数

- 自作関数は似たような処理をまとめるなどの用途で用いられる。
- 自作関数を作成する際には、使用するより前にプロトタイプ宣言を行い、一般には main より後で定義する。
- 自作関数に引数として渡される値はあくまで値のコピーであり、変数そのものではない。それ故、関数内での処理は呼び出し元の変数に影響を及ぼさない。

型の別名定義

- typedef 指定子を用いることによって型を定義することができる。

スコープと寿命

- オブジェクトの有効範囲をスコープといい、通常ファイル内ないし宣言された節内である。
- 外部リンケージを持つ関数は必ずグローバルに宣言されている必要がある。
- 変数が生成されてからメモリ上から取り除かれるまでの期間を変数の寿命と呼ぶ。
- 関数中で static をつけて宣言された変数を静的変数と呼び、プログラム終了までの寿命を持つ。静的変数の初期化は一度しか行われない。
- register をつけて宣言された変数はレジスタ変数と呼ばれ、レジスタ上に優先的に配置される代わりに、アドレスが取得できないなどの制限を受ける。
- 外部で定義された変数を利用する場合は extern 指定子を付す。

プリプロセッサとマクロ

- マクロは、ソース中のあるキーワードを置き換える役目を担い、関数形式マクロとオブジェクト形式マクロに大別される。
- 前処理命令を上手く使うことで、環境に応じてソースを変えるなどの処理を組み込むことができる。

第7講 様々な関数

前講では関数の原理的な部分についてじっくりと学んだ。今度は、活用する方法として、関数の利用の幅を広げていこう。まず、標準ライブラリの利用法として、よく使う関数をいくつか紹介する。次いで、自作関数の幅を広げる上でよく用いられる再帰処理について説明する。また、同様に (使われることはやや少ないが) 自作関数の幅を広げるという意味で可変引数関数の自作方法やインライン関数についても述べる。

7.1 標準ライブラリの利用

C 言語には多くの標準ライブラリがあり、これらを有効に活用することでプログラミングの幅が広がる。ヘッダの詳細については付録を見てもらうとして、ここではよく使う関数や、知っておくと便利な関数に焦点を当てる。

7.1.1 時間計測

【時間計測】

時間のかかる処理 (l.10) の時間を計測してみる。

リスト 7.1: 時間計測の方法

```
1 #include <stdio.h>
2 #include <time.h>
3
4 int main(void){
5     int i,j,k,t;
6     clock_t before, after;
7
8     before = clock();
9
10    for(i=0;i<1024;i++) for(j=0;j<1024;j++) for(k=0;k<1024;k++) t=i*j*k;
11
12    after = clock();
13    printf("%f\n", (double)(after - before) / CLOCKS_PER_SEC);
14    return 0;
15 }
```

C 言語で時間計測をする場合には、時間計測したい直前のポイントと直後のポイントで時間を取り、その時間の差をとって出力を行う。

時間を取る関数は `time.h` に入っており、次のような関数がある。

- `time` 関数：秒単位で時間を取得する (起点は 1970 年 1 月 1 日 00:00:00 UTC(世界標準時))。返却値は `time_t` 型である。
- `clock` 関数：マクロ `CLOCKS_PER_SEC` に対し、 $1.0/\text{CLOCKS_PER_SEC}$ 秒単位で時間を取得する (起点はプロセス開始時)。返却値は `clock_t` 型である。

`time` 関数は精度があまりよくないが、長時間計れる場合が多い。一方、`clock` 関数は精度こそいいものの、`clock_t` 型の実装によってはすぐにオーバーフローしてしまう¹²。この為、必要に応じて両方を使い分ける必要がある。

【time 関数】

`time` 関数を用いる場合は、

```
time_t before, after;
```

のように宣言した後、適切な場所で `time` 関数を呼び出して直前と直後の時刻を取得し、

```
difftime(after, before)
```

により時間の差を計算することで、`double` 型で時間の差が出る。`difftime` は処理系によって変わる `time_t` 型の定義に対して適切に設定された関数であるので、単に引き算をするよりもこちらを使うほうが推奨される。

【clock 関数】

`clock` 関数を用いる場合は、まさしくリスト 7.1 のように

```
clock_t before, after;
```

と宣言した後、適切な場所で `clock` 関数を呼び出して直前と直後の CPU 時刻を取得し、

```
(double)(after-before)/CLOCKS_PER_SEC
```

¹例えば、`CLOCKS_PER_SEC` が 10^6 であり (つまり、1 マイクロ秒単位とし)、`clock_t` 型が 32bit 符号なし整数型で実装されていたとすれば、71 分 30 秒程度までしか計れない。32bit 符号あり整数型ならばその半分である。筆者の環境では 1 マイクロ秒単位で `clock_t` は 8 バイトだったので、符号付き整数型を仮定しても 2924 世紀は計れるとわかる。

²`time_t` 型にも同様のオーバーフローの問題がある。32bit 符号付き整数型が一般の実装であるので、これを仮定すると、2038 年にオーバーフローが起こる (**2038 年問題**)。しかし、現在では 64bit の実装が増えてきているので、この問題が起こる 2038 年には問題は軽減されているかもしれない。なお、64bit 符号付き整数型の場合、およそ西暦 3000 億年までオーバーフローしないので大丈夫である。

により、double 型・秒単位での時間を計測することができる。一般に、time 関数よりもこちらのほうが精度がよく、また clock_t が 64bit であれば³そう簡単にオーバーフローしないので、普段はこちらを使えば良い。32bit 環境で、30 分を超えるであろう処理を計測する場合に限り、time 関数を用いて計測を行えば良い。

なお、ここでは時間の計測に話を絞ったが、time 関数は本来カレンダー時刻を取得するための関数である。clock の方が計測に向いているからと言って、不要な過去の関数になっているわけではないので注意されたい。

7.1.2 文字の分類に関する関数

ASCII コードかどうかわからない環境も含めて文字を扱う場合、その文字が数字かどうか、英小文字かどうかなどを判定するのは煩わしいことになる。これらを解決してくれるのが ctype.h⁴に収録されている、文字の分類に関する関数である。

【大文字/小文字の判定】

入力される文字が大文字か小文字かを判定するプログラムを作成する。アルファベットでない文字が入力された場合、その旨を出力する。

【解説】

文字を 1 文字入力するには getchar 関数を用いた。
文字の分類には文字コードを使わず、ctype.h をインクルードしたその中にある関数を用いた。これにより、プログラムの移植性が向上する。

リスト 7.2: 文字の判定

```
1 #include<stdio.h>
2 #include<ctype.h>
3
4 int main(void){
5     char ch;
6     ch=getchar();
7     if(isupper(ch)){
8         puts("Large");
9     }else if(islower(ch)){
10        puts("Small");
11    }else{
12        puts("Not_␣Alphabet");
13    }
14    return 0;
15 }
```

【is～～関数】

ctype.h の関数は、後に説明する toupper 関数と tolower 関数を除いて is～～の形式⁵である。これらは何れも、引数が～～の種別に適合していれば真値を、適合していなければ偽値を返す関数である。例えば、リスト 7.2 では isupper 関数と islower 関数を使っているが、isupper 関数は引数が upper=大文字であるかどうか、islower 関数は引数が lower=小文字であるかどうかを判定する関数である。

³これをチェックするには time.h をインクルードした上で sizeof(clock_t) を出力すればよい。

⁴ctype は Character TYPE の略である。

⁵これは、[is+種別] の名前であり、is は be 動詞の is である。

以下に、is〜〜関数のうち、よく使われるものを記す(全て見たい場合は付録参照)。

is〜〜〜系関数

- isalnum:半角英数字判定
- isalpha:アルファベット判定
- islower:英小文字判定
- isupper:英大文字判定
- isdigit:数字判定

【大/小文字変換】

ctype.h に入っている、is〜〜〜でない関数はただ二つ、toupper 関数と tolower 関数だけである⁶。この2つの関数は、それぞれ英小文字/英大文字が引数に取られた時のみに機能し、それに対応した英大文字/英小文字を int で返す。それ以外の場合は、文字コードをそのまま、やはり int で返す。従って

```
isupper(c)?tolower(c):toupper(c)
```

などとすれば、大文字と小文字を変換することができる。

【ctype.h の引数と返却値】

実は、先に書いた返却値が int であるという仕様は、toupper/tolower 関数に限ったことではなく、ctype.h 全ての関数に言えることである。

ctype.h の引数と返却値

ctype.h の関数の引数や返却値は全て int 型である。

実際に使用する場合は char 型が整数型であるので暗黙の型変換が起こるため問題ないだろうが、時として「引数の型が違う」等の警告などを出す場合があるので書いておいた。

7.1.3 プログラムの終了

例外処理などの際に、プログラムを終了させたい場合がある。main 関数であればその場で return 文を用いればよいが、main 関数以外の関数からプログラムを終了させたい場合もあるだろう。これを実現するための関数が stdlib.h⁷に入っている。

exit 関数と abort 関数である。

⁶これらは、to+upper,to+lower という意味である。

⁷STanDard LiBrary の略。

【exit 関数】

exit 関数はプログラムを正常終了させる関数である。

exit 関数

exit 関数は

```
exit(return_val)
```

の形式で用いる。この時、引数に与えた値がプログラム (=main 関数) の返却値となる。

プログラムを正常終了させた場合は、その後に OS が適切な処理を施してくれる場合が多く、通常この関数を用いて終了すれば、main 関数を `return 0` で終了させた場合とほぼ同様の操作をおこなってくれる⁸。

【abort 関数】

exit は正常終了であったが、異常終了させる関数もある。それが abort 関数である。

abort 関数

abort 関数は

```
abort()
```

の形式で用い、これによりプログラムが異常終了させる。

この関数は、OS 等のホスト環境に対して、そのプログラムが異常終了したことを示してプログラムを終了させる。この関数では exit 関数のような終了処理は行われませんが、OS が異常終了の際の処理を施してくれるため、主としてデバッグの際に便利である。または、ユーザーが変な操作をした場合に、その警告を促すなどの用途でも用いられる。

【終了関連マクロ】

通常、プログラムの終了の成否は各々 0 か非 0 に対応しており、0 が正常終了である。だが、環境によっては必ずしも 0/非 0 により正常終了/異常終了が区別されるとは限らない⁹。この環境依存の問題を解決するために、`stdlib.h` において終了関連のマクロが定義さ

⁸一応、処理を具体的に示しておく、

- 後で学ぶファイルの操作において、ファイルが開かれていれば自動で閉じてくれる。
- 同じくファイルの操作において、出力ストリームをフラッシュしてくれる。
- 一時ファイルを削除する

などの処理を行う。これ以外の処理を使う場合には `atexit` 関数というものを使うのだが、関数ポインタを理解しないと使いこなせないなのでここでは説明しない。

⁹ここの main 関数の終了には 0 を返すものとしてきた。実際、大抵の PC 環境では `return 0` としておけば正常終了になる。だが、組み込みプログラミングなどにおいて特別な処理を行う場合には、変わってくる場合がある。とはいえ、特殊なデバイスを考えない限り、これらのマクロは利用しなくても良いだろう (実際、筆者の Linux 環境でこれらのマクロの定義を見てみると、0/1 と定義されていた)。移植性を十分高めたいと考える場合に限り、これらを利用すると良い。

れている。

終了に関連したマクロ

EXIT_SUCCESS 正常終了を示す。

EXIT_FAILURE 異常終了を示す。

これらのマクロを利用すれば、特殊な処理系において終了処理を記述する場合でも、適切な終了方法を選択してプログラムを終了させられるようになり、移植性に富む。

7.1.4 型に関する標準ライブラリ

変数の型について思い出すと、変数の型において「何型が何 bit である」という正確な定義は殆ど無かっただろう。勿論、sizeof 演算子を用いて大きさを取得すればその型についての性質はわかるだろうが、計算機イプシロンなどいちいち計算して出すのは面倒である。そこで、変数の型を扱うために必要な情報が書かれたヘッダがいくつか用意されている。ここでは、これらについて簡単に説明する。

【float.h と limits.h】

float.h は浮動小数点数関連の、limits.h は整数関連の情報を記録したヘッダであり、マクロのみが書かれている。詳細は付録を参照されたいが、これらのヘッダにおけるマクロの命名規則のみ説明しておくこととする。

これらのヘッダにおいて、全てのマクロは

(型名)_(属性名)

の形式で書かれている。例えば、ULONG_MAX というマクロは unsigned long の最大値という事が読み取れるだろう。

これらのマクロは型の様子を知りたい場合の他、「十分大きい値を代入したい」「誤差を上手く処理したい」などの場合に利用できる。

【fenv.h による例外処理】

fenv.h¹⁰は浮動小数点環境について扱うヘッダで、C99 によって新たに定義されたヘッダである。これは、浮動小数点演算において出てくる例外 (exception) について処理する (例外処理 (exception handling)) など、浮動小数点絡みの細かな部分の扱いに用いる。以下、例外処理に的を絞って記述する。

例外とは、プログラムが処理を実行している途中で、コンピュータが処理できない何らかの異常が発生することを言う。異常と一口に言っても内容は様々である。

¹⁰Float ENVironment の略である。

例外の具体例

- ファイルが開けなかった
- 入力形式がおかしい
- メモリを確保できなかった
- メモリの変な場所にアクセスした
- 許されない演算を行った (演算例外)

このように、例外 (異常) として考えられることを列挙していけば枚挙に暇がない。これらの例外が現れた場合、プログラムは通常適切な事後処理を施して (異常) 終了するように書かれる。C 言語では例外を処理するための構文がない¹¹ため、プログラマが考えて例外を検知するようにプログラムを書かなければならない。

fenv.h には先に挙げたような例外のうち、浮動小数点数演算例外について処理する為の関数が揃っている。これを用いれば、0 割などの例外処理を読みやすく実現できる。

【0 割例外処理を含む除算】

2 数の除算を行う処理について、0 割例外を用いて例外処理を行う。

【解説】

l.7 で 0 割例外を発生させておき (0 割例外が起きていることを示しておき)、それを l.8 で受けて処理を行なっている。なお、gcc でコンパイルする際には -lm オプションが必要である。

リスト 7.3: 0 割例外処理

```
1 #include<stdio.h>
2 #include<fenv.h>
3
4 int main(void){
5     double x,y;
6     scanf("%lf_%lf",&x,&y);
7     if(y==0) feraiseexcept(FE_DIVBYZERO);
8     if(fetestexcept(FE_DIVBYZERO)) puts("Exception!");
9     else printf("%f/%f=%f\n",x,y,x/y);
10    return 0;
11 }
```

リスト 7.3 のように、例外が起きていることを feraiseexcept 関数を用いて知らしめておき、fetestexcept 関数によって「ある例外が発生しているかどうか」を調べ、例外発生時

¹¹Java や C++ といった比較的新しい言語では try~catch などの例外処理用構文がある。

の処理を書くことで例外処理を行える。この書き方を採用することによって例外処理が明確になり、プログラムが読みやすくなる。簡単なプログラムの場合には必要ないが、大規模プログラムの場合などには上手く利用すると良い。

【inttypes.h と stdint.h】

整数について扱うヘッダとして、C99 で新たに追加されたのが `inttypes.h` と `stdint.h` の 2 つのヘッダである。

`int` 型は何 bit か、規格には書かれていない。この為、`int` 型を使った場合に、環境によって齟齬が生じる可能性がある。これらを解決するのが `stdint.h` の役目である。

マクロの一覧は例によって付録を参照してもらうこととして、ここでは簡単な使い方を紹介しておく。もしも、厳密に 64bit の符号付き整数型を使いたい場合 (環境依存性を無くしたい場合)、

```
#include<stdint.h>
```

として `stdint.h` をインクルードした後

```
int64_t a,b,c;
```

などのように `int64_t` という名前の型を使えば、これは厳密に 64bit の符号付き整数型を表してくれる。この 64 の部分を 8,16,32 に変更することでビット数を変えることができる。この他、「その環境で使える最大の符号なし整数型を使いたい」というような場合には `uintmax_t` 型などを使うことができる。

`stdint.h` には、これらの型を使う場合の属性関連のマクロも定義されており、特に整数型の大きさに注意を払ったプログラムを書く場合に利用できる。

しかし、ここで注意しなければならないのは、出力などの際である。`printf` 関数を用いて先に書いたような型の変数を出力しようとした時、どのような書式指定子を用いればよいだろうか？あるいは、絶対値を計算する場合、`abs`,`labs`,`llabs` の 3 種類があるわけだが、これらをどのように使い分ければよいだろうか？これらの問題を解決してくれるのが `inttypes.h` である。

書式指定子は、表 7.1 に示すような形式で記す。ここで、各形式における ? は実際には書式を示す 1 文字になり、* は型の幅になる。すなわち、`uint64_t` 型をそのまま出力したいのであれば `PRId64` になるし、`intmax_t` 型を 10 進表記でそのまま入力したい場合には `SCNdMAX` となる。

ここで、注意して欲しいのは、これらを書式指定文字として用いる場合には、% の後にダブルクォーテーション (") で囲んで書かなければならないという事である¹²。例えば、先の例で言えば

```
printf("%"PRIu64"\n",var1);
scanf("%"SCNdMAX",&var2);
```

¹² マクロの展開結果を見てみると、括らなければならない理由がよくわかる。

表 7.1: 書式指定子マクロ (記号は本文参照)

型名	printf 系関数	scanf 系関数
<code>int*_t</code>	<code>PRI?*</code>	<code>SCN?*</code>
<code>int_least*_t</code>	<code>PRI?LEAST*</code>	<code>SCN?LEAST*</code>
<code>int_fast*_t</code>	<code>PRI?FAST*</code>	<code>SCN?FAST*</code>
<code>intmax_t</code>	<code>PRI?MAX</code>	<code>SCN?MAX</code>
<code>intptr_t</code>	<code>PRI?PTR</code>	<code>SCN?PTR</code>

のようになる。

`abs` 関数などについては、`intmax_t` 型向けの関数が `inttypes.h` 内に定義されており、普通の関数と同様に用いれば良い。一例を示しておこう。

【絶対値の出力】

コンパイル環境において扱える最大の整数範囲で、絶対値を出力するプログラムを作成する。

【解説】

ここまでに学んだ文法を用いて、`intmax_t` 型の整数を宣言し、それについて入力・絶対値・出力の各処理を書けば良い。

リスト 7.4: 最大範囲の絶対値

```

1 #include<stdio.h>
2 #include<stdint.h>
3 #include<inttypes.h>
4
5 int main(void){
6     intmax_t num,ans;
7     scanf("%"SCNdMAX",&num);
8     ans=imaxabs(num);
9     printf("%"PRIuMAX"\n",ans);
10    return 0;
11 }
```

7.1.5 代替綴りとトライグラフ

C 言語は日本語や英語にとどまらず、世界中で広く使われている。この際、環境によっては記号が足りない場合などがあり、その場合の代替記法を用いなければならない。これらの代替記法を提供するのが `iso646.h` とトライグラフである。この2つの記法を採用することにより、ISO/IEC646 規格において規定された文字コードのみのソースを書くことができる。通常は必要ないが、複数の国で使うようなソースを書く場合には出てくることもあるので紹介しておく (とはいえ、主要な国はこれを使わなくても大丈夫なのだが)。

【代替綴りと `iso646.h`】

`iso646.h` は記号の代替表記を定義するヘッダである。このヘッダは C95 で追加されたヘッダであり、記号が使えない場合だけに限らず使うことができる。

一例として、ハット (^) 記号を考えよう。これは C 言語では XOR 演算の演算子であるが、他の言語では累乗の演算子となっている場合があり、直感的に合わないという場合がある。このようなときには

```
#include<iso646.h>
```

と iso646.h をインクルードしておいて

```
a xor b
```

などと、代替表記をすれば直感にもわかりやすくなるだろう。

【トライグラフ】

トライグラフ (trigraph) ないし **3 文字表記**は、?? の後に記号を一文字加えることで他の記号を表すという代替表記である¹³。3 文字表記は表 7.2 に示す全 9 種類である¹⁴。

表 7.2: トライグラフの一覧

トライグラフ	記号	トライグラフ	記号	トライグラフ	記号
??=	#	??/	\	??<	{
??([??'	^	??>	}
??)]	??!		??-	~

この表記を用いると、例えば、

```
printf("How are you???/n");
```

は

```
printf("How are you?\n");
```

と展開される。日本語環境でこれを使うことはまずないだろうが、逆に??と連続して書いてしまって意図しないでトライグラフを用いてしまうことがある (とはいえ、実際にはトライグラフの有無をコンパイルオプションで指定できるが) ため、ここで簡単に紹介した。

7.1.6 man コマンド

Unix(Linux) ではこれらの関数を簡単に調べるために man コマンド¹⁵があり

```
man 関数名
```

とすることで、C の関数の用途を調べることができる。

これによって開かれた画面では上下のカーソルキーでページ送りできる他、終了時には q キーを押せば良い。

ここまでにいくつかの関数を紹介し、また、今後も多数の関数が出てくるがこれらを逐一覚える必要は全くない。使いたい時に、本書の付録を引いて用途に合いそうな関数を探し、見つかったらそれを man コマンドで調べればよいだけである。

¹³この 3 文字表記で?が使われるため、?そのものはエスケープシーケンス\?となっている。

¹⁴iso646 において、文字コードによっては表せない ASCII 文字は 10 種類あり、トライグラフはひとつ足りていないように見える。これは、記号\$を C コード中で制御文字として使用することがないので、省いたためである。

¹⁵MANual コマンドの略である。

7.2 再帰関数

再帰 (recursive) は自分自身に再度帰着させるという概念である。これはプログラミングに特有のものではないが、プログラミングにおいても重要な役割を果たす。再帰は確かにその言葉通り、自分自身に再度帰着させるという意味であるが、これではわかりにくい。そこで、「例示は理解の試金石」ということで、具体例から再帰処理を理解していこう。

階乗を再帰的に定義することを考える。やや冗長かもしれないが、次の定義は階乗の定義になっている。

- 0 の階乗は 1 である。
- n の階乗は、 $n-1$ の階乗に対して n を乗じたものをいう。但し、 n は自然数とする。

これに従って「3 の階乗」を理解していくと

1. 3 の階乗は、2 の階乗に対して 3 を乗じたものである。
2. 2 の階乗は、1 の階乗に対して 2 を乗じたものである。
3. 1 の階乗は、0 の階乗に対して 1 を乗じたものである。
4. 0 の階乗は 1 である。
5. 従って、逆にたどれば、3 の階乗が求められる。

という順になる。これは階乗の定義に階乗そのものを用いている。このようにある処理の中で自分自身を呼び出すものが再帰処理である。

再帰処理を含む関数を再帰関数 (recursive function) という。以下、この実装方法について学んでいこう。

【再帰を用いた累乗】

再帰関数を用いて累乗 (整数乗) を行う関数を実装する。

つまり $0^0 = 1$ と定義した。

- 負数乗は逆数の自然数乗をとることと計算した。

【解説】

- 自然数乗は繰り返し 2 乗法を用いれば速いが、ここでは愚直に掛け算を行った。
- 0 乗は全ての数について 1 とした。

リスト 7.5: 整数乗を行う関数

```
1 double powint(double f,int n){
2     double ret=1;
3     if(n<0) return powint(1/f,-n);
4     while(n-->0) ret*=f;
5     return ret;
6 }
```

n が非負整数であれば、このプログラムはこれまでの文法で学んだ事で理解できる。このことからわかるとおり、この関数において重要なのは 1.3 の処理である。

【負数の場合の処理】

リスト 7.5 の 1.3 の処理は、if から明らかに n が負の数の場合に実行される。すると、今度は `powint(1/f, -n)` として、また `powint` 関数が呼ばれる。すると、今度は $1/f$ を新たな f (以下 f') として、 $-n$ を新たな n (以下 n') として `powint` の処理が実行される。 $n' > 0$ であるので、これは f' の n' 乗である。この計算結果を中で呼ばれた `powint(f', n'` が引数の方の `powint)` が返す。すると、この結果を元々の `powint(f, n` が引数である `powint)` も返す。これにより、計算が実行されている。

ここまでの説明よりわかるとおり、`powint` 関数は内部的に自分自身=`powint` 関数を呼び出す処理を行なっている。このような関数が再帰関数であり、用いる場合はリスト 7.5 のように (他の関数から呼び出すのと同様に) 自分自身を書けば良い。(なお、必ずしも `return` の値として書く必要はない)。

7.2.1 再帰関数のメモリ上での動作

リスト 7.5 での再帰呼び出し回数は、高々 1 回であった。今度は、もっと多くの回数の呼び出しを行う再帰処理を見てみよう。

【アッカーマン関数】

次の式により定義されるアッカーマン関数を実装する。

$$\text{Ack}(m, n) = \begin{cases} n + 1 & (m = 0) \\ \text{Ack}(m - 1, n) & (n = 0) \\ \text{Ack}(m - 1, \text{Ack}(m, n - 1)) & \end{cases}$$

リスト 7.6: アッカーマン関数

```
1 typedef unsigned int u_int;
2 u_int ack(u_int m, u_int n){
3     if(m==0) return n+1;
4     if(n==0) return ack(m-1,1);
5     return ack(m-1,ack(m,n-1));
6 }
```

この関数において、4 1 を渡してみると、計算に随分時間がかかった他、5 5 などを渡すと Segmentation fault (メモリ上のアクセス禁止領域にアクセスした時に出る警告) と出てプログラムが終了してしまった。これは一体なぜなのだろうか。再帰関数の動作を理解し、この疑問を解明しよう。

【再帰呼び出しの様子】

再帰呼び出しを行う場合、その呼び出した側の関数はどうなっているのだろうか。また、呼び出された側の関数はどうなっているのだろうか。

通常の関数の場合、別の関数を呼び出したら、その処理が終わるまで待っており、その処理が終われば次第続きの処理を行う。これは再帰関数でも同じである。再帰呼び出しを行った場合、一連の再帰処理が全て終わるまで待ち続け、終わった後に処理が続けられる。その間関数は開きっぱなしである。つまり、複数回の再帰呼び出しを行った場合、そ

の最も根元にある関数は、再帰が全て終わるまでずっと待ち状態を続けているのである。再帰が深くなった場合、このように待ち続けている関数が増えてくるし、呼び出し回数も馬鹿にならない。例えば、アッカーマン関数において、3 3 を渡せば関数の呼び出し回数は 2432 回になり、4 1 を渡すと 2862984010 回もの呼び出しが行われることになる。ここから、仮に 1 億回の呼び出し処理を 1 秒でこなすとしても、4 1 を渡した場合には 30 秒弱かかる計算になってしまう。

このように再帰処理で計算を行う場合、計算回数 (呼び出し回数) が爆発的に増えてしまう場合がある。これは、一度計算した値についても、再度 (多数の呼び出しを含む再帰処理の) 計算を行ってしまうからである。例えばアッカーマン関数において 4 1 を渡した場合、Ack(0, 1) は 65512 回も計算されている。これは明らかに無駄である。そこで、後で学ぶ配列などを用いて、一度計算した値をメモしておき、それを用いて再帰を高速化する **メモ化再帰** (Memoization ないし Memoized recursive) の方法が知られている。

【再帰処理とメモリ】

先に書いたとおり、再帰では呼び出し側の関数はずっと待ち続けている。これは、実行されている関数が常にメモリに居座るという事である。

自作関数を呼び出した場合、メモリ上のスタック領域と呼ばれる場所に関数が展開される。そして、関数が終了すれば通常その場所は明け渡される。だが、再帰処理の場合は、明け渡すことなく多くの呼び出しを行なってしまうことになりかねない。これが一定回数を超えると、スタック領域の許容量に達してしまい、「これ以上メモリがない!」という状況になってしまう。だが、再帰処理の終了の条件を満たしていないため、プログラムは更にメモリを取ろうとしてしまう。そこで、メモリ上のアクセス禁止領域に踏み込んでしまい、Segmentation fault になってしまったのである。このように、スタック領域からあふれるエラーをスタックオーバーフロー (stack overflow) と呼び、再帰処理の場合には特に注意を払わなければならない。

7.2.2 再帰関数の注意点

再帰処理は先に書いたメモ化再帰の技法なども含めて、簡単にソースを書くために、あるいは読みやすいソースを書くために必要である。構造化定理に立ち返れば、再帰処理は必ずしも必要ではなく、反復処理で書き換えられるはずだが、実際に反復処理に書き換えようとするとな大な労力を要するようなものも少なからずある。

だが、再帰処理は反復処理ではない。これは、先に書いたメモリ上での動作を見れば明らかであろう。反復処理であれば、無限ループを書いたと言っても「終わらない」だけで済むが、無限再帰を組んでしまうとメモリが不足し、最悪の場合 OS などに異常をきたしてしまう場合もある (通常は OS の安全機能が働くが、穴がないとは限らないため)。

以上の点から、再帰処理を行う際には、とりわけメモリに注意しなければならないことがわかる。以下に、その注意点をまとめておこう。

再帰処理を使う際の注意点

- 再帰処理を行う場合は充分少ない回数で再帰が終了することを保証して作成しなければならない。
- 関数 A から関数 B を呼び出し、関数 B から関数 A を呼び出すような再帰処理 (複合再帰処理) は再帰処理を行なっていることがわかりづらくバグの元なので作らないほうが良い。
- 再帰処理を行う際には、スタック領域でメモリを取るなので、ある程度メモリを空けておいて実行する方が良い。また、必要に応じてスタックサイズを大きくする。
- 単純に再帰をするのではなく、メモ化するなどの工夫を凝らして、再帰処理の呼び出し回数を低減させなければ、計算回数が増えやすい。

7.3 可変引数関数

printf 関数や scanf 関数は引数の個数が不定であり、第 1 引数の文字列リテラルによってその引数の個数が決まる。このように、引数の個数が不明であるような関数のことを可変引数関数 (variable arguments function) と呼ぶ。本節ではこの可変引数関数の作り方について説明を行う。

可変引数関数には大きく分けて 2 種類の作り方がある。この 2 種類は、引数の個数を確定する手段が異なっている他は、同様に実装できる。可変引数の個数を決定するためには

- 引数の最後にターミネータ (終端を示す値) を入れる。
- 可変引数以前の引数に、可変引数の個数を情報として含む引数を与える。

の 2 種類がある¹⁶。これは後に扱う文字列と同様である。つまり、文字列は C 言語では NUL(NULL 文字) というターミネータを用いるが、Pascal などの言語では、長さを示す値を文字列に付加して格納しているのである (詳しくは文字列の解説を参照のこと)。以下、この 2 種類と、可変引数リストを引数に取る関数について説明を行う。

7.3.1 stdarg.h の利用

可変引数関数では、1 個以上の引数の個数を任意に変えることができる。これは、stdarg.h¹⁷ の宣言によるものである。

可変引数関数を用いる場合の手順は次のようになる。

¹⁶この 2 つの方法は、配列引数の関数で配列のサイズを知りたい場合などにも共通する。

¹⁷STanDard ARGument の略である。

可変引数関数を用いるための手順

1. `stdarg.h` をインクルードしておく。
2. 関数の宣言の際に、可変引数部を `func(par1, par2, ...)` のように、`...` を用いて宣言する。
3. 関数内で `va_list` 型変数を宣言する。
4. 関数内で `va_list` 型変数を `va_start(va_list 型変数, 可変引数の直前の引数)` として、`va_list` 型変数を初期化する。
5. 引数を前から順に `va_arg(va_list 型変数, 可変引数の型)` により呼び出す。
6. 利用し終わったら `va_end(va_list 型変数)` により、後始末を行う。

ここで、型 `va_list` は可変引数関数の引数一覧を示す型である。この手順は引数の個数の取得方法によらないので、一連の方法として理解しておくが良い。実例は後の引数の個数の取得方法別の解説で見せる。

可変引数関数のプロトタイプ宣言についても、通常関数と同じように扱うことができる。一方、呼び出しを行う場合は `printf` 関数のように、コンマ区切りで可変引数を列挙すれば良い。

7.3.2 ターミネータを用いる方法

【総和計算関数 (ターミネータ利用)】

`int` 型の引数の和を計算する可変引数関数 `sum` を作成する。この時、0 という要素が出るまで足し続けるものとする。

【解説】

仕様から、可変引数のターミネータに 0 を用いる。また、`va_start()` を利用するためには可変引数の直前の引数が必要になるため、ここでは `i1` という変数を宣言しておいた。

リスト 7.7: ターミネータ利用による総和計算関数

```
1 #include <stdarg.h>
2
3 int sum(int i1, ...) {
4     va_list vars;
5     int s=i1,n;
6     va_start(vars, i1);
7     do{
8         n=va_arg(vars,int);
9         s+=n;
10    }while(n!=0);
11    va_end(vars);
12    return s;
13 }
```

以下、リスト 7.7 の解説を行う。

1.3 `...` の宣言により、可変引数関数であることを示す。

- l.4 可変引数リストを格納するための変数 `vars` を宣言する。
- l.6 可変引数の始まりを、`va_start` により設定する (可変引数リストの初期化)。
- l.8 `va_arg` を用いて、可変引数リスト `vars` の「次の引数」を `int` 型で読み込む。
- l.10 `do` 以下の処理を、読み込んだ可変引数が 0 になるまで繰り返す。
- l.11 用いた可変引数リストについて、`va_end` を用いて後始末を行う。

このように、可変引数は、ひとまずリストに全て格納しておいて、随時 `va_arg` により引数を呼び出してくる形になる。これらのリストの扱いを行うための関数が格納されているのが `stdarg.h` である。

この実装から明らかなように、ターミネータの場合、読み取った値が終端条件であるかどうかを確かめることで引数の終了を確かめられる。このターミネータは自分で決め、終了処理も自分でやらなければならない (可変引数が尽きたからと言って勝手に動作が終わるわけではない) 点に注意しなければならない。

7.3.3 可変引数より前の引数で個数を示す方法

`printf` 関数などを見ると明らかなように、可変引数より前の引数から可変引数の個数が読み取れるならば、それを用いて可変引数処理を行うことができる。ここでは、直接個数を指定する形の総和を作ってみよう。

【総和計算関数 (個数情報利用)】

`int` 型の引数の和を計算する可変引数関数 `sum` を作成する。ただし、第 1 引数として総和を取りたい値の個数が与えられる。

【解説】

今度は、第 1 引数に総和をとりたい可変引数の個数が与えられるように総和関数を作りなおした。これにより、`va_start()` を利用するための直前の引数は、この個数を与える引数となる

ように変更した。

リスト 7.8: 個数情報利用による総和計算関数

```
1 #include <stdarg.h>
2
3 int sum(int num,...){
4     int i,s=0;
5     va_list vars;
6     va_start(vars,num);
7     for(i=0;i<num;i++){
8         s+=va_arg(vars,int);
9     }
10    va_end(vars);
11    return s;
}
```

リスト 7.8 を見てもらえば明らかなように、可変引数の利用方法自体は先のターミネータを利用したものと何ら変わらない。ただ、ループの処理が変わっているだけである。

個数の情報を引数から読み取る方法を応用すれば、可変引数のうち前いくつかは用途 A で用い、それ以降は用途 B で用いるなどの方法を取ることもできる。また、このように可変引数にいくつもの意味を混在させる場合など、ターミネータと個数の両方法を混在させることも可能である。いずれにせよ、可変引数関数は引数が尽きたからと言って勝手に動作が終わるわけではないため、終了処理を適切に書いてやる必要がある。

7.3.4 可変引数を一括処理する関数

printf 関数は可変引数関数の代表例であるが、この可変引数部を `va_list` に変えた関数として `vprintf` 関数がある¹⁸。これら `printf/scanf` 系関数は `stdio.h` に入っている。この利用例を見てみよう。

【改行される printf】

最後に自動的に改行される `printf` 関数として、`lineprintf` 関数を作る。

【解説】

第 1 引数 `format` は見慣れないと思うが、これは書式文字列を指定するものである。この詳しい意味については後の章で学ぶ文字列やポインタについて学んだ後わかるだろう。これと出力するとき、可変引数を、`va_list` 型変

数で与えるのが `vprintf` 関数である。

リスト 7.9: 改行付き `printf`

```
1 #include<stdio.h>
2 #include<stdarg.h>
3
4 int lineprintf(char *format, ...) {
5     va_list ap;
6     va_start(ap, format);
7     vprintf(format, ap);
8     putchar('\n');
9     va_end(ap);
10    return 0;
11 }
```

リスト 7.9 の 1.7 にあるように、`vprintf` 関数 (及びそれに類する関数) では、`va_list` 型変数を引数にとり、それに応じた処理を行う。この場合は、`va_start` により可変引数リストを初期化した後、それらの `va_list` を利用する関数を用い、`va_end` により後始末を行う。

7.4 インライン関数

関数マクロは、単に式を展開するだけであるので処理が速いが、型チェックがないのが難点であった。そこで、型チェックがあり、速度面で十分速い関数を作りたいと考えた。これがインライン関数 (inline function) である。インライン関数は、コンパイル時にインライン展開と呼ばれる技法を用いて最適化される関数で、通常の関数に比べて呼び出しのオーバーヘッドがなく、速くなるという利点がある。

インライン関数は C99 において新たに登場した関数の定義方法であり、次のように定義する。

¹⁸C99 では `scanf` 関数の可変引数部を `va_list` に変えた `vscanf` 関数ができた

インライン関数の定義方法

ある関数をインライン関数として定義する場合、

`inline` 通常の関数定義

のように、関数の定義の前に `inline` キーワードを付す。

この定義方法によって定義された関数は、コンパイル時にインライン展開を用いて可能な限り速い関数になるように最適化される。实例として、Euclid の互除法により 2 数の最大公約数を求める関数を `inline` 関数で定義する場合

```
inline int euclid(int m,int n){
    return n?euclid(n,m%n):m;
}
```

のようになる。

インライン関数には、次のような原則があり、これに則って利用する必要がある。

インライン関数の原則

- `inline` キーワードをつけた関数は、コンパイル時に、そのコンパイル単位内で見える位置に定義が書かれている必要がある。
- `inline` キーワードをつけた関数に対し、これにインライン展開の技法を適用して最適化するかどうかはコンパイラ側の自由である。すなわち、`inline` キーワードをつけたからと言って必ずインライン展開されるとは限らない^a。
- `inline` キーワードをつけた関数が、実際にインライン展開される場合、その関数の意味が改変されたり損なわれたりすることはない。
- インライン関数の展開は、関数形式マクロのそれに似ているが、引数の型チェックがあることや、引数に式を入れても値が評価されて渡されるという点が違う(=通常の関数と同様)。

^aこの点は `register` 記憶クラス指定子に似ている。

先の制約「定義が可視である」ことから、インライン関数を定義する場合はある一つのファイル内でその関数が完結するように書くのが一般的である¹⁹。

¹⁹外部参照定義も適切に行えば可能であるが、ここでは触れない。

本講の要点

本講では、標準関数の使い方と、自作関数の応用的な使い方について学んだ。

標準関数

- 時間計測には `time.h` の `time` 関数や `clock` 関数を用いる。
- 文字コードに依存しない文字処理には `ctype.h` を用いると良い。
- `main` 関数以外からプログラムを終了させる場合、正常終了を行う `exit` 関数や異常終了を行う `abort` 関数を用いる。
- 標準ライブラリには型について適切に扱うためのライブラリがあり、これを用いて移植性の高いプログラムを書くことができる。
- ASCII コードの文字のうち、環境によっては存在しない文字を書くために、代替綴りの方法が定義されている。
- `man` コマンドを用いれば関数のマニュアルを読むことができる。

再帰関数

- 再帰関数とは、自身をその中で呼び出す関数のことである。
- 再帰処理を行う場合は充分少ない回数で再帰が終了することを保証して作成しなければならない。

可変引数関数

- 可変引数関数の実装の際には `stdarg.h` をインクルードした上で、可変引数リストを格納する変数を宣言し、その変数を初期化し、次いで順に可変引数を型を指定して呼びだし、最後に後始末を行う。
- 可変引数関数を実装する際には、可変引数の個数を示す引数を先行する引数に入れるか、ターミネータを用いて、引数の終了を示さなければならない。

インライン関数

- インライン関数は、できるだけインライン展開と言われる技法を用いて展開されるように指定された関数である。
- インライン関数は、`inline` キーワードをつけて宣言し、コンパイル時に見える位置で定義する。

第8講 派生型(1) 静的配列と文字列

今回から4講にわたり、変数のより応用的な利用法として「派生型」について学んでいく。厳密には、ここまでに学んだ関数も「関数型」と呼ばれる派生型であるため、本書の第6～12講は派生型とその扱い方を学んでいると言える。とりわけ、タイトルに「派生型」と付した講は、「いかにも変数を応用しています」という感のある内容である。

今回は、派生型のうち最も基本的と言える配列(array)と、その応用である文字列(string)について学んでいく。

8.1 基本型と派生型

まず、派生型とは何かを見ていくことにしよう。なお、本節は抽象概念の説明であるので、理解し難いと思ったらひとまず飛ばして次節に進んで良い。

派生型(derived type)とは、これまで紹介してきた整数型・浮動小数点数型・複素数型・虚数型・文字型などの基本型(primitive type)に対して、派生(derivation)と呼ばれる操作を有限回行うことによって定義される型のことである。派生の操作は、次の5種類で尽くされる。

派生の操作

- 関数型とする。
- ポインタ型とする。
- 配列型とする。
- 構造体型とする。
- 共用体型とする。

ここに示した5種類の操作を基本型に施すことで、基本型を元にした、なにか特殊な型ができる。これらの特殊な型を総称して派生型と呼ぶのである。

8.1.1 派生操作の組み合わせ

派生操作は複数組み合わせて利用することができる。例えば、int型に対して、配列型にする操作を施したあとに構造体型にしても問題ない。派生操作の組み合わせとして問題になるのはいずれも関数型絡みのものばかりである。

組み合わせられない派生

- 関数型にした型はポインタ型にしかできない。
- 配列型にした型を関数型にはできない。

ここで注意して欲しいのはいずれも1段階はさめば大丈夫という事である。例えば、配列型にした型を、更に構造体型にして、そのあと関数型にすることは可能である。

8.1.2 基本型でも派生型でもない型

ここで紹介した基本型と派生型以外にも、特殊な型として次の2つが知られている。

基本型でも派生型でもない型

- void 型
- 列挙型

これらについて注意すべき点として、これらは分類上、独立した型として扱われるが、一部の派生操作が可能であるという点を挙げておく。例えば、void 型は関数に用いて関数型にすることができる。このため、あくまでも「分類の上では」特殊な型として認識しておいていただければ良い。

8.1.3 型についてのまとめ

ここまでに学んだ型についての知識は、基本型への操作として捉えることができる。すなわち、何らかの型の変数/関数を宣言するという事は、基本型ないし特殊な型に対して

- 派生する。
- 修飾子を付す。
- 記憶クラス指定子を付す。

の3つの操作を0回以上行い、その後にオブジェクトを記す、という事である。換言すれば、基本型ないし特殊な型に対して、上記の3操作を適切に施すことで、C言語で扱える任意の型を記述することができるという事である。

8.2 静的配列

抽象概念は終えて、具体的に派生型を見ていくこととしよう。

配列は、変数を一括管理する時などに便利な派生型で、メモリ上に連続的に配置された変数に通し番号をつけたものである。実際のイメージとしては図8.1のようになる。

配列は、宣言の際に個数がわかっており、スタック領域に確保される静的配列 (static array) と、宣言の際に個数がわかっておらず、実行時に個数を計算してヒープ領域に確保

… 別利用	a[0]	a[1]	…	a[k]	別利用 …
-------	------	------	---	------	-------

図 8.1: 配列配置のイメージ

される動的配列 (dynamic array)¹とに分けられる。ここでは、派生で言うところの「配列型」である、静的配列について扱う。

8.2.1 配列に関する用語


配列の各部の呼び名等について、簡単に説明しておく。配列は

1. 配列名
2. 要素数 (配列のサイズ)
3. インデックス (添字)
4. オフセット (開始インデックス)

といった要素からなる。配列名はその名の通り配列の名前であり、要素数はその配列が全部でいくつの変数からなるか (あるいはメモリ上でどれだけのサイズを確保しているか) を示す。インデックスは配列の通し番号のことである。オフセットは配列の先頭の番号で、C では一般に 0 始まりである (これを 0-offset や 0-indexed と表現する)。

8.2.2 1次元配列

通し番号が 1 つだけの配列を 1 次元配列と呼ぶ。以下では、1 次元静的配列について見ていこう。



【ホフスタッター数列の計算】

次の漸化式によって定義される数列をホフスタッター数列と呼ぶ。

$$\begin{cases} Q_n = Q_{n-Q_{n-1}} + Q_{n-Q_{n-2}} & (n \geq 2) \\ Q_0 = Q_1 = 1 \end{cases}$$

1000000 未満の自然数 n が入力される時、ホフスタッター数列の第 n 項 Q_n を計算して出力するプログラムを作成する。

¹可変長配列 (variable length array) とも呼ばれる。可変長配列は長さが変わることを明示的に表し、動的配列はヒープ領域に確保することを明示的に表すものとする、という人もいる。この場合、「可変長の静的配列」といえば、コンパイル時ではなく実行時に個数の決まる、スタック領域に確保される配列、という意味になる。

【解説】

再帰呼び出しをするとあっという間にスタックオーバーフローしそうな式である。そこで、配列を用いて各項の値を計算・保持しておいて出力することにする。なお、配列にメモしつつ再帰を用いる方法もあり、その場合はメモ化再帰と呼ばれる。逆に、このようにボトムアップに計算していく方法は動的計画法 (Dynamic Programming, DP) と呼ばれる。

リスト 8.1: ホフスタッター数列の計算

```
1 #include<stdio.h>
2 #define NUM 1000000
3
4 int main(void){
5     unsigned int array[NUM]={1,1},i,n;
6     scanf("%u",&n);
7     for(i=2;i<=n;i++){
8         array[i]=array[i-array[i-1]]+array[i-array[i-2]];
9     }
10    printf("%u\n",array[n]);
11    return 0;
12 }
```

それでは、配列について見ていこう。

【配列の宣言】

リスト 8.1 では、配列として `array` という変数 (配列変数) が宣言されている。一般に、配列の宣言は次の形で行う。

一次元配列の宣言

一次元配列を宣言する際には

型 配列名 [要素数]

の形式で行う。

静的配列の場合、宣言と同時にメモリ上に確保され、利用が可能になる。この際、確保される領域はスタック領域であるので、制限がある。そのため、静的配列の要素数は必要最小限の量に留めなければならない。また、`[]` の中はあくまでも要素数であり、要素番号の最大値とは異なることに注意しよう。すなわち、`array[5]` と宣言した場合、実際にメモリ上に置かれるのは `array[0]` から `array[4]` の 5 個という事になる。また、一般に `[]` の中は自然数でなければならない。従って、全部で 10 万要素の配列を定義する場合、`array[1E+5]` としてもうまくいかず、`array[(int)1E+5]` とする必要がある。さらに、C99 以外の規格の場合、`[]` の中は定数である必要もある。

以上をまとめておこう。

静的配列の宣言時の注意

- 配列のご利用は計画的に。借り過ぎ使いすぎに注意しましょう。
- 配列宣言の際に [] の中に書くのは要素数であり、最大要素番号はこれより 1 小さくなる (0-offset であるため)。
- 要素数は整数型、自然数でなければならない。
- C99 以外の規格では、静的配列宣言時の要素数は定数である必要がある。

【配列の初期化】

配列を宣言した場合、配列は通常の変数と同様、何が入っているかわからない。そこで、配列も初期化する必要がある。

配列の初期化には何種類かの方法がある。例えば、for 文を用いて

```
for(i=0;i<NUM;i++) array[i]=0;
```

のように、すべての要素に代入してしまう方法もある。あるいは、すべての要素を 0 で初期化する場合、string.h に入っている memset という関数を用いて

```
memset(array,0,sizeof(array));
```

とすれば、全要素を 0 にすることができる。なお、ここで第 3 引数に出てくる sizeof(配列名) は、配列の大きさ (バイト単位) になる。従って、これを配列の第 0 要素で割って

```
sizeof(array)/sizeof(array[0])
```

としたり、型で割って

```
sizeof(array)/sizeof(int)
```

とすることで、個数を取得することもできる (後述)。

ここまでに述べた方法はあくまでも代入であるので、初期化というには少し語弊がある。宣言と同時に代入を行うものを一般に初期化という為である。C では、配列の初期化は直接要素を書くことで行う。

配列の初期化方法

配列を行う際には、宣言時に

```
型 配列名 [要素数]={各要素の値(コンマ区切り)}
```

の形式で記す。この時、要素数は省略することができて、初期化の内容に応じて必要な数の要素を自動で計算してメモリ上に確保してくれる。

例えば、3 項からなる配列を 1,3,6 と初期化したい場合は

```
int array[]={1,3,6};
```

のように記す ([] の中に 3 を書いても良い。)。なお、この方法で初期化する場合、必要な番号までの全要素を列挙する必要がある。この難点をなくすため、C99 では

```
int array[3]={[1]=3,[2]=6};
```

のように、特定の要素だけを指定した初期化ができるようになった。

逆に、要素数のほうが初期化子よりも多い場合、残りの数は 0 で初期化される。この為、全ての数を 0 で初期化したい場合は

```
int array[NUM]={0};
```

のようにしておけば簡単である。リスト 8.1 の l.5 も同じ方法である。

やはり、これもまとめておくことにしよう。

配列の初期化方法のまとめ

- 配列を初期化する場合は、様々な処理に先立って繰り返し文などを用いて代入処理を行えば良い。
- 配列を宣言と同時に初期化したい場合は、配列の宣言の後に={ }を記し、{ }内に順に初期値をコンマ区切りで書いていく。
- C99 では、宣言と同時に初期化する際に、要素番号を指定した初期化ができるようになった。(指示付き初期化子 (designated initializer))

【配列の参照】

配列は変数であるので、変数と同じように参照できる。

配列の参照

配列要素を参照する際には、配列名の後に [要素番号 (インデックス)] と記す。

例えば、リスト 8.1 の l.10 では、配列 array の第 n 要素を出力している。なお、配列の後の [] の中は計算式でもよい (リスト 8.1 の l.8 など)。

配列を呼び出す場合、その添字が存在しない番号、例えば最大値を超えたり、負の数になってしまったりした場合、アクセスすべきでないメモリ領域にアクセスしたとして "Access violation" や "Segmentation fault" などといった実行時エラーを吐く場合がある。そのため、呼び出し時にはこれらのエラーが起きないように、添字に注意しなければならない。

【範囲外アクセスを防ぐための工夫】

しかし、添字に注意すると言っても、人間の注意力には限界がある。そこで、普段から範囲外アクセスを防ぐような、安全なコードを書く方法を探るようにしよう。これにより、範囲外アクセスの危険性を減らすことができる。ここでは、このような「範囲外アクセスを防ぐ工夫」を紹介する。

範囲外アクセスを防ぐためには、ある要素を単独で呼び出す場合、単純にその添字を if 文でチェックするなどの方法がある。だが、実際「単独で」呼び出す場合に範囲外アクセスになる場合はあまり多くないのである。一括処理する際に、見落としなどがあって変な要素にアクセスしてしまう場合が多い。これらを防ぐための方法として、for 文の書き方を一定するという方法がある。

配列に順次アクセスする際の for 文

1. for 文を用いて配列に順次アクセスする際には

```
for(i=0;i<個数;i++)
```

の形式でアクセスすれば安全である。

2. 先で「個数」と書いた部分は配列の要素数であるが、これにはいくつかの取得方法がある。

- リスト 8.1 のように、マクロを用いて一貫して記す。
- `sizeof(配列名)/sizeof(配列名[0])` の形式で個数を計算する。
- `sizeof(配列名)/sizeof(型)` の形式で個数を計算する。(型がわかっている場合のみで、型を変える際の対応が面倒。)

とりわけ、マクロを用いる方法は改変の際などに対応しやすく、一般によく用いられる。C 言語には個数を自動で取得するような万能の機構は無いので²、いくつかの方法を上手く使い分けて範囲外アクセスを避けなければならない。

8.2.3 多次元配列

行列の成分のように、2 つ以上の通し番号が欲しい場合も少なくない。このような場合に対応するために、1 次元配列の番号付の方法を変更して 2 つ以上の通し番号を付けられるようにした多次元配列という機能がある。これは厳密に言えば「配列の配列」という方が正確で、メモリに面的に広がるのではなく、配列がいくつも連続して置かれており、その各配列に対しても通し番号をつけた、というものである。以下、2 次元配列を例に、多次元配列の利用法について見ていくことにしよう。

【正方行列の積】

入力される 2 つの正方行列の積を出力するプログラムを作成する。次元については、コンパイル段階でマクロを用いて決めることとする。

²C++ や Java などでは、配列の大きさを簡単に取得する方法が存在する。

【解説】

- 1次元配列と同様に、配列の要素数はマクロで指定している。ここでのマクロは行列の次元を示している。
- 行列を半角空白及び改行区切りで2個入力する。例えば、2次元の場合

```
1 2
3 4
3 2
4 1
```

のように行い、上2行が1つ目の行列、下2行が2つ目の行列を表す。

- l.19 の printf 第3引数は、行列を1行出力する毎に改行になり、それ以外の場合は区切り文字として空白を出力する。

リスト 8.2: 正方行列の積

```
1 #include<stdio.h>
2 #define DIM 3
3
4 int main(void){
5     double mat1[DIM][DIM],mat2[DIM][DIM];
6     double ans[DIM][DIM]={0},{0};
7     int i,j,k;
8     for(i=0;i<DIM;i++) for(j=0;j<DIM;j++) scanf("%lf",&mat1[i][j]);
9     for(i=0;i<DIM;i++) for(j=0;j<DIM;j++) scanf("%lf",&mat2[i][j]);
10    for(i=0;i<DIM;i++){
11        for(j=0;j<DIM;j++){
12            for(k=0;k<DIM;k++){
13                ans[i][j]+=mat1[i][k]*mat2[k][j];
14            }
15        }
16    }
17    for(i=0;i<DIM;i++)
18        for(j=0;j<DIM;j++)
19            printf("%lf%c",ans[i][j],j==DIM-1?'\\n':' ');
20
21    return 0;
22 }
```

【多次元配列の利用】

多次元配列の利用方法は、リスト 8.2 を見ればわかるとおり、一次元配列と同じである。1.4 にあるように、添字を示す [] の組が増え、それぞれについて要素数を示すようにしなければならないという事、1.5 にあるような形式で初期化できるという事、1.7 のように、各要素に番号を指定することによってアクセスできることなど、いずれも 1 次元配列と同様であることがわかるだろう。なお、初期化には、「指定していない要素は 0 で初期化される」という規格を用いていることに注意されたい。

多次元配列を用いる場合には、1 次元配列以上に使用するメモリ量に注意しなければならない。例えば、リスト 8.2 でマクロ DIM の値を 100 にすると、3 つ存在する 2 次元配列は、各々 $100 \times 100 \times 8 \text{Byte} = 80000 \text{Byte} = 80 \text{kB}$ となり、100 という数字に比べてずいぶん大きくなる。

【多次元配列のメモリ上での配置】

多次元配列は実際には「配列の配列」とであると述べた。例えば

```
int array[3][4]
```

という 2 次元配列を考えてみる。これは、array[0], array[1], array[2] という 3 つの 4 要素の配列の集合である。これが多次元配列が配列の配列であると述べた所以である。

そして、配列の配列というだけあって、多次元配列のメモリでの配置は一次元配列を配列として並べた形になっている。先の例で言えば、まず array[0] が array[0][0] より array[0][3] まで順に並べられ、次いで array[1] が array[1][0] より array[1][3] まで…という形式である。これを図示すると図 8.2 のようになる。

...	a[0][0]	a[0][1]	...	a[0][N]	a[1][0]	a[1][1]	...
...	a[0]				a[1]		...

図 8.2: 多次元配列配置のイメージ

メモリはランダムアクセスであるが、配列の場合には図 8.2 のように、シーケンシャルに配置される。この時、順に見ていくほうが一般的には速いため、多次元配列のすべての項に対する操作を書く際にはリスト 8.2 の 1.8 と同様に、後側の添字を先に変化させた方が効率よく処理を行うことができる。

8.2.4 一次元静的配列を引数に取る関数

ここまで、配列について学んできたが、配列を引数に取る関数を作ることでもある。但し、多次元配列を引数に取るのは少し難しいので、後に学ぶポインタを利用したほうがわかりやすい。ここでは、一次元静的配列を引数とする関数の作成方法について学ぼう。

【ベクトルの内積】

二つのベクトルが配列で与えられる時、その内積を計算する関数を作成する。引数はベクトルを表す配列と、その要素数(=ベクトルの次元)。

【解説】

ベクトルの内積は、各成分の積の総和であるので、成分毎の積をとって順に足していき、最後に総和を返す関数を作成すれば良い。

リスト 8.3: ベクトルの内積

```
1 #include<stdio.h>
2
3 #define DIM 5
4
5 double inner_prod(double v1[],double v2[],int size){
6     int i;
7     double ret=0;
8     for(i=0;i<size;i++) ret+=v1[i]*v2[i];
9     return ret;
10 }
11
12 int main(void){
13     int i;
14     double vec1[DIM],vec2[DIM];
15     for(i=0;i<DIM;i++) scanf("%lf",&vec1[i]);
16     for(i=0;i<DIM;i++) scanf("%lf",&vec2[i]);
17     printf("inner_prod=%lf\n",inner_prod(vec1,vec2,DIM));
18     return 0;
19 }
```

【配列引数とその呼び出し】

関数を学んだ時に、引数は原則として変数の宣言と同じであるという事を学んだ。配列引数の場合もほとんどそれと同じ³である。すなわち、配列の引数を取る場合、配列の宣言の場合と同様の形式で記せばいいのである。

³ほとんどと書いたのは、後に記す参照などの点が異なるからである。仮引数で宣言した場合は通常の変数は同時に定義もなされているが、配列を仮引数に書いた場合、これは配列としては定義されず、等価なポインタが定義されるだけである。また、実引数の配列そのものが仮引数の配列にコピーされるわけではなく、実引数の配列のアドレスが、仮引数のポインタに代入されるだけである。このように、配列引数は内部的にポインタと絡んでいる部分が多いため、ポインタを理解してから再度詳細を理解して欲しい部分である。本書でも、ポインタを学んだ後その観点から配列や文字列を扱う。

配列引数

配列を引数に取る場合は、引数部分に

型 配列名 []

と記す。ただし、要素数が既定である場合には

型 配列名 [要素数]

の形式で宣言しても良い。

要素数が既定である場合の方法についても記したが、一般には要素数は不明であるので、別の引数として要素数を与える必要がある (例: リスト 8.3 の `inner_prod` 関数の第 3 引数)。C99 対応環境であれば、これを更に応用して

```
int arrayfunc(int m,int array[m]);
```

のように書くこともできる。但し、これを書いたからと言って「個数が異なっていればコンパイルエラーになる」などの利点はない⁴。この形式は多次元配列引数の場合などにも用いることができる。

上記によって配列を引数に取る関数を定義することができたが、呼び出しを行う場合はどうだろうか。勿論、配列の各項を渡す場合は通常の変数と同様に

```
func(array[3])
```

などと記したわけだが、ここまで習ってきた配列引数は「配列全体を引数に取る」場合であって、1 要素を取る場合ではない。配列全体を渡す方法は、次に示すとおりである。

配列引数関数の呼び出し

配列全体を引数に取る場合、その関数の呼び出しの際の当該引数部は

関数 (配列名)

のように、配列の名前だけ^aを記す。

^a配列の名前だけを書いた場合、これはその配列の先頭アドレスを意味する。したがって、これは本質的にポインタ渡しであり、後に示す引数配列への直接操作が可能になる所以でもある。詳細はアドレスやポインタを学んだ後に理解されたい。

配列全体を引数にとった例はリスト 8.3 の l.17 にある。このように、実引数に配列名だけを記した場合は配列全体の引数となり、配列の後にインデックスを指定した場合は、通常の変数を渡すのと同様に、配列の特定の要素を関数に渡すことになる。

【引数の配列への直接操作】

配列全体を引数に取る場合に、注意しなければならないことがある。それは、配列全体を引数に取る場合には、配列の値が仮引数の配列に代入されて別の配列として扱われるのではないため、関数内部での配列の操作が元の配列に影響を与えるという事である。

⁴これは、仮引数に現れる配列が、内部的に、それと等価なポインタとして解釈されるためである。

【配列のシャッフル】

元々並んでいる配列をシャッフルする。

【解説】

l.13 の for の終端値に意味はなく「大きい値」というだけである。arrayswap は単に要素を交換するだけの関数であるが、この添字を乱数で決定することでシャッフルを実現している^a。

リスト 8.4: 配列のシャッフル

```
1 #include<stdio.h>
2 #include<stdlib.h>
3 #include<time.h>
4 #define NUM 128
5
6 void arrayswap(int ar[],int size,int i,int j);
7
8 int main(void){
9     int array[NUM];
10    int i,r1,r2;
11    for(i=0;i<NUM;i++) array[i]=i;
12    srand(time(NULL));
13    for(i=0;i<NUM*NUM;i++){
14        r1=rand()%NUM;
15        r2=rand()%NUM;
16        arrayswap(array,NUM,r1,r2);
17    }
18    for(i=0;i<NUM;i++)
19        printf("%d%c",array[i],i%16==15?'\\n':' ');
20    return 0;
21 }
22
23 void arrayswap(int ar[],int size,int i,int j){
24     int tmp;
25     if(i>=size || j>=size) return;
26     if(i<0 || j<0) return;
27     tmp=ar[i];
28     ar[i]=ar[j];
29     ar[j]=tmp;
30 }
```

^aこれによるソート方法をボゾソートと呼ぶ。

リスト 8.4 では、arrayswap 関数で直接配列に対して操作を加えており、それが呼び出し側の配列 array に対する直接の操作となっている。これは、配列引数でやり取りされ

る値がアドレス⁵であり、呼び出された側の関数がアドレスを用いて元の配列を参照して利用する⁶ためである。従って、配列を引数に取る関数を作る場合、仮引数の配列に対する操作は元の (実引数の) 配列に対する操作となる。このことに注意されたい。

8.3 文字列

すでに学んだ文字型を配列にすれば、文を格納することができそうである。この「文を格納する働き」はあちらこちらに現れるため、一般化した取り扱いのための約束事が定められている。それが、文字型の配列=文字列に関する文法である。以下、文字列の扱い方について見ていこう。なお、ここでは char 型に収まる 1 バイトの文字について説明することとし、日本語等 2 バイト以上の文字からなる文字列については付録で簡単に紹介するに留める。

8.3.1 文字列とその取り扱い

文字列は「文字型配列」であるので、配列としてアクセスできるはずである。ここではまず、文字列が配列である例を示す。

【Hello World 再び】

”hello, world”と 2 度出力するプログラムを作成する。

リスト 8.5: Hello World 再び

```
1 #include<stdio.h>
2
3 int main(void){
4     int i;
5     char str[13]={ 'h','e','l','l','o',' ',' ',' ','w','o','r','l','d' };
6     for(i=0;i<13;i++) putchar("hello, world"[i]);
7     putchar('\n');
8     for(i=0;str[i]!='\0';i++) putchar(str[i]);
9     putchar('\n');
10    puts(str);
11    return 0;
12 }
```

⁵メモリ上でそのオブジェクトがどこに位置するかという番号のこと。第 0 講で簡単に紹介した他、ポインタの解説の際に再度取り上げる。

⁶このように、オブジェクトの実体を参照して利用する方法を参照渡しなどと呼ぶ。だが、C 言語での参照渡しは、実際にはアドレスという「値」を渡して、それを参照しているので、参照渡しとは実質「アドレスの値渡し」である。このことを意識すると、C 言語の引数は常に値渡しであると言える。

【文字列に関する規則】

文字列は配列であるので、リスト 8.5 の 1.8 のように、1 文字ずつアクセスすることができて、この時は通常の文字型変数として扱うことができる。だが、それとひとつ違う点があるが、1.8 の条件式に見られる。

ここに書かれている `'\0'` は **NULL 文字** (NULL character, NUL) であり、文字コード 0 の文字である。これは、文字列の終端を示す。

文字列を使う場合、“〇文字まで許容する”というプログラムを書く場合が多いため、文字列の要素数満杯に文字を代入することはまずない。そのため、文字列は何処で終わるかを示す記号を用意した。それが文字列の終端を示す NULL 文字であり、文字列を操作する関数などでも、“NULL 文字が出てくるまで操作を続ける”というように処理が記述されている場合が多い。配列の場合はその要素数によって配列の長さを取得したが、文字列の場合は NULL 文字によって文字列の長さを取得しているのだと考えれば良い。

文字列を配列のように扱う場合などに、終端の NULL 文字は忘れがちである他、文字列の大きさは NULL 文字が入る分も考慮して許容文字数+1 文字にする必要があるなど、注意しなければならない存在である。しかし、終端の NULL 文字は文字列を扱う際にはなくてはならないものなのである。例えば puts 関数は、終端にある NULL 文字を改行文字に変更して出力する関数で、NULL 文字がなければ正しく動作しない。普段 puts の中に文字列リテラルを指定しているが、文字列リテラルの終端には NULL 文字が補われているため、改行されるのである。

【文字列リテラル】

文字列リテラル (string literal) はダブルクォーテーションで囲まれた文字列で、文字列の定数、定文字列といった意味である。以下、文字列リテラルの性質について述べる。

文字列リテラルの性質

- 文字列リテラルの終端には NULL 文字が補われている。
- 文字列リテラルは定数であるので、それに対して書きこむことはできない^a。
- 文字列リテラルを用いて、

```
char str[13]="hello, world";
```

のように初期化することも可能である。リスト 8.5 の 1.5 は配列であることを明示するため冗長に初期化したが、この書き方をしたほうが読みやすい。

- 文字列リテラルは配列名と同じような扱いであり、“文字列リテラル”[i] のようにすることで、文字列リテラルの第 i 文字目を参照することができる。但し、書き込み不能である点と、0-offset である点に注意しなければならない。

^a実際の文字列リテラルの型は `const char *` 型になる。これはポインタを学んだ後に再度理解して欲しい。

以上のように、文字列リテラルはあくまでも「書き換え不能で定義時から中身が定まっている文字型配列」であり

```
const char str[13]="hello, world";
```

などとしているのと同じ事なのである。

【文字列の入出力方法】

リスト 8.5 では、出力に puts 関数を用いたり、一文字ずつ出力したりした。今度は、入力された文字列を出力するプログラムを見てみよう。

【文字列の入出力】	リスト 8.6: 文字列の入出力
入力された文字列を「オウム返し」にするプログラム。1 回の実行で 2 行の入力に対応するが、1 行目の入力には空白文字 (スペース・タブ) を含んではならない。また、各行に入力される文字は 63 文字までとする。(文字列の大きさを 64 としたため)	<pre>1 #include<stdio.h> 2 3 int main(void){ 4 char str[64]; 5 scanf("%63s%c",str); 6 printf("%s\n",str); 7 fgets(str,sizeof(str),stdin); 8 puts(str); 9 return 0; 10 }</pre>

文字列を出力する際には、先に述べた puts 関数の他、printf 関数で %s 書式指定子を用いる方法がある。ここで注意しなければならないのは、%s を書式指定子にした場合、printf の可変引数に配列引数が入るという事である。先に述べた一次元配列を引数に取る関数と同様、ここでも配列名のみを引数にする必要がある (リスト 8.6 の l.6 など)。

入力の方法には scanf を用いる方法と fgets 関数を用いる方法がある。

scanf を用いる場合は、printf 同様に %s 書式指定子を用いるが、この時、文字列の要素数を越えた代入 (バッファオーバーラン (buffer over-run)) はメモリアクセス上危険であるため⁷、防止機構をつけなければならない。この防止機構となるのが入力幅の指定である。リスト 8.6 の l.5 のように、% の後に文字数を書くことで代入される文字数を制限することができ、これによりバッファオーバーランを防ぐことができる。

稀に”scanf で文字列を読み込むとバッファオーバーランを防げない”といっている人を見るが、これは上記のような入力幅指定機能を使っていないためであり、上手く使うことで scanf でも問題なく防げることを付記しておく⁸。

また、scanf で文字列を入力する場合には %[] 書式指定子や %[~] 書式指定子を用いる方

⁷C99 までには文字列の入力関数として gets 関数という関数があったのだが、バッファオーバーランを防止する機能がなく、しばしばハッキングの種となったため、C11 において廃止された。

⁸scanf で本当に問題になるのは、入力形式に従わない入力や、代入するとオーバーフローを起こすような値の入力である。これらは、他の関数の助けを借りて解決しなければならない。

法もある。`%[]` 指定子は、`[]` 内に文字集合を書き、その文字集合に属さない文字が出てくるまで文字列を読み込む、と書式を示す。この時、アルファベットや数字はハイフン(-)を用いて連続指定が可能である。例えば`%[.0-9A-Fa-f]` という指定であれば、ドット・数字・大小文字の A から F のみからなる文字列を読み込み、これ以外の文字が出てきた時に読み込みをやめる、という指定になる。一方で`%[~]` 指定子は、`[~と]` の間に文字集合を書き、その文字集合に属する文字が出てくるまで文字列を読み込む (つまり、`%[]` 修飾子の論理否定形) という指定になる。それ故、`scanf` で 1 行分の文字列を読み込みたい場合などは`%[^\n]` とすればよい (改行文字が出てくるまで文字列を読み込むため、1 行読み込んでくれる)。これらの修飾子を用いた際の可変引数の書き方などは、`%s` と同様である。

`scanf` での文字列入力について、注意と共にまとめておこう。

`scanf` での文字列入力のまとめ

- `scanf` で文字列を入力してもらう際には入力幅指定の上で`%s` 書式指定子または`%[],%[~]` 書式指定子を用いる。
- 前項の書式指定子に対応する文字列は、文字列の名前のみを記し、`&`はつけない^a。
- `%s` 書式指定子では、標準の区切りとされているスペースやタブ、改行を含んだ文字列を入力してもらうことはできない。
- `%[]` は中に文字集合を伴い、その文字集合のみからなる文字列を読み込む。一方`%[~]` は、中に書いた文字集合に含まれない文字からなる文字列を読み込む。どちらの文字集合指定でも、アルファベットや数字は-を用いて連続指定ができる。
- `scanf` を用いて (文字列に限らず) 入力を行った場合、最後に使われた区切りの文字は「まだ読み取られていない文字」として残されたままになり、次の文字列読み込みの際に、その文字列の冒頭に入力されてしまう。これを防ぐため、文字列の入力が後に控えている場合、`scanf` の書式文字列の末尾に一文字読み飛ばしを意味する`%c` を付すと良い。

^aこれは、一般の変数に対しては`&`を付すことによって、その変数のアドレスを示すことになるのに対し、文字列などの配列はその名称を記すことによってアドレスを示すためである。詳細はポインタの解説の際に理解されたい。

次に、`fgets` 関数による読み込みを見てみよう。

`fgets` による標準入力からの文字列入力

`fgets` 関数により標準入力から文字列を読み込む場合には

```
fgets(読み込み先文字列名, 読み込み最大文字数, stdin);
```

を実行する。

`fgets` からの読み込みはまるごと 1 行か、もしくは読み込み最大文字数に達するかの、どちらか早い方である。すなわち、まるごと 1 行がすっぽり文字列に入るのであればその 1

行を改行文字も含めて読み込み、入らなければ最大文字数の部分まで読み込むことになる。最大文字数まで読み込んだ場合、次の呼び出しで続きから読み込むことになるが、この詳細についてはストリームの章に解説を譲る。

fgets において入力区切り文字となるのは改行のみである。このため、ある英文を読み込む時、単語毎に読み込むのであれば scanf の方が利便性が高いという事になる。

なお、scanf 関数も fgets 関数も、文字列がいっぱいになるまで読み込んだ場合以外は終端に自動で NULL 文字を付してくれるので、自力で NULL 文字を付す必要はない。

8.3.2 文字列操作関数

文字列を扱う関数は string.h、stdio.h、stdlib.h を始めとして多くのヘッダファイルに入っており、挙げていくと枚挙に暇がない。これらについての説明は付録に譲るとして、ここではいくつかの例を示すに留める。

【DL 速度計算プログラム】

接頭辞つき Byte 単位 (B,kB,MB,GB,TB) で入力されるファイルを、同じく接尾辞つき bps 単位 (bps,kbps,Mbps) で入力される回線速度によってダウンロードするのにかかる時間を秒単位で出力するプログラムを作成する。入力の例としては

650MB

7.2Mbps

等となり、この場合は 650MB のファイルを回線速度 7.2Mbps で落とすのに必要な時間を計算せよという事になる。

リスト 8.7: ダウンロード時間の計算

```
1 #include<stdio.h>
2 #include<string.h>
3
4 int main(void){
5     char str[2][64],file[8],net[8];
6     double files,nets,second;
7
8     fgets(str[0],sizeof(str[0]),stdin);
9     fgets(str[1],sizeof(str[1]),stdin);
```

```
10
11     sscanf(str[0],"%lf%7s",&files,file
12 );
13     sscanf(str[1],"%lf%7s",&nets,
14 net);
15
16     switch(file[0]){
17     case 'T':
18         files*=1000;
19     case 'G':
20         files*=1000;
21     case 'M':
22         files*=1000;
23     case 'k':
24         files*=1000;
25     }
26     files*=8;
27
28     if(!(strcmp(net,"Mbps",4)))
29         nets*=1000*1000;
30     else if(!(strcmp(net,"kbps",4)
31 )))
32         nets*=1000;
33     second=files/nets;
34     printf("%lf_\second\n",second);
35     return 0;
36 }
```

【文字列からの読み込み】

リスト 8.7 の l.11 に見られる `sscanf` 関数は、第 1 引数に示す文字列を、第 2 引数の書式指定文字列に従って解釈し、`scanf` と同様に可変引数に代入する関数である。ここでは `fgets` と組み合わせることによって利用したが、本来は文字列の書式チェックを行った後にこれを用いて読み込むと利便性が高い関数である。なお、`sscanf` 関数は第 1 引数に文字列を持ってくること以外は `scanf` 関数と同様に使うことができる。

【文字列の比較】

リスト 8.7 の l.26 に見られる `strncmp` 関数は文字列同士を比較する関数である。第 1 引数の文字列と第 2 引数の文字列を先頭から比較し、第 3 引数に示される文字数まで同じであれば 0 を、同じでなければ非 0 (厳密には、比較した部分について、第 1 引数のほうが辞書順で早い場合は負の値、遅い場合は正の値) を返す。なお、文字列同士が完全に同じかどうかを比較する関数として `strcmp` 関数もあり、これは `strncmp` 関数から第 3 引数をなくしたものである。

なお、文字列同士の比較を

```
str1==str2
```

のような形式で行なっても、正しい結果は得られない。これは、何度か述べたように、文字列の名前だけ書いた場合には文字列のあるアドレスを表すだけであり、上記の文が「文字列 `str1` と文字列 `str2` のアドレスが等しいかどうか」＝「文字列 `str2` が文字列 `str1` そのものであるかどうか」という意味になってしまうためである。一方、`strcmp` 系関数を用いた場合は「文字列 `str1` の中身が文字列 `str2` の中身に比べてどうか」という事になるので、文字列が同じかどうかの比較をすることができる。

【その他の文字列関数】

文字列を取り扱う関数には他にも多くある。

文字列を扱う関数の例

- `stdio.h` には、`sprintf` や `snprintf` 等、`printf` と同じ形式で出力先を文字列にした関数がある。
- `stdlib.h` には、`strtod` や `strtol` など、文字列から数値を読みだして整数型や浮動小数点数型の値を返す関数がある。
- `string.h` は文字列に関する主要な関数が揃っており、長さを求める `strlen` 関数や文字列への代入を行う `strcpy`/`strncpy` 関数、文字列同士の結合を行う `strcat` 関数など多様な関数がある。

文字列を扱う際には、先の「文字列が配列である」「終端文字が NULL 文字である」という概念さえ理解しておけば大抵の処理を書くことができるが、必要となるであろう処理の多くは標準関数として提供されている。この為、付録 A などを見て文字列を扱う関数にはどのようなものがあるか知っておくのも良い。

本講の要点

本講では最初に派生型について述べた後、静的配列と文字列について学習した。

基本型と派生型

- C 言語に用意されている整数型・浮動小数点数型等をまとめて基本型という。
- 基本型に対して関数・配列・構造体・共用体・ポインタの操作を有限回行なうことができる型を派生型という。
- 派生操作は組み合わせることができる。
- void 型や列挙型等、基本型でも派生型でもない型が存在する。

静的配列

- 配列とは、メモリ上に連続して配置された変数に通し番号をつけたものである。
- 配列のご利用は計画的に。借り過ぎ使いすぎに注意しましょう。
- 配列の宣言時には [要素数] を配列名の後ろに付す。
- 配列は 0-offset であり、要素数-1 番がその配列の末項である。
- 配列を呼び出す際には配列外アクセスをしないように注意しなければならない。
- 配列の添字を 2 個以上にすることもできる (多次元配列)。
- 配列にアクセスする際にはシーケンシャルになるようにアクセスしたほうが効率がよく、特に多次元配列の場合後側の添字から順に変化させると良い。
- 配列引数関数を作る場合、配列の要素数も同時に教えるべきである。
- 配列引数関数に配列全体を実引数として渡す場合には配列名のみを記す。
- 配列引数関数での配列操作は実引数の配列にも影響を及ぼす。

文字列

- 文字列とは文字型の配列のことで、終端が NULL 文字によって示される。
- 文字列入力の際にはバッファオーバーラン防止処理を施さなければならない。
- 文字列リテラルは書き換えることができない。
- 文字列を扱う関数は string.h はじめ、stdio.h や stdlib.h などにも入っている。

第9講 派生型(2) 構造体・共用体他

ここまでは変数を一つ一つ扱ってきた。だが、座標を扱う際など、別々の二つの変数を用いてひとつの実体を表すのは不自然であり、見づらい。また、座標を扱う関数などを作る場合、座標を返却値にしたい場合もあるが、関数ではひとつの値しか返すことができず、不便である。これらの問題を解決するために便利なのが、「いくつかの型を組み合わせで新たな型にする」構造体や共用体である。構造体はいくつかの型をセットにして、共用体はひとつのビット列に対していくつかの型による見方を当てはめて用いる派生型で、何れも似たような宣言の仕方を行う。本講ではこれらについて学んでいく。また、特殊な型ではあるが、構造体や共用体と似た形式で定義を行う列挙型についても本講で扱う。

9.1 構造体

まず、いくつかの変数をひとまとめにして扱う、いわば「変数のセットメニュー」である構造体について学んでいこう。

9.1.1 構造体の概念

構造体 (structure) とは、複数の変数の集合体を1つの変数として扱うことができるようにした派生型である。例えば、2次元直交座標を表すときには、 x 座標と y 座標は常にセットで扱えたほうがわかりやすい。このようなときに構造体は威力を発揮する。

配列は同一の型の変数をメモリ上に連続的に並べ、それに通し番号をつけたものであった。構造体は任意の型のいくつかの変数をひとまとめにして名前をつけたものであり、メモリ上に連続的に配置されるなどの特徴はない。

構造体はセットであることを強調してきたが、これはひとつの実体を表すのにいくつかのデータを用いるという事に対応する。例えば、書籍を考えてみよう。書籍は書名・作者名・価格・ISBNコードなどいくつかの要素を持っている。このそれぞれの要素に変数を割り当てた後、これらをひとまとめにして扱えば書籍を表す変数になるだろう。このとき、構造体の各要素のことをメンバ (member) と呼ぶ。例えば、先に挙げた書籍構造体には、書名を表すメンバ、作者名を表すメンバなどがあると言える。

9.1.2 構造体の利用

構造体の概念を理解した所で、今度は実際に使ってみよう。Joker 抜きの特ランプを使ったハイ&ローを実装してみる。

【トランプ版ハイ&ロー】

トランプを用いた1回勝負のハイ&ローを作成する。

リスト 9.1: トランプ版ハイ&ロー

```
1 #include<stdio.h>
2 #include<stdlib.h>
3 #include<time.h>
4
5 int main(void){
6     struct trump{
7         char suit;
8         short seq;
9     }cards[52];
10    int r1,r2,i;
11    char c;
12
13    for(i=0;i<52;i++){
14        switch(i/13){
15            case 0:
16                cards[i]=(struct trump){ 'S
17                    ',i%13+1};
18                break;
19            case 1:
20                cards[i]=(struct trump){ 'H
21                    ',i%13+1};
22                break;
23            case 2:
24                cards[i]=(struct trump){ 'D
25                    ',i%13+1};
26                break;
27            case 3:
28                cards[i]=(struct trump){ 'C
29                    ',i%13+1};
30                break;
31        }
32
33    srand(time(NULL));
34    r2=r1=rand()%52;
35    while(r2==r1) r2=rand()%52;
36
37    printf("now:%c-%d\n",cards[r1].
38        suit,cards[r1].seq);
39    printf("is_next_H/L?>>>");
40    c=getchar();
41
42    printf("next:%c-%d\n",cards[r2]
43        .suit,cards[r2].seq);
44    if(c=='h' || c=='H'){
45        if(cards[r1].seq<cards[r2].seq)
46            puts("You_win!");
47        else
48            puts("You_lose...");
49    }else{
50        if(cards[r1].seq>cards[r2].seq)
51            puts("You_win!");
52        else
53            puts("You_lose...");
54    }
55
56    return 0;
57 }
```

【構造体の宣言】

先に述べたように、構造体は型である。従って、その宣言には

- 新たな構造体型の宣言
- 作成した構造体型の変数の宣言

の2段階がある。ここで、後者の「作成した構造体型の変数の宣言」は、今までの変数と同様に、

構造体型名 変数識別子名

で行うことができる。そのため、ここで紹介するのは主に前者の「新たな構造体型の宣言」である。

構造体で新たな型を作成する場合には、次のように行う。

構造体型の宣言

新たな構造体の型を宣言する場合には

```
struct 構造体タグ名{
    メンバの型 メンバ識別子;
    メンバの型 メンバ識別子;
    :
    :
};
```

のように行う。これにより宣言された構造体の名前は `struct` タグ名となる。

構造体タグ名とは、構造体そのものの名前であり、これを決めることによって構造体の名前が定まる。注意しなければならないのは、中身が全く同じであっても構造体タグ名が違う構造体は別の型であるという事である。

リスト 9.1 の 1.6~1.9 に見られるように、構造体型の宣言と構造体変数の宣言を同時に行うこともできる。このような場合、あるいは `typedef` を用いる場合には、タグ名を省略して

```
struct{
    メンバの型 メンバ識別子;
    メンバの型 メンバ識別子;
    :
    :
} 変数名;
```

のように宣言することもできる¹。このようにタグを省略した場合、タグ名はコンパイラが勝手に決める。従って、タグ名を省略した、中身が全く同じである二つの構造体を作ったとしても、これらは見た目が同じだけで別の型なのである。

ここでは型が同じであるかどうかについて強調してきた。これは、このあとに学ぶ代入の時などに重要になってくるためである。構造体は同一タグ名で宣言されたもののみが同じ型の変数になるという事を、ここで再強調しておく。

【構造体の初期化と代入】

構造体を初期化する場合は、通常の配列と同じようにメンバ毎の初期値を順に `{}` とコンマ区切りで記す。勿論、配列と同じように、一部のメンバのみを初期化することもでき、この形での初期化を行った場合、初期化を行わなかったメンバは 0 で初期化される。

¹C11 において無名構造体機能が追加されたと書かれているページを見ることがある。これは、ここで示したような「タグのない構造体」の機能ではない。C11 においてサポートされた無名構造体とは、後に述べる構造体のネストの際などに、内側の構造体型のメンバにメンバ名をつけずに用いることができるという機能である。この、無名構造体 (無名共用体) の機能は C++ との互換のために C11 で追加された機能であるが、実際に利用する機会はないと思われるので、本書ではこの脚注以外には述べないものとする。

構造体の初期化

構造体を初期化する場合

```
struct tag var={初期値 1, 初期値 2,...};
```

といった形式で、宣言時に {} とコンマ区切りで^a初期値を記す。

C99 においては配列同様に指示付き初期化子が存在し

```
struct tag var={  
    .メンバ名=初期値,  
    .メンバ名=初期値,  
    :  
}
```

という形式で初期化を行うことができる。

^a構造体の型宣言時におけるメンバの宣言はセミコロン区切りであるので、混同しないように注意。

このように、初期化の際には直接 {} に中身を書くことができるが、代入の場合はまた話が少し変わってくる。一般に、構造体に代入を行う場合、同じ型の構造体でなければ代入できない。これは、少し考えてみれば当然であろう。例えば、人間の身長と体重は共に実数であるし、座標は共に実数である。だが、座標に人間の身長と体重を代入するのは全く意味がない。したがって、もしも似たようなメンバがあったとして、違う型から値を代入したい場合は、この後に述べるメンバ毎の操作に従って、メンバ毎に代入していかなければならない。だが、この方法ではいちいちメンバ名を書くことになり、「セットとして扱う」という利点が活かされていないことになる。そこで、C99 において、複合リテラル (Compound Literal) という記法が導入された。これは {} 内に記された値を構造体型にキャストすることにより、一時的にその型の構造体として用いるようにする機能である。

複合リテラルの記法

複合リテラルは

```
(構造体型名){メンバ1の値,メンバ2の値,...}
```

によって書くことができる。この記法にも指示付き初期化子を利用できる。

例えば、リスト 9.1 の l.16 を見てみよう。これはメンバ `suit` の値が 'S' であり、メンバ `seq` の値が `i%13+1` であるような値を、構造体型変数である `cards[i]` に代入している。このように、構造体をまるごと代入する場合には、初期化の時と同様の記法で記し、それを構造体型にキャストすることによって簡単に処理を行うことができるのである。

【構造体メンバへの操作】

構造体は色々なメンバの集合体であるため、当然メンバ毎に参照できる。

構造体のメンバの参照

構造体メンバを参照するには、選択演算子 (selection operator) の一つであるドット演算子 (dot operator) . を用い

構造体変数名. 構造体メンバ名

と記述する。

これにより、構造体のメンバを直接参照できるので、複合リテラルを用いない場合はC99以前の規格の場合は、メンバ毎に代入することで構造体への代入を実現することができる。

リスト 9.1 においては、例えば 1.34 で用いられている。このように、構造体のメンバはドット演算子を用いて参照してしまえば、後は通常の変数として用いることができるのである (勿論、メンバが派生型である場合には、その派生型の変数として用いることができる)。

【構造体メンバのメモリ上での配置】

配列の場合、その各要素は連続的に配置されていた。だが、構造体には、連続的に配置されるという性質はない。これは、後で学ぶポインタの運用の際に重要になることである。構造体がメモリ上でどのように配置されるか、ここで簡単に説明しておく。

構造体は、メモリの先頭から末尾に向かって、メンバの順番を崩さないように配置される。つまり、構造体の第1メンバは第2メンバよりも必ず前にある。しかし、第1メンバと第2メンバの間に「穴」があく場合 (=第1メンバと第2メンバが連続でない場合) が存在する²。第2メンバと第3メンバの間も同様の関係であり、以下同様である。但し、メンバが配列である場合、その配列メンバは当然連続である。例えば、第1メンバと第3メンバが int 型変数で、第2メンバが int 型の配列であるような構造体であれば、第2メンバはその中で連続であるが第1メンバと第2メンバ、第2メンバと第3メンバの各々の間には穴があるかもしれない。以上を図示すると図 9.1 のようになる。

...	メンバ 1	穴	メンバ 2	穴	...	メンバ n	穴	...
...	構造体							...

図 9.1: 構造体のメモリ上での配置のイメージ

なお、構造体型の配列を作った場合、各要素は連続的に配置されるが、要素毎にメモリをみた場合、その要素の確保したメモリ領域には穴が存在する可能性がある。

²このような穴は、データ型の大きさとメモリの区切り方 (アラインメント (alignment)) の違いから生じるものである。したがって、アラインメントと型の配置をうまく合わせることで、穴を減らして効率的にメモリを利用できる。

sizeof 演算子を構造体に用いた場合、この「穴」の分も構造体の大きさに含まれる。従って、構造体のサイズがメンバのサイズの合計よりも大きくなることもある。

【構造体のネスト】

派生型は組み合わせることができると説明したとおり、構造体を配列にしたり、配列を構造体のメンバに入れたりすることもできる。また、構造体そのものを構造体のメンバに入れることもできる (構造体のネスト)。実例として、円と点に関する扱いを見てみよう。

【円と点の位置関係】

与えられる点を与えられる円の中にあるか外にあるか、あるいは境界上かを判定するプログラムを作成する。なお、位置の判定は関数として独立させる。

【解説】

点を表す構造体 `struct point` を先に定義し、その後でそれを用いた円を表す構造体 `struct circ` を定義している。ここが構造体ネストである。なお、1 行目に点の座標を半角空白区切りで、2 行目に円の中心の座標と半径を半角空白区切りで入力するようにしている。

リスト 9.2: 円と点の位置関係

```
1 #include<stdio.h>
2 #include<math.h>
3
4 struct point{
5     double x;
6     double y;
7 };
8
9 struct circ{
10     struct point cent;
11     double r;
12 };
13
14 int check(struct circ c,struct
15     point p);
```

```
16 int main(void){
17     struct circ circle;
18     struct point p;
19     int flag;
20     scanf("%lf%lf",&p.x,&p.y);
21     scanf("%lf",&circle.cent.x);
22     scanf("%lf",&circle.cent.y);
23     scanf("%lf",&circle.r);
24     flag=check(circle,p);
25     switch(flag){
26     case -1:
27         puts("The point is in the
28             circle!");
29         break;
30     case 0:
31         puts("The point is on the
32             circle's boundary!");
33         break;
34     case 1:
35         puts("The point is out of
36             the circle!");
37         break;
38     }
39     return 0;
40 }
41
42 int check(struct circ c,struct
43     point p){
44     double dx,dy,dist;
45     dx=p.x-c.cent.x;
46     dy=p.y-c.cent.y;
47     dist=hypot(dx,dy);
48     if(dist<c.r) return -1;
49     else if(dist>c.r) return 1;
50     else return 0;
51 }
```

リスト 9.2 を見てもらえばわかるとおり、構造体をネストしたからといって何ら特別の記法が必要になるわけではない。せいぜい、1.21 に見られるようにドット演算子を 2 回用いて参照を行なっている程度である。このように、構造体であるメンバを構造体に入れる場合も、他の変数のメンバを構造体に入れる場合も、その変数の記法に従ってコードを記述すれば良い。

構造体のネストが効果を発揮しているのは、実体の間の関係がよくわかるという事である。再度リスト 9.2 に目を向けよう。ここでは、円の中心を点の構造体としてネストしている。勿論、円の中心は点であるのでこれは自然なネストであるが、円の中心をわざわざ 2 つの double 型変数によって表すよりも見やすいのではないのだろうか。円の構造体を「中心を表す 2 つの実数と半径を表す実数からなる集合体」とするより、「中心の点を表す構造体と半径を表す実数からなる集合体」としたほうが、円をイメージしやすいのではないか。これが構造体のネストを利用する意義である。この考えは Java や C++ などで用いられる「継承」という機能に応用されている。C 言語の学習を終えた後、継承の機能があるプログラミング言語を学んだならば、この構造体ネストの内容を基にしていると考えればより理解しやすいだろう。

【構造体を用いる関数】

リスト 9.2 では、構造体を引数に取る関数があらわれている。だが、構造体を用いたからと言って何ら特別なことは為されていない。構造体を引数にとったり、あるいは返却値にしたりする場合も、通常の変数と同様に引数/返却値に記述すれば良いのである。

ところで、配列引数の場合は、関数内から仮引数へ操作を行うと実引数にも影響を及ぼした。しかし、構造体の場合はどうなのだろうか。勿論これは実験すればわかるので、自分で適当なプログラムを書いて実験してみれば良い。例えば、次のような関数を作ってみれば良い。

```
struct tag{
    int x;
    int a[5];
};

void input(struct tag num){
    num.x=100;
    num.a[0]=12345;
}
```

後は main 関数で struct tag 型の変数を宣言し、これを input 関数の引数にして呼び出した後、そのメンバ x および a[0] の値を出力してみれば良い。

さて、実際に実験してみればわかったと思うが、構造体を用いた場合、構造体は値のコピーが行われる「値渡し」であるので、呼び出された側の関数で構造体を操作したからと言って実引数の構造体には影響を与えない。これは構造体内の配列メンバについても同じ

なのである。従って、配列を構造体に”包んで”渡せば、関数での操作が実引数に影響されることはなくなるのである³。

構造体を引数や返却値に用いた場合、例えば2つ以上の値を返す関数を作ることができて便利であるし、可読性も上がる。だが、構造体を引数などに使う役目はそれだけではない。構造体は先にあげた配列の例のように、引数を渡す際のオブジェクトの役目を果たするのである。前講では扱わなかった多次元配列を引数にする関数も、多次元配列を構造体に包んで渡してやれば比較的簡単に実現することができる。勿論、配列を渡す際には要素数情報と共に渡すのであるが、その要素数情報も構造体のメンバに入れて「配列を拡張したような構造体」を作れば⁴配列に関して必要な情報を一手に扱うことができる。

【構造体を使うタイミング】

ここまで、構造体を学んできた上で気づいたであろうが、構造体はマシンやコンパイラに対しての機能と言うよりも、人間にわかりやすくソースを記述するための機能である。そのため、構造体を使うタイミングというものも重要になってくる。勿論、先に書いたように関数から2つ以上の値を返したい場合に一時的に用いることもあるだろうが、大抵の場合、構造体はプログラム中で一貫して用いられるものである。

構造体を作る際の指針として特別に記述しなければならないことは殆どないが、強いてあげるならば「関係性のあるものを構造体としてまとめ、関係性のないものはまとめない」という事だろうか。概念の説明の際に記述した書籍の例にしてもそうであるが、「ある実体の持つ属性」のみをまとめたものがひとつの構造体になるのが望ましい。書籍のISBNコードと弁当の値段をひとまとめにしたとして、それが一般にわかりやすく使いやすい集合体と言えるだろうか。構造体を用いる場合には「その構造体は何を表すのか」を明確にして、関係性のある変数をまとめるべきなのである。

9.1.3 ビットフィールド

構造体を用いる際に、少し特殊なメンバとして int 型をより小さいビット数に分割した整数型を用いることができる。この機能をビットフィールド (bit field) という。この機能は使用メモリ量を節約する場合や、ビット毎にフラグを使いたい場合などに用いられる。

【誕生日のチェック】

入力される2人の誕生日が年まで同じか、月まで同じか、あるいは同じでないかどうかをチェックする。年・月・日はまとめて4バイト以内に収める。

³但し、値をコピーするためにもうひとつ配列を用意するため、当然ながらメモリを食う。

⁴Java では配列そのものが「C での素朴な配列に色々機能をつけて拡張したもの」になっており、C++ では”vector”という利便性の高い配列が存在している。ここからわかるとおり、「配列を拡張したような構造体」は一般によく用いられるものである。

リスト 9.3: 誕生日のチェック

```

1 #include<stdio.h>
2
3 int main(void){
4     struct{
5         signed int year:20;
6         unsigned int mon:4;
7         unsigned int day:5;
8     } birth[2];
9     short tmp,i;
10
11     for(i=0;i<2;i++){
12         scanf("%hd",&tmp);
13         birth[i].year=tmp;
14         scanf("%hd",&tmp);
15         birth[i].mon=(unsigned int)
16             scanf("%hd",&tmp);
17         birth[i].day=(unsigned int)
18             tmp;
19     }
20     if(birth[0].mon==birth[1].mon
21        && birth[0].day==birth[1].
22        day){
23         puts("Same birthday!");
24     }else
25         puts("Not same birthday!");
26     return 0;
27 }
```

ビットフィールドを用いる場合には、次のように行う。

ビットフィールドメンバの宣言

構造体のメンバをビットフィールドのメンバとする場合、メンバ宣言時に

メンバ型名 メンバ名:ビットの大きさ;

と、:ビットの大きさを付す。

ここで、メンバ型名は通常 `signed int` か `unsigned int` である。ビットフィールドの宣言の場合、単に `int` と書いた時の符号の有無は環境に依存する。そのため、リスト 9.3 でも全てのビットフィールドの符号を明示しているのである。

ビットフィールドは通常とは違う切れ目を作るため、アドレスを取得できなかつたり、定まった大きさの倍数にならないと足りない分だけ構造体に「穴」ができたりするといった欠点を持つ。だが、それらの欠点を含めても余りあるメモリ使用量削減効果を発揮する。とりわけ、メモリ量が大きく制限されている環境において配列などを用いたい場合、その値の最大値などを考慮してビットフィールドを用いればメモリの節約になることは多い⁵。しかし、使い方を上手く考えなければ、効果的な運用とはならない場合もある。

たとえば、本書で説明している gcc などでは、構造体を $4n$ バイトで表そうとする為、4 バイトに満たないビットフィールドを作った場合に「穴」が生まれる⁶。また、この $4n$ バイトになるような制限のために、却ってサイズを大きくしてしまう場合もある。例えば9 ビットのビットフィールド3つの後に17ビットのビットフィールドを作り、更にその後に `short` 型のメンバを取ろうとすると、12 バイトの構造体になってしまう場合がある。これ

⁵例えば、1990 年代頃までのコンシューマ向けゲーム機などでは、メモリ節約のためにビットフィールドを使っているように見受けられる「パラメータの限界値」が散見される。

⁶これは `char` 型や `short` 型を用いて4の倍数の大きさでない構造体を作っても同様である。

は、合計で $27+17+16=60\text{bit}$ ⁷で、8 バイトに収まるはずと考えられる。だが、4 バイトごとの境界線をまたぐことができない環境⁸というものがある。ここで 12 バイトになってしまったのは、4 バイトごとの境界線をまたぐことができない環境であったために 9 ビット 3 つの後に穴が空いて 4 バイトとなり、更に 17 ビットの後にも穴が空いて 4 バイト、short の後にも穴が…となったものである。この場合、short 型 4 つ+int 型 1 つで 12 バイトであるので、制限が多いビットフィールドを用いるメリットは感じられない。

以上のように、ビットフィールドはよく考えて用いなければならないが、有効に利用すればメモリの節約効果は大きい。

なお、ここでは int 型を基にしたビットフィールドについて紹介したが、C99 においては _Bool 型のビットフィールドも許される。この為、フラグの集合体などを利用する際にもビットフィールドを有効活用することができる。

9.2 共用体

構造体と同じ形で宣言/参照するもうひとつの派生が共用体 (union) である。

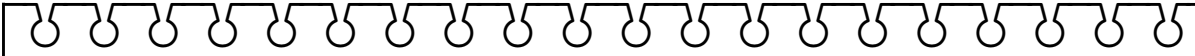
9.2.1 共用体の概念

共用体は構造体と同じくいくつかの型のメンバを持つ「メンバの集合体」である。しかし、当然構造体とは違う型である。

共用体はあるメモリ領域に違う見方を当てはめた集合体である。例えば、ビット列が全て 1 であるような 32bit を考えよう。この時、このビット列を int 型で見るとこれは -1 であるが、unsigned int 型で見るとこれは UINT_MAX の値になる。float 型の場合、これは NaN を表す値である。このように、同じビット列でも違う型であればその値はかわってくる。この、ひとつのビット列に対して「いくつかの見方」を与えたものが共用体である⁹。

9.2.2 共用体の利用

共用体を利用する例はそれほど多くは思いつかないが、例えばベクトルと複素数を対応付けるなどが考えられる。その例を見てみよう。



【ベクトルの回転】

ベクトルの 90 度回転を行うプログラムを書く。

⁷ここでは short を 2 バイトとした。以下同様。

⁸厳密には、int の大きさ毎に境界があるという方が多い。ここでは int 型を 4 バイトとして書いた。

⁹キャストすれば良い、と思うかもしれないが、キャストは「値を保存する変換」であって「ビット列を保存する変換」ではない。従って、-1 を float 型にキャストしても NaN にはならない。int 型の -1 を unsigned int にキャストすると最大値になるのは、unsigned int の最小値が 0 で、そこからオーバーフローを起こすためである。

【解説】

共用体を用いてベクトルを一時的に複素数とみなすことにより、 $e^{\frac{i\pi}{2}}$ をかけるだけで計算することができる。

リスト 9.4: ベクトルの回転

```
1 #include<stdio.h>
2 #include<math.h>
3 #include<complex.h>
4 #include<tgmath.h>
5
6 int main(void){
7     struct vec{
8         double x;
9         double y;
10    };
11    union cvec{
12        struct vec v;
13        double complex comp;
14    }cvecs;
15    const double pi=atan(1.0)*4;
16
17    scanf("%lf_%lf",&cvecs.v.x,&cvecs.v.y);
18
19    cvecs.comp*=exp(I*pi/2);
20    printf("%lf,%lf\n",cvecs.v.x,cvecs.v.y);
21    printf("%lf+%lfi\n",creal(cvecs.comp),cimag(cvecs.comp));
22    return 0;
23 }
```

リスト 9.4 をみてもらえばわかるとおり、共用体の宣言や定義、参照方法は構造体と同じである。ただ、struct と書いていたところが union に変わっただけである。

注意しなければならないのは共用体の初期化である。共用体を初期化する場合、先頭で宣言したメンバの型で行う。

共用体の初期化

共用体の初期化を行う際には

共用体名 変数名={先頭メンバの型での値};

と記す。先頭メンバ以外で初期化したい場合は、指示付き初期化子を用いる。

なお、共用体に複合リテラルはない。また、共用体へのキャストは、キャスト元が共用体の中にある型の場合のみに許される。この時、当然キャスト元の値の型のメンバを用いて代入が行われる。例えば、リスト 9.4 の l.19 のように直接 `cvecs.comp` とメンバを指定

して代入することもできるが、double complex 型の計算を行った後に共用体型へのキャストを行なって

```
cvecs=(union cvec)(cvecs.comp*exp(I*pi/2));
```

のようにしても、同じ動作を示す。

以上のように、共用体はほぼ構造体と同じ方法で利用することができるが、構造体が複数実体をひとつにまとめているのに対して、共用体が一実体をまとめているという違いから、運用にも少し違いが出てくる。この点に注意すれば、例えば浮動小数点数のビットの状態を見るなどの利用方法が考えられるだろう。

9.3 列挙型

列挙型は派生型ではなく、基本型でもない「特殊な型」である。

9.3.1 列挙型の概念

列挙型 (enumeration) は、その変数の取りうる値を全て列挙することからこの名前になった型である。例えば、曜日を表す型を作ると考えると、取るべき値は Sunday, Monday, ... の全部で 7 つである。これらをそれぞれ 0, 1, ..., 6 に対応させておけば、コンピュータ内でも扱いやすいし、曜日のフラグとしても使いやすい。そこで、0 は Sunday, 1 は Monday, ..., 6 は Saturday という事を列挙し、これを新しい型とすることができる。この機能が列挙型である。

列挙型はその性質上、フラグとして用いられる場合が多い。とりわけ、C99 以前の規格において `_Bool` 型と同等の機能を実現するのに用いられることが多かった。

9.3.2 列挙型の利用

それでは、実際に列挙型の扱いを見ていくことにしよう。先に扱った「円と点の位置関係」のプログラム (リスト 9.2) は、列挙型を用いるとよりわかりやすく書き直せる。

【円と点の位置関係 (列挙型版)】

リスト 9.2 のプログラムを、列挙型を用いて書きなおしてみる。

【解説】

点が円の内側・境界・外側のどれに位置するかを表す `flag` を列挙型を用いて実装した。int 型の時より「読んでわかる」ようになったことを実感してほしい。

リスト 9.5: 円と点の位置関係 (列挙型版)

```

1  #include<stdio.h>
2  #include<math.h>
3
4  struct point{
5      double x;
6      double y;
7  };
8
9  struct circ{
10     struct point cent;
11     double r;
12 };
13
14 enum place{
15     IN=-1,
16     BOUNDARY=0,
17     OUT=1
18 };
19
20 enum place check(struct circ c,
21                  struct point p);
22
23 int main(void){
24     struct circ circle;
25     struct point p;
26     enum place flag;
27     scanf("%lf%lf",&p.x,&p.y);
28     scanf("%lf",&circle.cent.x);
29     scanf("%lf",&circle.cent.y);
30     scanf("%lf",&circle.r);
31     flag=check(circle,p);
32     switch(flag){
33     case IN:
34         puts("The point is in the circle!");
35         break;
36     case BOUNDARY:
37         puts("The point is on the circle's boundary!");
38         break;
39     case OUT:
40         puts("The point is out of the circle!");
41         break;
42     }
43     return 0;
44 }
45
46 enum place check(struct circ c,
47                  struct point p){
48     double dx,dy,dist;
49     dx=p.x-c.cent.x;
50     dy=p.y-c.cent.y;
51     dist=hypot(dx,dy);
52     if(dist<c.r) return IN;
53     else if(dist>c.r) return OUT;
54     else return BOUNDARY;
55 }

```

リスト 9.2 とリスト 9.5 はほとんど同じプログラムである。違うのは、次に示すところと列挙型の定義がある所だけである (冒頭の行数はリスト 9.5 での行数を示す)。

l.20 リスト 9.2 の l.14 に対応。関数 `check` の型が違う。

l.25 リスト 9.2 の l.19 に対応。 `flag` の型が違う。

l.32 リスト 9.2 の l.26 に対応。 `case` の値が違う。

l.35 リスト 9.2 の l.29 に対応。 `case` の値が違う。

l.38 リスト 9.2 の l.32 に対応。 `case` の値が違う。

l.50 リスト 9.2 の l.44 に対応。 返却値が違う。

l.51 リスト 9.2 の l.45 に対応。 返却値が違う。

l.52 リスト 9.2 の l.46 に対応。 返却値が違う。

【列挙型の宣言と定義】

列挙型の宣言・定義方法は、ほとんど構造体・共用体と同じである。例えば、列挙型変数の宣言方法や、列挙型タグなどの扱いについては構造体・共用体と同じで、ただタグの前が `enum` に変わっただけである¹⁰。だが、列挙型を詳述する `{}` の中は話が違い、コンマ区切りである。

列挙型の宣言

列挙型を宣言する際には

```
enum 列挙型タグ名{
    列挙名 1=整数定数値 1,
    列挙名 2=整数定数値 2,
    :
    列挙名 n=整数定数値 n
};
```

の形式で行う。なお、C99 においては最後の整数定数値 (整数定数値 `n`) の後にコンマを置くことが許される。

このようにして宣言された列挙型は、原則コード中で列挙名の形式で扱われる。例えば、リスト 9.5 で `enum place` であるところは、全て `IN,OUT` などと書かれている。従って、整数定数値に意味がない場合も少なくない。このような場合、例えば

```
enum place{IN,BOUNDARY,OUT};
```

のように、整数定数値を省いて宣言を行なっても構わない。

逆に整数定数値が必要になる場合はどのような場合かを考えてみよう。実は、列挙型は内部的に整数型として扱われる。従って、リスト 9.5 の l.31 の `switch` の後を `(flag*flag)` のようにすると、このプログラムでは `IN` の場合が出なくなる。これだけでは一見何の意味もないように見えるが、列挙型が演算できると嬉しいこともある。例えば先の曜日の例を考えれば良い。現在の曜日に、何日後であるかの日数を足し、7 による剰余を取れば列挙型が適切な曜日を示す変数になってくれる。また、列挙子の最後に一つ追加で列挙子を置けば、その列挙子の値が個数になる、という活用方法もある (番兵 (sentinel)¹¹ の一種)。

また、整数定数値を明示する場合には、気をつけなければならないこともある。それは、異なる列挙名に同じ整数定数値を割り当ててしまう場合である。この場合でもコンパイルは無事に通るが、違う列挙名であるのに等しかったり、`switch` 文が正しく動作しなかったりする場合が起こる。

以上からわかるとおり、列挙型では必要のない限り、整数定数値の対応付けを明示しなくても良い (寧ろ、エラーを起こさないために、必要外にしないほうが安全かもしれない)。仮に整数定数値を全く決めなかった場合、通常は最初の列挙名を 0 に対応する値と

¹⁰ 勿論、構造体と同じようにタグ名の省略なども可能である。

¹¹ 番兵は、ターミネータと似たようなものであるが、ターミネータが実データに存在しない終端を示す値という定義なのに対し、番兵はターミネータを含み、終端処理を簡単化するための値またはダミーデータのことを示す。例えば、二次元配列に入れられた迷路において、その端を「壁」で囲んでしまった場合などは、壁が番兵である。

してコンパイラが順に整数定数値を定めてくれる。また、値を明示的に定めた後続の列挙名には、値を明示的に定めた列挙名に+1,+2,...した値が自動的に割り当てられる。そのため、余程のことがない限り、全ての整数定数値を記す必要はない。

ここでみたように、列挙型はフラグとして利用すると便利である。また、各列挙名は定数であるため、変数ではなく定数であることを明示するために大文字ばかりで名前を付ける場合も少なくない。大文字ばかりで名前をつける、と言えはすぐに思いつくのがマクロであろう。確かに、マクロと同様にフラグなどに用いることができるが、マクロと違う点として

- 列挙型は整数型であり、整数以外の値を置換するわけではない。
- 列挙型の実体はあくまで変数及び値であり、コンパイル時に処理されるわけではない。

などが挙げられる。この点に注意して活用されたい。

最後まで触れなかったが、列挙名やメンバ名、タグ名等は全て識別子であるので、識別子の命名規則に則って名付けを行わなければならない。また、違う構造体/共用体/列挙型のメンバ(列挙名)が同一であっても良いし、構造体/共用体のメンバと同じ名前の変数を用いたり、(列挙型も含めて)タグ名と同じ名前の識別子を用いても構わない(タグ名の前につく struct や union,enum で識別可能であるため)。

本講の要点

本講では、構造体・共用体・列挙型について学んだ。

共通の性質

- 構造体・共用体・列挙型の型の名前は、”(struct/union/enum)+タグ名”である。
- 宣言時にはタグ名を省略でき、その場合はタグ名が内部で自動的に決まる。

構造体

- 構造体はひとつの実体に複数の属性をもたせた、変数の集合体である。
- 構造体は同一の型でなければ代入ができない。簡単に代入を行うためには複合リテラルを活用する。
- 構造体(共用体も)のメンバの参照には直接選択演算子.を用いる。
- 構造体のメモリ上での配置は、後ろのメンバのほうの後側に配置される以外に定まった規則はない。多くの場合、4バイト毎に区切りが定まっており、その区切りを必要以上にまたがないように「穴」を作るため、構造体メンバの連続性は保証されない。
- 構造体を関数に渡す場合は値渡しである。
- 構造体は様々な派生型を関数とやり取りする際のオブジェクトの役目を果たす。
- 構造体を用いる際にはint型を任意のビット数に分けて用いることができる(ビットフィールド)。
- 構造体を用いる際には関係性の見えるものをひとつにまとめなければならない。

共用体

- 共用体はビット列に複数の読み方を当てはめられるようにしたものである。
- 共用体の初期化は原則先頭メンバで行われる。ただし、指示付き初期化子を用いて別のメンバを用いることもできる。

列挙型

- 列挙型は取りうる値に名前をつけ、列挙したものである。
- 列挙型は内部的に整数型として扱われ、整数に対する演算と同じ演算を行うことができる。

第10講 派生型(3) ポインタの概念

いよいよ、C言語において山場とされるポインタに入る。ポインタは、その広がりを考え、今回その基礎を説明し、次回は活用方法について学んでいく。

ポインタは難しいと一般に言われるが、あくまでも変数の一種であり、派生型の一種である。それ故、変数であることを念頭において今回の内容を学んでほしい。また、ポインタはここまでに学んできた内容とも密接に関わっている部分であるので、十分に復習しながら、関連付けながら読み進めていただきたい。

10.1 メモリ領域再論

ポインタの解説に入る前に、もう一度メモリ領域について復習しておこう。

C言語で扱う際のメモリ領域には大きく分けて次の4つがあった。

メモリの4領域

- **プログラム領域:** プログラムを実行するためのプログラムコードが置かれる領域。
- **静的領域:** 外部変数や静的変数等の、プログラムの実行の間=寿命となる変数を格納する領域。
- **スタック領域:** 一般の変数、関数の引数や返却値、長い計算式の一時変数などが置かれる領域。
- **ヒープ領域:** プログラム中で動的にメモリが確保される場合に使われる領域。

ポインタを用いれば、これらの領域のうちプログラム領域を除く各領域を扱うことができる。この際、重要なのが、メモリにはアドレス (address) という番号がふられており、これによってメモリ上に配置されているデータの場所が示される、という事である。アドレスはその環境によって「区切り」(アラインメント) が定められている。例えば、メモリは4バイト単位で用いられ、開始アドレスの1の位が一定であったり、といった現象が起こる。また、アドレスは1バイト区切りでしか取得できない。例えばビットフィールドを用いた時、そのメンバへの直接の (scanf などを用いた) 入力はできないし、各メンバの位置が4の倍数の位置になるように穴が空いた。穴が空くのはアラインメントに従って位置を揃えるためである。一方、入力できないのはビットフィールドが特殊な区切りになっているためにアドレスを取得できず、それ故に scanf 関数を用いることができないためである。アドレスを取得できなければ scanf 関数が使えないとは一体どういうことかは、ポインタを学ぶことで理解できる。

10.2 アドレスとポインタ

ポインタは先に少々書いたとおり、アドレスと密接な関係がある。ここではまず、C 言語におけるアドレスの扱いについて学んだ後、それを保持する変数の必要性について論ずることでポインタを自然な形で導入していく。

10.2.1 アドレスの取得と意義

関数を扱う場合、原則的に実引数には影響を与えなかった。だが、例えば「実引数を初期化する関数」や「実引数に何かを代入する関数」等を実現することはできないのであろうか。この問いかけに対して、一つの答えとなっているのが慣れ親しんできた `scanf` 関数である。`scanf` 関数は「実引数に、入力された値を代入する」関数である。これは、関数の章で学んだ「関数は実引数に影響を与えない」ということとは異なっている。これは一体どうしたことか。

翻って `scanf` 関数の引数を見てみると、`scanf` 関数の引数には `&` という怪しげな記号が付いていることがわかる。これが全ての魔法の種である。では、早速この `&` について見ていくことにしよう。

【配列のアドレス確認】

配列の各要素が連続的に配置されているかどうかを、配列の各要素のアドレスを出力することによって確認する。ここでは 1 バイトであり連続性がわかりやすい `char` 型配列を用いる。

リスト 10.1: 配列のアドレス確認

```
1 #include<stdio.h>
2
3 int main(void){
4     char array[10];
5     for(i=0;i<10;i++)
6         printf("%p\n",&array[i]);
7     return 0;
8 }
```

【メモリの性質の確認】

プログラムそのものの解説に入る前に、出力結果を確認し、配列の性質などを確認しておこう。なお、リスト 10.1 は配列 `array` の各要素のアドレスを出力するプログラムである。

リスト 10.1 を筆者の環境でコンパイルして実行した所、第 1 回の実行では

```
0x7fff3416bda0
0x7fff3416bda1
0x7fff3416bda2
0x7fff3416bda3
0x7fff3416bda4
0x7fff3416bda5
```

```
0x7fff3416bda6
0x7fff3416bda7
0x7fff3416bda8
0x7fff3416bda9
```

となった。もう一度実行してみると、今度は

```
0x7fff0ab542e0
0x7fff0ab542e1
(中略)
0x7fff0ab542e9
```

と出力された。先頭に 0x が付いていることからわかるとおり、アドレスは 16 進数で表されており、単位はバイトである。実際にそれぞれの値を見てみると、第 1 回でも第 2 回でも、たしかに 1 バイト毎にアドレスがかわっており、連続的に配置されていることがわかる。

また、これに加えて何回か実行してみても、筆者の環境では必ず 1 の位が 0 であった。これは、先に書いたように「区切りがあって、始まる値が定まっている」という事に合致している。

更に、もうひとつ注目すべきこととして、実行するたびにアドレスが変わるという事が挙げられる。しかも、何度か実行してみればわかるが、この変化は規則的なものではない。このことこそ、RAM の RA=Random Access の証拠である。Random Access は実行するたびに異なるアドレスが割り当てられるものだった。そして、ここでの出力結果は確かに毎回変わっているのである。

なお、リスト 10.1 の配列の宣言を、例えば int 型にした所、int 型が 4 バイトである筆者の環境では

```
0x7fff69b80c20
0x7fff69b80c24
0x7fff69b80c28
0x7fff69b80c2c
:
```

と 4 バイト毎のアドレスが得られた。これは他の大きさの型でも同様である。

【アドレスの取得方法】

では、アドレスを出力したからくりについて説明していこう。リスト 10.1 において、知らない文法はただ 1 行、*l.6* だけであろう。*l.6* にある %p 書式指定子は、アドレスを出力するための書式指定子であり、対応する可変引数のアドレスを、前述のように 16 進で出力する。そしてもうひとつが、scanf でさんざん用いてきた &(アンパサンド) である。

&はそれを付した項にのみ働く単項演算子で、付したオブジェクトのアドレスを取得す

る演算子であり、アドレス演算子 (address operator) と呼ばれる。一般の変数や構造体 (共用体・列挙型) 変数及びそのメンバ¹、配列の各要素については、&を付すだけでアドレスを取得することができる。

一方、関数のアドレス²や配列のアドレス³を取得したい場合は、この演算子はいらない。関数と配列の場合は、その関数名/配列名のみを記せばその先頭のアドレスになる。これらをまとめておこう。

— アドレスの取得方法 —

あるオブジェクトのアドレスを取得する場合は通常

&オブジェクト

によってアドレスを取得する。但し、関数及び配列のアドレスを取得したい場合は

関数 (配列) 名

と、単に関数/配列の名前だけを記す。

なお、ビットフィールドはバイト区切りになっていないため、レジスタ変数はそもそもメモリ上に置かれていないため、アドレスを取得することはできない。

また、&はアドレス演算子である他に二項演算子としてビット毎のアンド演算を行う算術演算子でもある。これは文脈から読み取る必要があるわけだが、&の直前に項があるかないかを注意深く見て、混同しないようにしなければならない。

【何故アドレスを用いるのか】

先に習った&演算子は scanf の第2引数以降に、ここまで天下り的に「記すこと」と書いてきた&そのものである。このことは先にも触れたが、何故これによって実引数の実体に影響を及ぼすことができるのかを考えてみよう。

実体を渡す場合 (値渡し) とアドレスを渡す場合 (アドレス渡し) の違いは何だろうか。値を渡す場合、実引数の情報はただ値だけしか渡されない。元の型をキャストして渡したものなのか、計算式によって計算されたものなのか、それとも予定通りの型の変数そのものなのかは分からない。一方、アドレスを渡した場合には、そのオブジェクトの場所がわかり、ひいてはその実体にアクセスできる。換言すれば、値渡しの場合、変数という箱の中身しかわからず、どのような箱かを知ることができないが、アドレス渡しの場合は元の箱の場所がわかるためそれを直接弄ることができる、という事である。

¹共用体のメンバのアドレスは定義から考えてわかるとおり、何れも同じである。従って、単にアドレスを見ただけであれば共用体に直に&を付しても良い。

²関数にアドレスがあるというのは、一見想像しにくいかもしれない。だが、関数もスタック領域にあるという事、関数型は派生型の一種であることを思い返せば、関数にもアドレスがあってしかるべきだろう。

³配列のアドレスは配列の先頭アドレス=配列の先頭の要素であるので、&a[0]としても取得することができるが、「配列全体の先頭アドレス」というニュアンスが伝わりにくくなるため推奨し難い。

より厳密に値渡しとアドレス渡しの違いを記そう。

値渡しとアドレス渡しの差異

値渡し 関数がローカルで用意した変数に、元の値だけをコピーして利用する。

アドレス渡し 関数に元の変数の場所を渡し、その場所にある変数=元の変数を直接操作する。

このことからわかるとおり、変数の位置さえわかってしまえば、元の変数を適当な型に当てはめて使うことができるのである。これこそが”何故アドレスを用いるのか”の答えである。

【アドレス渡しはスコープに影響を与えるか】

アドレスを用いて他の関数から元のオブジェクトを操作できるというのは、スコープの考えからするとおかしな話のように思える。スコープの定義は、「そのオブジェクトがコード上のどの位置で有効であるか」であった。だが、これだけでは先の「アドレスを渡せば他の関数から元の引数を弄ることができる」という話と矛盾するように見える。

スコープは、より厳密に言うところ「そのオブジェクトが、その型・識別子で、コード上のどの位置まで通用するか」である。従って、あるオブジェクトがメモリ上にある時、これを他の関数などから「別の名前・別の型で」参照したり、書き込んだりすることは可能かもしれない。要するに、宣言された識別子が、どこまでその宣言通りに動作するかというのがスコープなのである。この「識別子が通用する範囲」のことを**名前空間 (namespace)**という。厳密に言えば、宣言によって名前空間が定まり、その名前空間がスコープになる、というのが本来の名前空間・スコープの考えである。

翻って、別関数にアドレスを渡して元の変数を操作することを考えよう。これはアドレスが渡された段階で別の名前空間に位置する。すなわち、操作する対象は同じなのだが、それを表す識別子(及び型)は異なるのである。main 関数では a という名前であるが、自作関数ではこれを b という名前にする。そして、実際は同じ物を指しているのであるが、main 関数の中では a と呼び表さなければならず、自作関数の中では b と呼び表さなければならない、という事になる。この a および b という名前が通用する範囲が名前空間、スコープなのである。

もう少し別の例を出してみよう。英語の apple と日本語の林檎は同じ物を指している単語である。だが、英語しか使えない人の集団では apple と呼び表さなければならず、日本語しか使えない人の集団では林檎とよび表さなければならない。この apple ないし林檎と呼ばれているものを食べる場合、英語圏の人は apple が口の中に入っていく様子を、日本語圏の人は林檎が口の中に入っていく様子を思い浮かべるだろう。この思い浮かべられた動作は全く同じであるし、対象となっている事物も同じなのであるが、呼び名(=識別子)は異なっている。これが main 関数と他の関数の間でも起こっているのである。そして、apple と言ってどこまで通用するか、林檎と言ってどこまで通用するかを示しているのが名前空間、スコープというものである。

【アドレスを用いた参照の危険性】

先のスコープの議論から、アドレスを用いて別関数からそのオブジェクトに対して操作を施す場合、それは既に異なるスコープに属しているのであり、元々学んできたスコープには何ら影響を与えないという事がわかった。そして、このことが厄介な(あるいは危険な)問題の引き金にもなるのである。

【const 修飾子付き変数への入力】

const 修飾子をつけた変数は Read Only であり書き換えることができない。しかし、それがローカルに宣言されている場合、名前空間が異なることを考慮に入れば別関数にアドレスを渡して操作できるのではないだろうか？この疑問を確かめてみる。

リスト 10.2: const 修飾子付き変数への入力

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

リスト 10.2 をコンパイルした所、筆者の環境では警告こそ出たもののコンパイルそのものには成功した。また、実際にプログラムを実行させてみた所、入力した値を出力するという、const 修飾子を付さない場合と全く同様の動作であった。これでは、const の意味がなくなってしまう。

実は、アドレスを用いれば、本来そのプログラムが参照すると想定されていた領域とは違う「想定外の領域」を参照する/書き換えることもできてしまうのである。ここでは、const 修飾子を用いた例でそれを示したが、同じ事はより大規模に拡張することができる。例えば、起動したプログラムが、ファイアウォールやセキュリティソフトの確保しているメモリ領域を書き換えて、それらのソフトの動作を変えてしまうこともある。通常は OS 等の安全機構が働くのであるが、OS を作ったのも人間であり、穴がないとは言えない。これらの穴を突き、意図的にメモリの改変を起こしたものがマルウェア (Malware)⁴の一種あるいは一機能となるのである。

以上のように、アドレスというのは取り扱いを誤ると大惨事をも引き起こしかねない、ある意味「プログラミングにおける原子力」のようなものである⁵。だが、その一方で、適切に活用した時の恩恵は計り知れない。アドレスを取り扱うときには、そのアドレスの示している部分に一体何があるのか、示しているものを明確にして取り扱わなければならない。さもなくば、誤ったプログラムによりシステムが不安定になることも、最悪システムを再インストールしなければならないような事態に遭遇することも考えられる。

⁴ コンピュータウイルス、トロイの木馬、スパイウェアなどの「悪意のこもった」ソフトウェアのこと。

⁵ この危険性＝取り扱いの難しさ故に、言語によってはこれに代わるより安全な機能を導入し、アドレスを直接的に扱う機構を隠蔽している場合もある。だが、アドレスはコンピュータに非常に近いところに位置するものであるため、これらの「より安全な機能」も内部的にはアドレスを利用しているのである。しかし、少なくとも C 言語ではアドレス (及びポインタ) に代わる機構はなく、むしろこれらが直接扱えることが利点でもあるため、その運用には十分に注意を払わなければならない。

C言語の特徴として、コンピュータに近いところまで扱うことができる、比較的低水準言語に近い言語であるという事を述べた。それ故、C言語は「何でもできる」のである。何でもできてしまうという事は、危険なソフトウェアも作れてしまうという事である。我々プログラマは、その倫理に則ってマルウェアを開発しないのは勿論、プログラミングの際に十分注意を払って、特にフェータルなバグは作りこまないように気をつけなければならない。「ポインタが難しい」と言われる理由は、まさにこの「適切に扱わなければならない…」というところにあるのではないだろうか。

10.2.2 アドレスのための変数＝ポインタ

先に述べたとおり、scanf関数では、アドレスを引数として渡している。引数として渡すためには、それを保持するために変数が必要になる。この、アドレスを格納するための変数こそポインタ (pointer) である。ここではまず、ポインタについて理論的な部分を述べていき、その後実際に利用してみることにする。

【ポインタは何故派生型なのか】

ポインタはアドレスを格納するための変数であると述べたが、これは一見、基本型として用意するべきものには見えないだろうか。int や float といった基本型からどのように派生すればポインタになるのか、「アドレスを格納するための変数」という事から直ちに思いつくだろうか。

アドレスはメモリについている番号であるから、メモリを抜きにしては語れない。メモリについて、もう一度考えてみよう。我々は、変数をどのようにして格納しているのであっただろうか。

答えはもちろん、ビット列として保存している、である。ビット列を定まった規則＝型に従って読むことでその変数の値を得ることができた。共用体はその読み替えを行うための機能であり、キャストは値をそのまま保持しつつ、違う規則でのビット列に変換する機能だった。可変引数リストから値を読み出す va_arg マクロは、第2引数に型を取ることで「何型かわからない」可変引数にビット列解釈の規則を与えていた。このように、メモリに保持されているビット列に意味を与えるために、データ型は必要不可欠である。

では、仮に、ポインタがアドレスだけを保持する型であったらどうだろうか。そのアドレスの指し示す先にあるオブジェクトを正しく把握することができるだろうか。scanf関数で書式指定子により型を与えなければならない事や、アドレスだけではビット列解釈の規則がわからないことを考えればわかるとおり、答えはNoである。ちょうど、所在地がわかっても建物を見なければその形がわからないのと同じように、アドレスだけではオブジェクトを正しく把握することはできないのである。アドレスによりオブジェクトの位置を知り、型によりオブジェクトの輪郭を知る。これによって初めてアドレスを用いてのオブジェクト操作が可能なのである。

であれば、ポインタに「指し示した先のビット列をどう読むべきか」という情報、つまり「指し示したオブジェクトの型」を与えてやれば良い。つまり、”double 型のオブジェ

クトが置かれているアドレスを格納するための変数”として double 型から派生したポインタや、”ある構造体変数が置かれているアドレスを格納するための変数”として構造体を指し示すためのポインタなどが考えられる。ポインタが派生型である理由は、アドレスと共に、指し示している先のオブジェクトをどのような規則に従って読むべきかを示さなければならないからである。

なお、単純にアドレスだけを格納し、その規則はキャストなどによって与えたいという場合もある。このような場合には汎用ポインタ (generic pointer) と呼ばれる、「アドレスを格納するためだけの型」を用いる。汎用ポインタは文法規則で見ると void 型へのポインタとして宣言することになる (宣言については後述)。だが、void 型というのは「持たないこと」を示す型であった。この「持たないこと」から、「特定の型に縛られない」と連想し、void 型へのポインタ=汎用ポインタ、となったものと考えられる⁶。

【ポインタのサイズ】

ポインタはアドレスを保持するための変数であるので、アドレスを保持するための大きさがあれば良く、その大きさは基本型によらない。アドレスは環境毎に決まっており、その上にオブジェクトを配置するのだから、どの基本型からの派生であるかによらず大きさが一定であるのは、想像に難くないだろう。

このポインタの大きさは、近年意識されることの多い、OS や CPU の”32bit”, ”64bit” という言い方によってわかる。実は、この”32bit”, ”64bit” は、どちらもポインタの大きさを示しているのである。従って、32bit 環境でのポインタは 4 バイトだし、64bit 環境でのポインタは 8 バイトである。

ポインタが 32bit であるという事は、アドレスが 32bit で表現されているという事である。32bit といえば、以前 2038 年問題を紹介したが、ポインタでもこのようなオーバーフローが起こりうるのではないだろうか。つまり、アドレスが 32bit で足りなくなる場合というのが考えられるのではないか。

この問題はすでに現実のものとなっている。32bit で表されるアドレスというのは、 2^{32} 個である。1 バイトに 1 つのアドレスを割り当ててるのだから、32bit で扱うことができるのは高々 4GB という事になる。一方、パソコンなどを買いに行くと「32bit の OS では 4GB までしかメモリを認識しない」等と書かれていることが多い。これこそがアドレスの不足である。ポインタが 32bit では、4GB 分のアドレスしか用意できないため、認識できる/使えるメモリの量は 4GB 程度⁷に制限されてしまうのである。

一方で、64bit の場合、他の要因で制限を設けなければ $2^{64}B=16\text{EiB}$ ⁸ ものメモリを扱うことができる。そのため、32bit でのメモリ枯渇問題は 64bit が主流になるにつれて解消されるだろう。

⁶考えられる、と記したのは正式にどのような議論があったのかは不明であるため。

⁷メモリ以外のハードウェア等にアドレスを割り当てないといけない場合があるため、実際は 4GiB まで使えることは少なく、3GB ぐらいまでしか認識しない場合が多い。

⁸エクサバイト。Ei 接頭辞は 2^{60} のことである。k, M, G, T ぐらいは聞いたことがあるだろう。T のあとは P(ペタ), E(エクサ), Z(ゼタ), Y(ヨタ) と続く。後ろに i をつければ、Pi(ペビ), Ei(エクサビ), Zi(ゼビ), Yi(ヨビ) となり、 2^{10} 単位となる。

【ポインタの宣言】

そろそろポインタについて理解を深めたところと思うので、実際にポインタを使ってみよう。

【再帰処理のメモリ確保状態】

再帰関数中のローカル変数のアドレスを出力し、再帰処理がスタック領域を確保している事を確かめる。

【解説】

再帰処理がスタック領域中に確保されているならば、再帰呼び出し毎に確保される再帰関数中のローカル変数のアドレスは順次小さくなっていくはずである。これを出力して確認する。

なお、入力値(整数)の深さの再帰を行うため、あまり大きい値を入れるとスタックオーバーフローしてしまう。逆にこれを利用して、スタックオーバーフロー時のメモリはどれぐらいになっているか確認するのも良い(その際は、

printf を条件付き実行にして間引くと良い。)

リスト 10.3: 再帰のメモリ確保

```
1 #include<stdio.h>
2
3 int n;
4
5 void rec(int k){
6     int *p;
7     p=&k;
8     printf("%d-rec:%p\n",*p,p);
9     if(k<n) rec(*p+1);
10 }
11
12 int main(void){
13     scanf("%d",&n);
14     rec(0);
15     return 0;
16 }
```

リスト 10.3 でポインタを利用しているのは、1.6 である。

ポインタの宣言

ポインタを宣言する際には、

指し示す先の型 * 識別子

と、*をつけて宣言する。なお、汎用ポインタを宣言する際には、「指し示す先の型」を void にすれば良い。

宣言を行う際、*の付け方は幾つかある。識別子につけて

```
int *p;
```

という形式でも良いし

```
int* p;
```

と型名の後ろに記しても良い。また、

```
int * p;
```

のように、スペースを用いて独立させても問題ない。但し、どの場合でも、ポインタを宣言する場合の*は直後の識別子名のみにかかる点に注意しなければならない。例えば、

```
int* p1,p2;
```

と書いた場合、p2 は int 型になる。もしも p2 も int *型にしたい場合には

```
int* p1,*p2;
```

のように、p2 の前にも*を付さなければならない⁹。

【間接演算子】

ポインタにアドレスを代入する場合、リスト 10.3 の l.7 のようにポインタをそのまま記せばよく、右辺がアドレスになるだけである。一方、先までの説明からわかるとおり、ポインタはその指し示した先の実体を操作できるのであるから、「ポインタの示した先のオブジェクトを参照する」機能が必要になる。これを行うのが間接演算子 (indirect operator) である。

ポインタの示すオブジェクトの参照

あるポインタが示しているオブジェクトを参照したい場合には

*ポインタ名

のように、ポインタ識別子の前に*を付す。

これを用いてオブジェクトを直接参照したら代入、演算など元の変数を扱うのと同じ要領で扱うことができる (リスト 10.3 の l.8 や l.9 など)。但し、これはポインタに指し示す先の実体があるからできるのであって、参照先実体が不明であるポインタにこれを用いて参照を行なってはならない。先にも書いたとおり、意図しない箇所の書き換えは重大な問題を引き起こす。それゆえ、ポインタを利用する場合には格納アドレスが指し示す場所を明確にして用いなければならない。

間接演算子の*は宣言に用いた*とは別物の単項演算子である。C 言語で用いられる*には&演算子同様に何種類もの意味があるので、構文などから解釈しなければならないし、別の構文にならないように注意しなければならない。以下に、C 言語中で用いられる*の意味をまとめておく。

⁹この規則のため、マクロを用いて型の別名を定義した場合と typedef を用いて型の別名を定義した場合の結果が異なってくる点に注意されたい。仮に、

```
intp p1,p2;
```

という宣言があった場合、この intp がマクロを用いて

```
#define intp int*
```

と定義されていたら、p1 は int *型、p2 は int 型である。一方、typedef により、

```
typedef int * intp
```

と宣言されていた場合、p1,p2 とも int *型になる。

C 言語中でのアスタリスクの意味

- ポインタ型の単項オペランドを伴い、間接演算子として働く。
- 型名の後ろに付し、ポインタ型への派生を示す。
- 二項オペランドを伴い、掛け算を行う。
- /の前後に付され、コメントの開始/終了を示す。

特に、ポインタ参照先を除数にして割り算を行う場合に注意しよう。

`a/*b;`

などとした場合、これは「aをbの参照先の値で割る」を意図しているのだろうが、コンパイラは/*以降をコメントとみなしてしまい、コンパイルできなくなってしまう。この問題を回避するためにはa/(b)と括弧を付したり、a/ *bとスペースを入れたりして、/と*を独立させれば良い。

ポインタ・アドレス関連の演算を表 10.1 にまとめておこう。

表 10.1: アドレス・実体の関係

変数の種類	実体	アドレス
基本型・構造体・共用体	識別子を書く	&演算子を付す
関数型	識別子後に (引数一覧) を書く	識別子を書く
配列型	識別子後に [要素番号] を書く	識別子を書く
ポインタの中身	*演算子を付す	識別子を書く

注意しなければならないのは、表 10.1 のうち、ポインタの欄のアドレスはあくまでも「ポインタに格納されているアドレス」であり、ポインタそのもののアドレスではない。ポインタも変数であるから、もちろんアドレスを持つ。ポインタそのもののアドレスをとりたい場合には、通常の変数同様&演算子を付せば良い。すなわち、ポインタ p について

`p+i`

などとした場合、この p は p に格納しているアドレスの意味になるが、

`&p+i`

などとした場合、&p は p の置かれているアドレスの意味になる。

ここまでの議論からわかるとおり、&演算子と*演算子は互いに打ち消しあう。すなわち、

`&*p`

などとした場合、これは単に p と書くのと同じという事である。

10.3 ポインタ演算

ポインタ (アドレス) にも演算の機能がある。

【文字列の小文字出力】

入力された文字列の英字をすべて小文字に直して出力するプログラムを作成する。

【解説】

文字列の終端にある NULL 文字をターミネータとして、入力された文字列を全て小文字に変更して出力する。入力には `fgets` 関数を用い、空白などであっても対応できるようにしている。

リスト 10.4: 文字列の小文字出力

```
1 #include<stdio.h>
2 #include<ctype.h>
3
4 int main(void){
5     char str[256];
6     char *p;
7     fgets(str,sizeof(str),stdin);
8     p=str;
9     do{
10         putchar(tolower(*p));
11     }while(*(++p)!='\0');
12     return 0;
13 }
```

ポインタに対して許される演算は、整数の加減算と、ポインタ同士の減算のみである。

【ポインタと整数の加減算】

今回、リスト 10.4 の l.11 で用いているのが、ポインタと整数の加減算 (インクリメント・デクリメントも同様) である。

ポインタと整数の加減算

ポインタ `p` と整数 `i` の加算をする場合、

`p+i`

はポインタ `p` の示すアドレスから、`sizeof(*p) × i` だけ進んだ場所を示す。減算の場合も同様。

ここで用いている `p` のインクリメントは、`p` の指し示している位置から一つ後ろにポインタを送る、という意味になる。最初は `str[0]` を指している `p` であるが、一度インクリメントされる毎に、ここでは `char *` 型であるので、`char` 型変数 1 個分ずつ後ろに送られ、`str[1]`, `str[2]`, ... と順に指し示す位置を変えていっている。これにより、`p` の参照先を変えることで文字列を全て見ているのである。

【ポインタ同士の減算】

一方で、同じ型のポインタ同士の引き算も行うことができる。

ポインタ同士の減算

ポインタ同士の減算は、その2つのポインタの指し示しているアドレスの差異が、そのポインタの派生前の型幾つ分であることを示す。

p1, p2 を共に int* 型とする。p2-p1 が 3 ならば、p1 より p2 が int 型変数 3 個分だけ後ろを指している (*p1 が配列の第 0 要素、*p2 が配列の第 3 要素のような状況) ことになる。先のポインタと整数の演算の関係式を整数について解いたものと思えばよい。

【ポインタを用いた配列アクセス】

ここまでの方法を用いると、ポインタを用いて配列にアクセスすることができるようになる。一般に、配列とポインタの参照の間には次の関係が成り立つ。

ポインタを用いた配列の参照

配列のアドレス (ないし、それを指し示すポインタ) array 及び int 型変数 i に対して

`*(array+i)`

という参照と

`array[i]`

という参照は全く同じものである。

この原理は、配列が連続的に並んでいるという事、配列のアドレスは配列の先頭要素のアドレスであるという事が理解できていれば容易にわかるであろう。間接演算子を用いた表現は array から i 個進んだところの中身は? という意味であるので、もちろん array[i] であるはずである。

実際コンパイラは、array[i] という表現が出てきた場合、これを *(array+i) と同じものだと解釈している。また、要素数を指定しない配列 (不完全配列 (incomplete array)) を引数にとる関数を使った場合、呼び出し側は array[] を *array と等価とみなす。配列がアドレス渡しなのは、この等価とみなすステップがあるためである。

ここで、ちょっとしたマジックを紹介しておこう。配列 array と整数型変数 i について

`i[array]`

と書いてもコンパイラを通り、意図した動作になるのである。なにか配列を使っているプログラムについて、このようにひっくり返してコンパイル・実行してみよう (尚、[] 内が 1 変数でない場合は、[] 内の式に () をつけてひっくり返すこと。例えば、array[a+b] は (a+b)[array] となる。)

これがうまくいく理由こそ、先の置き換えにある。i[array] は内部的に *(i+array) と展開される。この演算は整数とポインタの加算であるので、コンパイルエラーとはならない。ここで、加法演算には交換則が成立するので、2 項を交換して *(array+i) とすれば、array[i] と同じものになり、なるほどたしかにひっくり返しても問題ないという事がわかるだろう。とはいえ、この方法はわかりにくいので、普段使うことは推奨されない。あくまでもマジックとしての紹介である。

【間接演算子の演算順序について】

ここまで、何の気なく括弧をつけて間接演算子を扱ってきたが、この演算順序はどうなっているのだろうか。ここでは、この問題について説明していく。なお、必要に応じて、宣言の差異についても説明するが、利用方法については次講に回す。

間接演算子は単項演算子である。単項演算子は、他の2項演算子などよりも先に評価される。そのため、+などを用いる場合には、括弧を付していたのである。

一方で、配列の要素や関数の引数、構造体のメンバは単項演算子よりも先に評価される。これは、各種派生型を自然に扱えるようにするためなのだが、これが間接演算子と混ざると少し複雑になる。

まず、配列の場合であるが、`*a[5]` はどういう意味であろうか。これは、`a[5]` に格納されているアドレスの参照先、という意味になる。すなわち、`*(a[5])` と解釈される。一方で、`(*a)[5]` という書き方もあり、この場合は `a` に格納されているアドレスの参照先のアドレスから更に5個分進めた場所の値、という意味になる。これは、宣言時においても同様で、`int *a[5]` は、`int *`型の5要素の配列であるが、`int (*a)[5]` は「5要素の `int` 型配列のアドレスを格納するためのポインタ」という意味になる。

次に、関数の場合について説明をしておこう。関数の場合、`(*f)(...)` という記述は `f(...)` という記述と同じとされるため、さほど困ることはない。だが、前者にも意味はあるので、利用法は次講にまわすが、関数へのポインタがあることだけ紹介しておく。実は、宣言時に型 `(*f)(引数の型リスト)` とすることで、型・引数が整合する関数のアドレスを格納するためのポインタを宣言できる。したがって、`(*f)(...)` という記述は、関数ポインタによって示される、参照先の関数の呼び出しを行うという意味となり、間接参照であることを明確にするための記法である。なお、`*f(...)` は、関数 `f` の返却値のアドレスの参照先実体である。

構造体や共用体のメンバについても考える。`*st.mem` などの場合、これは `*(st.mem)` と解釈される。つまり、「メンバのポインタ」の参照先の実体を示す。一方、`(*st).mem` と書けば、構造体(共用体)へのポインタ `st` の参照先にある実体のメンバ `mem` の意味になる。だが、この記法は、よく使われるため、より簡単な表記が用意されている。

間接参照演算子

構造体(共用体)へのポインタ `st` に対し、

`(*st).mem`

という記述は、間接参照演算子(indirection reference operator)またはアロー演算子(arrow operator)と呼ばれる `->` を用いて

`st->mem`

と書き換えられる。

この関係は公式として丸暗記してしまってもよいだろう。

このように、間接演算子の演算順序は、とりわけ派生型と組み合わせるときに厄介なことになりがちである。一般の単項演算子と同じ順序で、他の派生型の書き方に比べて遅いという事、適切に括弧をつけること、そして何より、今用いているポインタが何を指し示しているのか明確にすることにより、間接演算子を適切に扱うことができるのである。

なお、アドレス演算子も同様の演算順序を持つが、こちらは `(&st).mem` などの書き方が許されないため、ここで特筆する必要はないだろう。

最後に、違いをまとめておく。

括弧と間接演算子・宣言

- 関数について

- `*f(...)` は、関数の返却値のアドレスの参照先実体である。もちろん、`int *f(...)` は `int *` 型返却値の関数である。
- `int (*f)(...)` 等で、関数へのポインタを宣言でき、`f(...)` で間接参照可能であるが、明確にする際には `(*f)(...)` と記す。

- 配列について

- 宣言時に `int *a[5]` とした場合は `int *` 型の 5 要素の配列になる。この時は、`*a[3]` で、`a[3]` に格納されているアドレスの参照先実体を示す。
- `int (*a)[5]` は、5 要素の `int` 型配列へのポインタである。この時は `(*a)[3]` などとすることで、`a` の参照先の `[3]` 要素を示す。

- 構造体・共用体について

- `*st.mem` は `*(st.mem)` と同じで、メンバの参照先実体を示す。
- `(*st).mem` は `st->mem` と同じで、ポインタの参照先実体の構造体 (共用体) のメンバを示す。

10.4 ポインタを用いる関数

ここでは、ポインタを利用する関数について説明する。

10.4.1 ポインタを用いる自作関数

自作関数の引数や返却値にポインタを用いる場合、型が違う事を除けば通常の基本型を用いた関数と同じ手順で関数を作れば良い。

【二つの変数の値を交換する関数】

二つの int 型の値を交換する関数を作成する。

【解説】

元の変数に影響を与えるため、ポインタを引数に使う必要があることはわかることだろう。ポインタ引数によってアドレスを取得し、その中身を交換するという処理を記述する。この時、作成したい関数は交換処理のみで返却値が不要なので、void 型としている。

リスト 10.5: 交換を行う関数

```
1 #include<stdio.h>
2
3 void swap(int *a,int *b);
4
5 int main(void){
6     int a=3,b=5;
7     swap(&a,&b);
8     printf("a=%d,b=%d\n",a,b);
9     return 0;
10 }
11
12 void swap(int *a,int *b){
13     int tmp=*a;
14     *a=*b;
15     *b=tmp;
16 }
```

【ポインタ引数関数】

ポインタを引数に取る関数はリスト 10.5 の l.3 ないし l.12 のように、引数をポインタ型にすれば良い。関数の章で学んだ通り、仮引数は通常の変数の宣言と同様で、初期値が実引数の値になるものであるため、このように通常の宣言と同様に書けば良いのである。

ポインタ引数関数を呼び出す際には、scanf 関数同様に、&を付けるなどして、アドレスを引数として渡してやる必要がある。本来であれば、このアドレスが実引数であるので、冒頭に記した”実引数に影響を与える”というのは些か不正確な表現である。正しく言えば、”実引数に影響を与えることはできないため、アドレスを渡して間接的に変数そのものに影響を与えるようにした”となる。

【ポインタを返却値に取る関数】

ポインタを返却値に取る関数も、基本型などを返却値に取る型と同様に

```
int * func(...)
```

のように、ポインタ型を用いて宣言すれば良い。後は普段通りに関数を書けば良い。

ポインタを返却値にする場合、その示した先の変数の寿命に気をつけなければならない。例えば、関数内の局所変数のアドレスを返した場合を考えてみよう。この場合、関数が終了するとその局所変数はメモリから取り除かれる。だが、返却値であるポインタはその「取り除かれた場所」を示したままになっている。そこを参照しようとする、当然セグメンテーション違反になってしまう。従って、ポインタを返却値に取る場合は、寿命が尽きた部分のメモリを参照しないようにしなければならない。

10.4.2 ポインタを用いる標準ライブラリ関数

ポインタを取り扱う標準ライブラリ関数は数多くある。実際にどのような関数があるかは付録に譲り、ここでは文字列に関する例を出してポインタの利用方法について説明する。

文字列を扱う関数は、先に書いたとおり”不完全配列はポインタとして扱われる”規則のため、ポインタ引数のうまい活用で様々な扱うことができる。例えば `strncpy` 関数を用いて、文字列 `str1` の `m1` 文字目から `m2` 文字目までを `str2` にコピーしたい場合

```
strncpy(str2, str1+m1, m2-m1)
```

を実行すれば良い。

【NULL ポインタ】

ポインタを扱う関数の説明に先立ち、”条件に合う部分がない”などの際によく使われる **NULL ポインタ** (NULL pointer) について紹介しておく。

NULL ポインタは値が0のポインタとして定義されることが多い、”何をも指し示さない特殊なポインタ”で、ターミネータやエラー処理に使われる。NULL ポインタは幾つかのヘッダにおいてマクロ `NULL` で定義されており、これと比較することで NULL ポインタでないかどうかを判別することができる。

NULL ポインタと NULL 文字は名前こそ似ているものの、全く別物である。前者はポインタ型であり、プログラム中では `NULL` と書く。一方、NULL 文字は文字コード0の文字型定数で、プログラム中では `\0` と記される。もしも誤って、文字列 `str` に対して

```
while(str++!=NULL)
```

などとした場合、これは無限ループになってしまう(それ以前に、コンパイラで警告を出してくれたりエラーになったりすることのほうが多いが)。とりわけ初心者が多い間違いであるので、NULL 文字と NULL ポインタはきちんと区別するように心がけたほうが良いだろう。

【文字列関数について】

文字列関連の関数には、ポインタを用いる関数が多い。これは、文字列を配列として与えるよりも、途中の部分を示すことが可能なポインタで渡したほうが楽だからである。それ故、文字列を扱う場合には、ポインタを用いた配列アクセスの手法が役に立つ場合が多い。実際、ポインタと、ターミネータが NULL 文字であることを利用すれば、文字列関数の大半は実装できるのである¹⁰。以下の関数はこのことを踏まえ、「文字列をポインタとして使う」ことを前提に見てほしい。

¹⁰ 試みに、いくつかの関数を実装してみると練習になる。また、自分自身で文字列関連の関数を作る際にも、ポインタと NULL 文字の判定だけで作ったほうが、メモリが少なかったりアクセスが速くなったり汎用性が高くなったりするので、おすすめである(主に汎用性のためであるが)。

【文字列の分割】

以下、ポインタを用いた関数の例を2例紹介する。第1の例は strtok 関数を用いた文字列の分割である。

【多項式の分割】

展開・整理された、括弧がない多項式を、+や-を目印に区切って項別に出力するプログラムを作成する(但し、符号は消える)。入力の例としては

$$x^3+ax^2-3x+1$$

などで、出力は以下のようになる。

x^3
 ax^2
 $3x$
 1

リスト 10.6: 多項式の分割

```
1 #include<stdio.h>
2 #include<string.h>
3
4 int main(void){
5     char src[256], *p;
6     fgets(str,sizeof(str)-1,stdin);
7
8     p=strtok(str, "+-");
9     while(p!=NULL){
10         puts(p);
11         p=strtok(NULL, "+-");
12     }
13     return 0;
14 }
```

ここで用いた strtok 関数は第1引数に指定した文字列から、第2引数に指定した文字集合に含まれる文字を探し、見つかったらそれを NULL 文字に置き換える、という動作を行う。そして、置き換えた動作を行った場合または文字列終端にきた場合、呼び出し時の文字列への先頭ポインタを返す。

この strtok 関数は内部状態を持ち、2度目以降の呼び出しの際の第1引数に NULL ポインタを指定することによって、先に区切った続きの部分を探索してくれる。これを繰り返していくと、最終的に探索開始位置から文字列終端までに区切り文字が見つからなくなる。この状態で再度 strtok を呼び出すと、今度は NULL ポインタを返してくれるので、それによって終了を判別すれば良い。以下にまとめよう。

strtok の動作

1. 初回の呼び出しでは、リスト 10.6 の l.8 のように元の文字列と区切り文字を引数に指定する。この時、文字列の先頭から区切り文字を探す。
2. 2回目以降の呼び出しでは、リスト 10.6 の l.11 のように第1引数に NULL を指定する。この時、前回 NULL 文字を書いたところの続きから区切り文字を探す。
3. 区切り文字が見つかったらそれを NULL 文字に置き換え、呼び出しに使われた文字列のポインタを返す。
4. 区切り文字が見つけれなかった場合は、探索開始位置を返却するだけである。この次に strtok 関数を呼び出すと、NULL が返却される。

なお、文字列 "A+BC-D" を +- の記号で分割する場合に、文字列がどのように書き換えら

れるかを図 10.1 に示しておく。

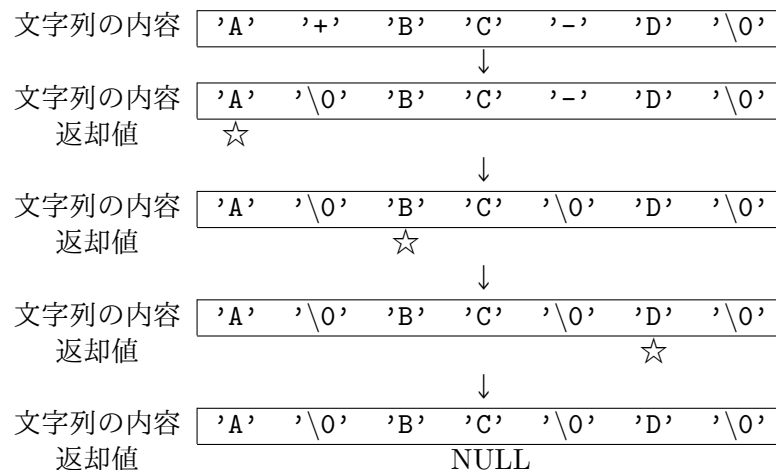


図 10.1: strtok 関数による文字列の分割

【整数/小数の判別】

もう一例、今度は入力の工夫を見てみよう。

【整数/小数を判別して型を選ぶ】

入力される数が整数か小数かを判別して代入し、出力する。

【解説】

整数型変数と浮動小数点数型変数をいちいち用意するのはメモリの無駄遣いであるので、ここでは共用体を用いている。浮動小数点数を使うべきかどうかは、入力に「小数を表す.(小数点)」があるかないかで判別できるので、これを文字列内から検索して、見つからなければ整数、見つければ浮動小数点数としている。ポインタ自体は用意していないが、アドレス渡し関数が多い点に注目されたい。

リスト 10.7: 整数/小数の判別

```
1 #include<stdio.h>
2 #include<stdlib.h>
3 #include<string.h>
4
5 int main(void){
6     char input[16];
7     union{
8         int i;
9         float f;
10    }num;
11    scanf("%15s%c",input);
12    if(strchr(input,'.')==NULL){
13        num.i=strtol(input,NULL);
14        printf("%d\n",num.i);
15    }else{
16        num.f=strtof(input,NULL);
17        printf("%f\n",num.f);
18    }
19    return 0;
20 }
```

ここで用いた”ポインタを用いる関数”は `strchr` 関数と `strtol`/`strtof` 関数である。以下、これについて説明する。

strchr 関数は第 1 引数の文字列の先頭から順に第 2 引数の文字を探索し、それが見つければその場所のアドレスを、見つからなければ NULL ポインタを返す関数である。複数ある場合は最初に見つかったものを返す。逆に、後側から順に文字を探す strrchr 関数や、文字列から文字列を探す (第 1 引数の文字列に第 2 引数の文字列が含まれているかどうかを調べる) strstr 関数もあり、これらは全て最初に見つかった場所のアドレスを返す関数である (見つからなければ NULL ポインタ)。従って、リスト 10.7 の l.12 は、文字列 input の中に . がなければ、という意味の if 文になる。なお、strchr で見つかった文字がその文字列の文字の何文字目にあるかを知りたい場合は

```
char *p;
if((p=strchr(str,c))!=NULL) i=p-str;
```

というように、ポインタの引き算を利用すれば良い (なお、str は文字列、c は文字、i は int 型変数である。)

strtol などの strtol*系関数は、*で示される型に文字列を変換する関数で、ato*関数の上位互換である。ここで紹介した strtol 関数は atoi 関数の、strtod 関数は atof 関数のバージョンアップ版と言えるものである。この関数は、ato*関数に第 2 引数を追加したもので、第 2 引数には文字型へのポインタのアドレスを記す。すなわち

```
char *p;
strtol(str,&p);
```

のような形式である。これにより、p に、該当する数字を変換したその直後のアドレスが代入される。すなわち、strtol*関数は

- 返却値として第 1 引数の文字列を変換した値を返し、(ここまで ato*と同じ)
- 第 2 引数のポインタにその変換した直後のポインタを返す。

という、atoi を多機能化したものなのである。ここで使ったように第 2 引数を NULL ポインタにすれば strtol*関数は ato*関数とほぼ同じ働きをする。但し、ato*関数と違い、指数表現にも対応しているなどの点が異なる。通常は strtol*関数のほうが高機能であるので、こちらにお世話になる方が多いだろう。

本講の要点

本講ではアドレス・ポインタについて学び、その基本的な使い方を概観した。

アドレスとポインタ

- アドレスやポインタを扱う際には、その指示アドレスを明確にして利用しなければならない。
- アドレスを取得する場合、その識別子に&を付す。但し、関数や配列の場合はその識別子のみを記す。
- ポインタはアドレスを格納するための変数であり、その位置にある実体を操作するのに用いる。
- ポインタは指示先実体の型に対する派生であり、これによって間接演算子で正しい型を当てはめて参照できるようにしている。
- アドレスを用いて別の関数に渡した変数は、同一実体でも違うスコープの変数として扱われる。
- ポインタのサイズはOS/CPUのビット数で、これにより認識メモリ量が定まる。
- ポインタを宣言する際は、複数宣言する際も識別子毎に*を付す。

ポインタの利用

- ポインタ (アドレス) に整数 n を加減した場合、その指示位置から派生元の型 n 個分前後した位置を示す。
- ポインタ (アドレス) 同士の減算を行った場合、前側の項の指示位置が後側の項の指示位置から、派生元の型幾つ分前後の位置にあるかを整数で返す。
- $*(a+i)$ と $a[i]$ は同じものと解釈される。
- 間接参照演算子は演算順序に注意しなければならない。この問題を解決する一つの手段が \rightarrow 演算子である。
- ポインタを引数に取る関数を作ることで、呼び出し元関数のオブジェクトを操作できる。
- 返却値がポインタ型である関数を作る際には、その指示先オブジェクトの寿命に注意しなければならない。
- NULL ポインタは「何をも指し示さないポインタ」で、エラー処理などに用いられる。

第11講 派生型(4) ポインタの活用

前講で学んだポインタを活用することで、C言語では非常に多くの機能を実現することができる。ここではポインタをより活用する方法について学んでいく。

11.1 各種派生型へのポインタ

ポインタは基本型に用いてももちろん有用であるが、各種の派生型と組み合わせる、すなわち派生型へのポインタを作ることにより多くの機能を実現できる。ここでは、派生型に対する様々なポインタを紹介し、派生型全体への理解を深めることにしよう。

11.1.1 配列とポインタの関係～ポインタは配列エイリアスか～

配列とポインタは似たものとされ、しばしば混同されていることもあるが、実際には別の概念である。ここでは、ポインタを用いて一次元配列を利用する方法を学び、その違いについて考えていくことにしよう。

【配列の総和 (積み残し誤差回避型)】

積み残し誤差を回避するため、分割統治法 (Divide and conquer algorithm, D&C) を用いて総和を計算する関数を作成する。

【解説】

配列を2項ずつまとめ、次いでその「2項ずつまとめたもの」を2つずつまとめ…を繰り返していくと、最終的には総和になる。この計算においては、和を計算する2項の絶対値の差異が緩和されやすく、積み残し誤差が起こりにくくなると言える。これを逆に見れば、総和→半分ずつの総和の和→四半分ずつの総和の和の和というように、分割してその部分の和を計算する再帰ができることがわかる。このように大きな問題をその部分問題に分割して解く方法が分割統治法である。

なお、ここで作成する関数は、総和を取りたい区間の先頭アドレスと終端アドレスを引数に取るものとする。

リスト 11.1: 分割統治法による総和

```
1 #include<stdio.h>
2
3 double rec_sum(double *from,double *to){
4     int tmp;
5     if(from==to) return *from;
6     tmp=(to-from)/2;
7     return rec_sum(from,from+tmp)+rec_sum(from+tmp+1,to);
8 }
9
10 int main(void){
11     double array[]={1E3,2.2E2,3.8,-4.66,5.2,0.0001};
12     printf("%f\n",rec_sum(array,array+5));
13     return 0;
14 }
```

【配列エイリアスとしてのポインタ】

先のリスト 11.1 のプログラムにおいて用いられているポインタは配列を用いることを前提としたポインタ、いわば「配列の別名」=エイリアスとしてのポインタである。配列はメモリ上に連続的に並んでいるという特性から、ポインタを用いて容易に配列の各要素にアクセスできる。配列引数の関数は、呼び出し側からは、それと等価なポインタ引数の関数とみなされる¹し、`[]`を用いた構文は、コンパイル時に`*(+)`を用いた構文に置き換えられたりする。ここまでは前講に説明した。

翻ってリスト 11.1 を見てみれば、これは配列を扱うためのポインタを用意して、これを用いて総和处理を行なっている。始点と終点を与えることによって比較的簡単に部分和を計算することもできる。このように、ポインタを配列の読替えとして用いる場合は少なくない。だが、このリスト 11.1 の `rec_sum` 関数は、「違う配列であっても」「連続したメモリ領域であれば」総和をとってしまう。構造体を考えてみれば、

```
struct tag{
    double a[20];
    double b[20];
}st;
```

などとしておけば、穴のない限り `a` と `b` にまたがった総和を計算できてしまうのである。これを防ぐためにも、前講で口を酸っぱくしていったとおり、ポインタは何を指し示しているか明確にして使わなければならないのである。

¹完全に等価、というわけではない点に注意。定義している関数内ではやはりポインタと配列は別物として扱われていて、左辺値にできないなどの差異がある。後で学ぶ多次元配列の場合には、もっと大きな違いが見られる。

【restrict 修飾子】

先のリスト 11.1 は同じ配列を 2 つのポインタで扱った。これとは逆に、引数の 2 つのポインタが各々違う配列を示す場合がある。引数の 2 つの配列が違っているとわかっていれば、書ける処理が増えるし、最適化もしやすくなる。これを実現するのが C99 において導入された **restrict** 修飾子である。restrict 修飾子は関数の引数としてポインタを用いる場合に限り利用される修飾子で、2 つ以上のポインタが指し示す領域 (2 つのポインタを用いてその関数内でアクセスされる領域) に重複がない、という事をコンパイラに知らせる役目を担っている。これにより、コンパイルの際の最適化が十分に行われ、プログラムの実行が速くなることがある。なお、restrict 修飾子をつけたからと言って、重複領域へのアクセスがコンパイルエラーとなるわけではないので、その点には注意されたい。

restrict 修飾子を用いる場合は、関数の引数において

```
type func(restrict type *p1, restrict type *p2)
```

のように、型名の前に restrict 修飾子を付せば良い。

【配列オフセットの変更】

C の配列は 0-offset である。これは、ポインタと配列の対応を考えれば自然に理解できることであるが、これを逆用して、配列を 1-offset にすることも可能である。また、数学などで正負両方を取りたい場合などに、配列の添字を正負両方に伸ばすようなことも可能である。これは、実に単純で

```
int a[5], *b;  
b=a-1;
```

などとすれば良いのである²。この場合、b[0] を参照しようとするすると配列外参照になるので注意されたい。この応用例を見てみよう。

【Pascal の三角形】

Pascal の三角形を出力するプログラムを作成する。

【解説】

パスカルの三角形の何段目まで出力するかは定数マクロ N によって規定するものとする。例えば N が 3 であれば、これは $(x+1)^2$ の係数を並べたところまで出力する、という事になる。つまり、 $(x+1)^{N-1}$ の係数を並べた段までを出力するという事である。なお、配列の添字はいずれも 0-offset であり、指数に対応したものであることを付記しておく。

²ただし、1-offset にしたい場合は、配列のサイズを少し広げて、0 を使わないという前提でコーディングしたほうが手っ取り早い。どちらかといえば、負の項の配列などを作りたい場合に便利な手法であろう。

リスト 11.2: Pascal の三角形

```
1 #include<stdio.h>
2
3 #define N 17
4
5 int main(void){
6     unsigned int array[N*(N+1)/2];
7     unsigned int *p[N];
8     int i,j,k;
9
10    for(i=0,k=0;i<N;i++){
11        p[i]=array+k;
12        k+=(i+1);
13    }
14
15    for(i=0;i<N;i++){
16        for(j=0;j<=i;j++){
17            p[i][j]=(j==0 || j==i)?1:(p[i-1][j]+p[i-1][j-1]);
18        }
19        for(i=0;i<N;i++){
20            for(j=0;j<=i;j++){
21                printf("%5u%c",p[i][j],(j==i)?'\n':' ');
22            }
23        }
24    }
```

リスト 11.2 では、ポインタの配列 `p` を用意し、それに対して、`array` の適切な位置を割り当てて、不揃いな 2 次元配列を実現している。いわば、一本の配列を切って分けた、という形である。その、分けたものを順に `p[0]`, `p[1]`, ... とすることによってあたかも 2 次元配列であるかのように利用しているのである。これも配列 (の一部分) に名前をつけているという意味で、ポインタを配列エイリアスとして利用している例と言えるだろう。なお、蛇足ながら、後半に出てくる `p[i][j]` は、`*(p[i]+j)` と書いたほうが、`p` がポインタであることが明確になってわかりやすいかもしれない。

【配列とポインタの違い】

ここまでの説明で、ポインタは配列のエイリアスとして使えるという事を記してきたが、ポインタは決して配列エイリアスのためだけにあるのではないし、ポインタと配列は違うものである。

配列とポインタの最たる違いは、配列はそれが定義された段階でその大きさ分のメモリが確保されるが、ポインタは他で実体を用意しなければならない、という点である。先までの例でも、配列を前もって用意しておいて、それに対するポインタを用いることでアクセスをしているのである。決して、ポインタを宣言することによって自動的に実体ができるのではない。

また、ポインタはあくまでもアドレスを格納するための変数である、という事を忘れてはならない。配列エイリアスとしてポインタを用いた時、先の実体確保を忘れるのと同じくらいよく間違えるのが `sizeof` 演算子関連である。配列の場合は `sizeof` で全体のサイズを取ることができ、それによって個数を知ることができた。だが、ポインタでは、`sizeof` を用いても配列全体の大きさを取ることができず、環境によって定まっているポインタの大きさが返ってくるだけである。

このように、ポインタは配列エイリアスとして用いることもできるのだが、決して配列と同じものではない、という点を理解しておきたい。

【浅いコピーと深いコピー】

配列をベクトルとしてみなして計算する場合など、配列そのものを交換したりコピーしたりする必要が出てくる。この時(特に交換するとき)には、ポインタをエイリアスとして用いた浅いコピー (shallow copy) を用いると高速に交換できる。だが、`memcpy` 関数などを用いて行う深いコピー (deep copy) を用いる場合とは違った問題も生ずる。

浅いコピーはポインタの間でその参照先アドレスをコピーすることによって、あたかもコピーしたかのように見せる方法である。例えば、

```
int array[2][5];
int *p[2],*tmp;
p[0]=array[0],p[1]=array[1];
tmp=p[0],p[0]=p[1],p[1]=tmp;
```

などとすれば、最初は `array[0]` を参照しているポインタ `p[0]` が、今度は `array[1]` を指すようになる。このように、配列エイリアスとしてのポインタを用意し、その参照先を変えることで役目を変えれば、繰り返し処理などが書きやすくなるという利点がある。だが、浅いコピーはあくまでもアドレスのコピーであるので、その実体はひとつしかないという点に注意しなければならない。つまり、

```
int array[5],*p[2];
p[0]=p[1]=array;
```

とした場合に、`p[0]` はたしかに `p[1]` のコピーであるが、`*(p[1]+3)=3` のような `p[1]` 側の書き換えが `p[0]` にも影響を及ぼす。その一方で、浅いコピーはアドレスだけのコピーであるので、配列の大きさに依存せず高速にコピーすることができる(定数時間)。

一方で、深いコピーは実体のコピーを行う方法である。先に浅いコピーにより2つの配列を交換する方法を見せたが、これを深いコピーで実装すると次のようになる。

```
int array[2][5],i,tmp;
for(i=0;i<5;i++){
    tmp=array[0][i];
    array[0][i]=array[1][i];
    array[1][i]=tmp;
}
```

のようになる。ここではメモリの節約のために、別の変数をひとつだけ用意したが、交換用に別の配列を用意して、それに一旦コピーした後に…という方法もある(コピーの際には `memcpy` 関数を用いると簡潔に書くことができる)。このように、配列の実体そのものをコピーするのが深いコピーであり、実体を複数作ることができる。そのかわり、コピーには時間がかかる(配列のサイズ n に対して $O(n)$ 程度)ため、何度も交換処理を行うような場合には使いづらい。

このように、配列のコピーには、実体をコピーする深いコピーと、参照のみをコピーする浅いコピーがあり、状況に応じて使い分けなければならない³。

11.1.2 文字列リテラルについて

ポインタは場所を示すため、配列のエイリアスとして用いたり、配列で条件に合致する位置を示したり、という使い方ができた。実際、前講で扱った文字列に関する関数群はポインタと共に利用されていた。これらの利用は、先に扱った配列へのポインタと同じように見なせる。というのも、文字列はあくまでも文字型変数からなる配列だからである。だが、少しだけ違った扱いをするものがある。それが、文字列リテラルである。非常に混同しやすいので、注意してみていくことにしよう。

【もっと Hello World】

Hello World と出力するプログラムを、ポインタを用いて書いてみる。

【解説】

配列 `str` には "hello, " が格納されており、ポインタ `p` は(メモリの別の部分に配置されている) "world" を示している。

リスト 11.3: Hello World(3 回目)

```
1 #include<stdio.h>
2
3 int main(void){
4     char str[]="hello, ";
5     const char *p="world";
6     printf(str);
7     puts(p);
8     return 0;
9 }
```

なお、リスト 11.3 では、`printf` の第 1 引数が書式文字列でも文字列リテラルでもないため、コンパイル時に警告が出るかもしれないが、無視して実行して問題ない。

【文字列リテラルによる初期化】

リスト 11.3 では、`l.4` と `l.5` でそれぞれ文字列リテラルを用いて初期化を行なっている。`l.4` は不完全配列のように見えるため、ポインタと等価であると思ってしまうかもしれないが、それは誤りである。これら 2 つの文は、全く違う処理を行なっている。

³Java などでは、普通にコピーした場合、基本型は深いコピーであるが参照型は浅いコピーになってしまうため、インスタンス(構造体型変数を拡張したようなもの)のコピーの際に注意しなければならないなど、これらのコピーをより強く意識する必要がある。

l.4 の処理は、str という文字型の配列を定義し、それについて、前から順に h,e,... と要素を代入して初期化する処理である。一方、l.5 の処理は、文字列リテラルへのポインタの初期化である。文字列リテラルといえど、データであるからにはメモリ上に配置されている。そのアドレスを p というポインタに代入しているのである。したがって、str[] は自分で定義された実体を持つが、p はあくまでポインタであり、メモリ上に置かれたリテラルへの参照にすぎないのである。すなわち、l.4 の初期化は深いコピーによる初期化、l.5 の初期化は浅いコピーによる初期化である。似たように見える処理であるが、書き換えの可否などで違いを感じることになる。

【ポインタと const】

もうひとつ、l.5 の const が気にならなかったらどうか。const 修飾子は、宣言を読む際に厄介な存在となりがちである。だが、決まりはただひとつ「直後の記号を修飾する」ことを覚えておけばいい。

l.5 に着目すると、const は char の前に付いている。したがって、p は const char 型への、書き換え任意なポインタである。そして、文字列リテラルはいずれも書き換え不能 (const char 型) であるので、これに対するポインタとしては適切なものになっているわけである。このように、文字列リテラルを用いる際には const char であることを忘れないようにしなければならない。

一方で、ポインタを read only にしたい場合はどうすればよいだろうか。この場合は、やはり const が直後を修飾するという決まりを用いて、変数名の直前 (ないし、* の直前) に const を配すれば良いのである。たとえば、

```
const char * const p="str";
const char const * s="chr";
```

などとすれば、p,s はともに Read Only のポインタとなる。ここで記した例では、最初の const は char にかかっており、後側の const は p ないし * にかかっているのである⁴。

以上のように、ポインタと const が絡むと、わかりづらい場合が出てくる。だが、const はすぐ後ろを修飾するという事さえ押さえておけば、構文で困ることはないだろう。最後に、const の位置による違いをまとめておく。

const とポインタ

- const type * ... の記述は、参照先実体が Read Only であるようなアドレスを格納するためのポインタである。
- type * const ... ないし type const * ... の記述は、ポインタの保持するアドレスが Read Only であるようなポインタである。

⁴このテクニックを使っても判別が難しい例として、const char * const p1,*p2 のような宣言がある。この場合の p2 は const char 型への読み書き可能なポインタであるが、わかりづらいので、分けて書いたほうが良いだろう。

11.1.3 関数へのポインタ

関数型をそのまま引数にとったり配列にしたりすることはできない。だが、例えば「入力される値に応じて用いる関数を変える」とか「関数に対して処理を行う関数を作りたい」という場合、引数や配列として使えると便利である。これを実現するのが関数へのポインタである。今述べた事例は、関数を扱うと言ってもその返却値に興味があるのではなく、関数そのものに興味があるのだから、ポインタを使うのは自然なことだろう。

【関数ポインタの宣言と利用】

いきなりだが、関数ポインタの宣言方法を示そう。

関数ポインタの宣言方法

関数へのポインタを宣言する際には、

関数の型 (*ポインタ名)(引数リスト);

とする。これにより、関数の型と引数リストの型及び順序が一致する関数のアドレスを保持できる。

関数ポインタを使うメリットは最初に述べたとおりである。具体例として、関数の配列を作成し、各々の数値積分を行うプログラムを見てみよう。

【関数の台形積分】

幾つかの関数の積分を、台形積分を用いて計算してみる。

【解説】

- 関数 `integral` は、第1引数のポインタで示される関数を、第2引数 `from` から第3引数 `to` の区間で、刻み幅を第4引数 `h` として、台形積分公式により定積分し、その値を返す関数である。
- 台形積分 (trapezoidal integral) 公式は、区間を台形に近似して積分する公式である。小区間 $[a, a+h]$ の積分を台形の面積として近似すれば、

$$\int_a^{a+h} f(x)dx \approx \frac{h}{2} (f(a) + f(a+h))$$

と近似することができる。当然、 h が大きいほど近似精度は悪くなるので、区間 $[a, b]$ での積分は $[a, a+h], [a+h, a+2h], \dots, [a+kh, b]$ のように区切り直し、各々の近似値を求めて総和を出す形で求める。

- 今回は関数のポインタの配列を作成し、cos,cosh,floor,exp,fabs の各関数の積分を計算・出力した。

リスト 11.4: 関数の台形積分

```

1 #include<stdio.h>
2 #include<math.h>
3
4 double integral(double (*func)(double),double from,double to,double h);
5
6 int main(void){
7     double (*funcs[5])(double)={cos,cosh,floor,exp,fabs};
8     double var;
9     int i;
10
11     for(i=0;i<5;i++) printf("%lf\n",integral(funcs[i],-1,1,1.0/1024));
12     return 0;
13 }
14
15 double integral(double (*func)(double),double from,double to,double h){
16     unsigned int i,j,k,n;
17     double ret=0,sect=to-from;
18     for(i=0;i*h<sect;i++)
19         ret+=h*(func(i*h)+func((i+1)*h))/2;
20     return ret;
21 }
```

リスト 11.4 では、先に挙げた「関数の配列化」と「関数引数の関数」を実現している。これを見てみれば、次のようなことがわかるだろう。

関数ポインタの利用

- 関数のアドレスは、その関数の名前だけを書くことで取得できる。
- ポインタで示されている関数も、ポインタ名の後に引数リストを付けることで呼び出すことができる。

これらの特徴のうち、後者については、

`(*func)(...)`

のように、間接演算子を用いても参照可能である。だが、いちいち括弧と間接演算子を付すのは煩わしく、引数リストによって実体かアドレスかが区別できるため、通常関数ポインタの直後に引数リストを記して呼び出す。

なお、ここでは標準関数のポインタを取得したが、自作関数のポインタなども同様に取得できる。また、注意すべき点として、引数リストが一致しない関数のアドレスは、ポイ

ンタに代入することができない(キャストすれば可能であるが、適切に扱われるかどうかは保証されない)。すなわち、関数ポインタは、返却値・引数リストが完全に一致する関数のアドレスのみを保持できる。

【関数ポインタを利用する標準関数】

ここで紹介した関数ポインタを引数にとる標準関数も存在する。atexit 関数や qsort 関数、bsearch 関数などがそうである。紙数の都合上ここでは名前だけの紹介にとどめるが、興味があれば調べられたい(一部は後の講で扱う)。

11.1.4 ポインタへのポインタ

ポインタは変数である。変数であるからには、もちろんアドレスが存在する。それを保持するためのポインタがポインタのポインタ (pointer to pointer) である。以下、「ポインタのポインタ」や「ポインタのポインタの... ポインタ」と記すのは煩わしいので、これらをまとめて多重ポインタと呼ぶことにする。

【多重ポインタの概念と利用】

多重ポインタは、要するに「ポインタ変数のアドレスを格納するためのポインタ変数」である。つまり、指示先にもまたポインタがある、という事である。これは、宣言や間接演算子の際の*を多重にして扱う。ポインタ pp がポインタ p のアドレスを格納しており、ポインタ p が変数 var のアドレスを格納しているならば、

`*(*pp)`

のようにすれば、var の値を参照することができる。同様に、多重ポインタの宣言は

```
int **pp;
```

と、*を重ねてやればよい。これを用いることで、配列エイリアスを多次元配列のように用いることができるなどの利点がある。ここでは、行列を作り、その行交換を行う関数を見てみることにしよう。

【行列の表現例と行交換】

ポインタによるエイリアスを用いて行列を表現し、行交換を行う。

【解説】

- 行列の実体は一次元配列 num で用意しておき、その行を表すポインタとして mat を、全体を表すポインタとして matrix を用意した。
- 交換は浅いコピーによっておこなっている。

リスト 11.5: 行列の表現例と行交換

```
1 #include<stdio.h>
2
3 #define NUM 9
4
5 void lineswap(double **a,double **b);
6
7 int main(void){
8     double nums[NUM*NUM];
9     double *mat[NUM],**matrix;
10    int i,j,k;
11    for(i=0;i<NUM;i++) mat[i]=nums+i*NUM;
12    matrix=mat;
13
14    for(i=0;i<NUM*NUM;i++) nums[i]=i;
15
16    lineswap(&mat[0],&mat[NUM-1]);
17    for(i=0;i<NUM;i++)
18        for(j=0;j<NUM;j++)
19            printf("%2.01f%c",*((matrix+i)+j),(j==NUM-1)?'\n':' ');
20    return 0;
21 }
22
23 void lineswap(double **a,double **b){
24     double *tmp;
25     tmp=*a;
26     *a=*b;
27     *b=tmp;
28 }
```

ポインタが置かれているアドレスの取得には、&演算子を用いるか、ポインタが配列になっているならばその配列名を記せばよい。これは通常の変数と同じであるので問題ないだろう。また、ここではエイリアスのように書かなかったが、1.19のprintfの第2引数はmatrix[i][j]と書いても同じである。

このように、多重ポインタは多次元配列と相性がよく、しばしば多次元配列のように扱われる。だが、両者は全く別物である。今度は、それを説明していくことにする。

【多次元配列へのポインタと多重ポインタの差異】

配列エイリアスとしてのポインタは先に紹介したが、多次元配列とポインタの関係は先にも増して難しい。ここでは、これらの差異を見ていくことにしよう。

以下、ポインタと配列について、次のように仮定して説明を行う。

```
int array[3][5];
```

```
int num[15];
int *p[5]={num,num+3,num+6,num+9,num+12},**pp=p;
```

また、サイズ n の配列型を `int[n]` 型等と記す。

上記の例では、どちらも

```
array[2][3];
pp[2][3];
```

のようにして各要素へのアクセスを行うことができる。このことから、二つの動作は一見同じように見えるのだが、

```
pp=array
```

として、アクセスを試みても、コンパイルエラーになってしまう。同じような記法、同じような動作なのだが、内部では違ったものとして扱われているのである。

まず、多次元配列の場合を考えよう。派生の定義に戻って考えれば、2次元配列 (=配列の配列) とは配列派生を2回行った型であると言える。すなわち、`int` 型を5個連ねて `int[5]` 型とし、これを3個連ねて `int[3][5]` 型の `array` を定義したのである。この派生をたどればわかるとおり、`array[2][3]` という形のアクセスのうち、`array[2]` の部分は、`int[5]` 型へのアドレスである。つまり、`array[2]` は、`array` の先頭アドレスから `int[5]` 型2個分進めた箇所、という意味である。そして、そこから今度は、`int` 型3個分進めた箇所 (の要素) を指し、これによって要素へのアクセスを行なっている。アクセスの状況とその時の型を、図 11.1 のように対比してみるとわかりやすい。

指す型	解釈部分
<code>int[3][5]</code>	<code>array</code>
<code>int[5]</code>	<code>array[2]</code>
<code>int</code>	<code>array[2][3]</code>

図 11.1: 多次元配列へのアクセスとそれが指す型の対比

このアクセス方式や派生過程からわかるとおり多次元配列は必ずその全要素が連続的に配置されている。すなわち、`array[2]` と `array[1]` は、必ず `sizeof(int)*5` バイトの違いを持たねばならない。この `array[1]` と `array[2]` が連続的であるという点は多次元配列にしかない特徴である。

一方、ポインタのポインタを用いたアクセスでは、連続性は各段階においてしか仮定されない。

```
pp[2][3];
```

のコードは、あくまでもポインタとして次のように解釈される: ポインタ `pp` が指している部分から `int *` 型2個分進んだ場所にある `int *` 型のポインタの指示先を `pp[2]` とする。この時、`pp[2]` の指示先から `int` 型3個分進んだ時に、そこにある要素の値を `pp[2][3]`

としてアクセスする。このことから、pp[1] と pp[2] の指示先に連続性は仮定されない。更に、アクセスの途中で用いられている型が違うこともわかるだろう。

ここまでの内容を鑑みれば、int **型のポインタに二次元配列の先頭アドレスを代入できないことは明らかであろう。途中で挟まれるポインタがなく、型がわからない＝飛ぶ量がわからない為、pp=array などとしてしまうと pp[2] が計算不能になってしまう。つまり、pp[2] と書いた時、これは pp から、派生元の型が何であるポインタ 2 個分なのか、コンパイラは判断できないのである。そのため、仮に先の array に対して、そのエイリアスとなるポインタ parray を作るならば、その宣言は次のようにしなければならない。

```
int (*parray)[3];
```

この宣言では、int[3] 型へのポインタとして parray を宣言するため、*parray を括弧でくくっている。このように多次元配列に対するポインタを作る場合は、型を丁寧に解釈し、派生元の型との齟齬が無いように宣言を書かなければならない。

11.2 メモリの動的確保

ここまで、静的配列を用いて実体を確保してきた。だが、スタック領域は決して大きくない上、静的配列は前もって定められた要素数以上使うことはできない。これらの欠点をなくし、ヒープ領域を動的に確保して実現されるのが動的配列 (dynamic array) である。

11.2.1 動的配列の概念と利用法

動的配列は先に述べたとおり、ヒープ領域に確保され、実行時に長さを決めることができる配列である。静的配列に比べて自由度が高いが、有効利用にはポインタへの理解が必要になるし、通常、人間の手で確保/解放⁵を行わなければならない (自由には責任が伴う!) という難点もある。静的配列に比べて自由度が高いという事は、管理をきちんと行わなければならないという事でもあるため、気をつけて扱わなければならない。

【動的配列の利用手順】

動的配列の利用手順は、次のようになる。

動的配列利用の手順

1. 動的配列の確保先アドレスを格納するためのポインタを宣言する。
2. メモリ領域を確保し、その先頭アドレスをポインタに割り当てる。
3. 動的配列を用いた処理を行う。必要に応じて拡大/縮小を行う。
4. 動的配列の利用が終了したらメモリ領域を解放する。

⁵C 言語にはないが、不要になったメモリ領域を自動解放してくれる機能としてガーベッジコレクション (garbage collection, GC) を採用している言語もある。

この手順を見てもらえばわかるとおり、動的配列は静的配列に比べて利用が面倒である。また、静的配列とは違い、配列を取り過ぎてもコンパイルエラーにならないし(当然、配列の個数は実行時に決まるため)、解放せずに終了してしまうとそのメモリが無駄に確保され続けたままになってしまうメモリリーク(memory leak)という現象を起こしてしまう。そのため、動的配列を用いる場合は、次の点に注意しなければならない。

動的配列利用上の注意

- 動的配列確保の際に、確保できなかったという返却値だったら、適切な例外処理を施すこと。
- 動的配列を利用している際は、終了時に必ずその解放を行うこと。^a

^aexit 関数などを用いて終了するようなプログラムの場合、atexit 関数を用いて、終了時の解放処理を登録しておくくと便利である。

なお、動的配列へのアクセス方法は、ポインタを用いたアクセスと同じである。すなわち、ポインタ `p` がある動的配列を指している時、その第 `i` 要素は

`p[i]`

あるいは

`*(p+i)`

のようにして表現することができる。このように、動的配列を取り扱う場合は、ポインタ/配列と同様にして取り扱うことができるのである(特に、配列エイリアスとしてのポインタの扱いはちょうど対応する)。従って、動的配列を学習する上で押さえておかねばならないのは、確保と解放である。

【動的配列の確保と解放】

では、動的配列を用いた例を見てみよう。

【Pascal の三角形再び】

Pascal の三角形を出力するプログラムを動的配列を用いて作成する。

【解説】

今度は、何段目まで出力するか入力してもらい、必要な分だけメモリ確保することにした。ポインタのポインタを用いて多次元配列のような配列を実現しているが、`pp[0], pp[1], ...` の各指示領域に連続性がない点には注意されたい。もしも多次元静的配列同様に連続性のある多次元動的配列を作りたいのであれば、まず一次元配列を確保し、それにエイリアスを付していく形になる。

リスト 11.6: Pascal の三角形 (動的配列版)

```

1 #include<stdio.h>
2 #include<stdlib.h>
3 typedef unsigned int u_int;
4
5 int main(void){
6     u_int **pascal,*pp;
7     int i,j,k,n;
8     scanf("%d",&n);
9
10    if((pascal=(u_int **)malloc(n*sizeof(u_int *)))==NULL) return -1;
11    for(i=0;i<n;i++){
12        if((*(pascal+i)=(u_int *)calloc(i+1,sizeof(u_int)))==NULL){
13            for(j=0;j<i;j++) free(*(pascal+j));
14            free(pascal);
15            return -1;
16        }
17    }
18
19    for(i=0;i<n;i++) for(j=0;j<=i;j++)
20        pascal[i][j]=(j==0 || j==i)?1:(pascal[i-1][j]+pascal[i-1][j-1]);
21    for(i=0;i<n;i++) for(j=0;j<=i;j++)
22        printf("%6u%c",*(*(pascal+i)+j),(j==i)?'\n':' ');
23
24    for(i=0;i<n;i++) free(*(pascal+i));
25    free(pascal);
26    return 0;
27 }

```

ここでまず着目して欲しいのは、l.10 および l.12 の確保部分である。

malloc/calloc によるメモリの確保

メモリを動的に確保し、それをポインタ p に割りつける場合、

p=(型 *)malloc(確保したいサイズ)

ないし、

p=(型 *)calloc(確保したい個数,1 個あたりのサイズ)

の形式で記す。なお、これらの関数は stdlib.h に収録されている。

malloc 関数と calloc 関数は、引数の違いと、calloc 関数に限り確保した領域を 0 で初期化するという点が違う。リスト 11.6 では、malloc/calloc を用いて確保を行い、その返却値が NULL である=確保に失敗した場合、例外処理を行うようにしている。

一方、l.13,14 や l.24,25 に見られる free 関数は解放を行う関数である。

メモリの解放

メモリを解放する場合は、解放したいポインタ p に対して

```
free(p)
```

を実行する。free 関数は stdlib.h に収録されている。

解放の際に注意しなければならないのは、同じ領域を2度以上解放したり、解放が必要ない領域を解放したりといった問題である。解放すべきでない領域を解放しようとする、大抵の場合アクセス違反が起こる (Segmentation fault ではない場合も多い)。従って、丁寧にソースを読み、解放する領域を確認しておくほうが良い。他、例えば動的配列のオフセットを変更する場合、面倒であっても確保用のポインタとアクセス用の (エイリアス用の) ポインタは別に分けたほうが安全である。

また、今回のようにポインタのポインタを用いて多次元配列のような機能を実現している場合は、その解放順序にも注意しなければならない。リスト 11.6 において、先に pascal 全体を解放してしまうと、pascal[0], pascal[1], ... にアクセスする術が失われてしまい、結果メモリリークやアクセスバイオレーションの原因となってしまう。

【動的配列の拡大/縮小】

動的配列の現在の状態はそのまま、動的配列を拡大/縮小したい場合がある。このような場合には realloc 関数 (これも stdlib.h にある) を用いる。

動的配列の拡大/縮小

既に確保されている動的配列 p を拡大/縮小したい場合には、同じ型の別のポインタ tmpp を用意し

```
if((tmpp=(型)realloc(p, 新サイズ))!=NULL) p=tmpp;  
else 例外処理;
```

のように行う (ここでは例外処理も最初から考慮した)。

11.2.2 フレキシブル配列メンバ

動的確保を利用する別の機能として、C99 から導入されたフレキシブル配列メンバ (flexible array member) の機能がある。これについて簡単に説明する。

メンバ2つ以上の構造体を宣言する際、その最終メンバを不完全配列型とし、例えば

```
struct st{  
    int a;  
    int b[];  
};
```

として良い。この時、この構造体型 (ここでは `struct st` 型) へのポインタ `p` を準備し、

```
p=(struct st*)malloc(sizeof(int)*(1+n));
```

とすることで、メンバ `b` に `n` 個の `int` 型領域が割り当てられる。このときの `b` がフレキシブル配列メンバである。

フレキシブル配列メンバには、次のような制約がある。

フレキシブル配列メンバの制約

- 不完全配列型は必ず構造体の最後の要素でなければならない。
- フレキシブル配列メンバを含む構造体の配列は使えない。
- フレキシブル配列メンバを含む構造体は、ほかの構造体の途中にあるメンバとしては使用できない。
- フレキシブル配列には `sizeof` 演算子が適用できない。

11.3 関数間で配列をやり取りするには

一次元静的配列を引数に取る関数については配列を学んだ際に学習した。だが、多次元配列を同様の方法で渡そうとしても関数に渡すことはできない。ここでは多次元配列を関数に渡す方法を説明する。

なお、何れの方法を用いたとしても配列の要素数を自動で取得することはできない⁶。そのため、可変引数関数の作成と同じく、配列の要素数を示す情報を渡さなければならない。可変引数の場合は個数を示す情報にかなりの多様性が見られたが、配列の場合は要素数を直接記す方法が殆どであると言って良い。

11.3.1 一次元配列に帰着させる方法

静的多次元配列は一次元配列の順番を書き換えたものに過ぎない。従って、そもそも多次元配列を用いず、その型へのポインタとして渡してしまい、一次元配列として扱えば良い。警告が出る可能性はあるが、コンパイルは通るし、動作も正しいだろう。だが、この方法は多次元配列を使う利点を消し去ってしまう方法であり、推奨できる方法ではない。勿論、先に書いたポインタによるエイリアスなどを用いて多次元配列のように扱うこともできるのだが、それなら呼び出し元で最初からポインタによるエイリアスを行ったほうが早い(後述)。

⁶C++やJavaでは要素数を自動取得することができる。これは、配列という概念は同じでも、その型に関する実装などが違うためである。Cの配列は、最小限度の機能のみを実装した、「もっとも素朴な (simple な) 配列」といえよう。

11.3.2 要素数を固定して渡す方法

先に学んだ、多次元配列へのポインタを思い出そう。多次元配列を直接多重ポインタに代入できないのは、その型の問題であった。それを考慮すれば、

```
type func(type (*array)[2][3]);  
type func(type array[][2][3]);
```

最初の次元数が省略できるのは、不完全配列がポインタと等価に扱われることを考えれば明らかであろう。もちろん、次元数を省略せずに

```
type func(type array[5][2][3]);
```

のように扱うことも可能である。だが、この方法では、第2次元以降の全要素の個数を固定しなければならない。

11.3.3 配列エイリアスのポインタを用いる

そこで、(動的配列と同様に) 配列エイリアスのポインタを作成し、そのポインタを渡してアクセスしてやれば良い。この方法であれば、静的配列でも動的配列でも用いることができて便利である。例えば、

```
int array[3][5];  
int *p[3]={array[0],array[1],array[2]},**pp=p;
```

などとして、int **型引数を取り、それに pp を渡してやれば一見落着である。

なお、ここまで学んだことをぶち壊すような意見であるが、いくつかの関数で使いたい配列はグローバルにしてやれば、それで話がすむ場合が多い。きちんと管理できるのであれば、グローバル配列を用いてやれば簡潔に記述することができるだろう。特に多次元配列の場合、複雑で混同しやすい引数渡しよりも、グローバル配列にする方が主流である。また、グローバル配列にすることによって、寿命やスコープの取り扱いを気にせずにするようになるため、「関数内で静的な静的配列を宣言してそれへのポインタを返すことで配列を返す」とする⁷より、分かりやすいやり取りを行うことができるようになる。

この他、構造体をオブジェクトとして用いる方法もあるが、値渡ししか参照渡ししかで混乱したり、配列のコピーを作るためにスタック領域を大幅に消費したりといった欠点もあるため、特段なければグローバル配列を用いることを推奨しておきたい。

⁷static 記憶クラス指定子を付さない場合、ポインタの返却後にその領域が解放されてしまうため、既解放領域へのアクセスバイオレーションが起きる。

本講の要点

本講では、ポインタをより実践的に使うために、派生型へのポインタを中心に学んだ。この章でも、次のスローガンをなお念頭に置いて欲しい。

ポインタは指し示しているものを明確に！

派生型へのポインタ

- ポインタを用いて配列エイリアスを実現できるが、配列とポインタは別物である。
- `restrict` 修飾子は引数にとった2つ以上のポインタの指示領域が互いに独立である事を示し、それによる最適化ヒントをコンパイラに渡す。
- 参照元のみのコピーを浅いコピー、参照先実体そのもののコピーを深いコピーといい、挙動が異なる。
- 配列を文字列リテラルで初期化すると各要素に文字が代入されるが、ポインタの場合は `const char` 型へのポインタとしてアドレスが代入されるにすぎない。
- `const` 修飾子は直後の語にかかる。
- 関数に対してもポインタを作成でき、これにより引数や配列に関数を用いられる。
- ポインタに対してもポインタが存在する。
- 多次元配列と多重ポインタは、そのアクセス過程で経由する型や連続性の保証の点などで異なっている。
- 多次元配列と多重ポインタの違いから、多次元配列を引数に取るのは面倒なので、グローバル化したり構造体に包む方が良い。

メモリの動的確保

- `malloc` 関数や `calloc` 関数を用いて、ヒープ領域を動的に確保できる。
- メモリ確保の際には例外処理を行うべきである。
- 動的に確保したメモリは、使い終わった後必ず解放せねばならない。
- 動的確保領域の伸縮を行いたい場合には、`realloc` 関数を用いる。
- 構造体の最終メンバを不完全配列化し、それを動的に確保できる機能をフレキシブル配列メンバと呼ぶ。

第12講 ストリームと入出力

ここまでの入出力は標準入出力のみであった。だが、実際にはファイルに書きこみたい場合も多い。ここでは、ファイルの扱いを学び、プログラミングの幅を広げよう。

12.1 ファイルとストリーム

ファイルの扱いの前に、まず、ファイルについての基礎知識を身につけておこう。

12.1.1 ファイルとは

ファイル (file) とは何かということに関しては、いまさら問うまでもないかもしれない。例えば、C 言語のプログラムを書き、それをファイルに保存し、さらにコンパイルによって実行ファイルを得る。デジタルカメラを使って写真を撮影する。これらの処理は、いずれもファイルを伴っている。OS の上でもファイルがデータを扱う単位となり、データの扱いを容易にしてくれている。実際、定義も「ハードディスクやフロッピーディスク、CD-ROM などの記憶装置に記録されたデータのまとまり」と、既に経験的に理解しているものと何ら変わらないだろう。

ところで、一口にファイルといっても様々である。いろいろな分類方法があるが、ここでは「テキストエディタを使って人間が読めるもの」と「テキストエディタでは人間が読めないもの」の2つに分けることにしよう。前者をテキストファイル (text file)、後者をバイナリファイル (binary file) と呼ぶ¹。C 言語のソースプログラムはテキストファイルであるが、コンパイルして得られた実行ファイルや写真の画像ファイルはバイナリファイルといえる。以下、もう少し厳密にまとめておこう。

テキストファイルとバイナリファイル

テキストファイル データを「文字」の単位で解釈するファイル。

バイナリファイル データを「バイト」の単位で解釈するファイル。

このまとめからいえば、テキストファイルかバイナリファイルかの別は、それぞれのファイルが持っているものではなく、むしろそのファイルがどう解釈されるかによるという事がわかるだろう。実際、テキストファイルをバイナリファイルとして扱うことも可能である。(逆に、バイナリファイルをテキストファイルとして扱えることは殆どない。)

¹バイナリ (binary) とは「2 進数」という意味である。

C 言語でファイルを扱う上ではテキストファイルとバイナリファイルとで、その処理を変えなければならない。この章では、ひとまずテキストファイルについての扱いを述べた後、バイナリファイルを扱う。

12.1.2 ストリームとは

ストリーム (stream) とは、一言でいえばデータを入出力する対象である。前述したファイルも、入出力対象としてストリームの一種と考えられるが、わざわざストリームを導入したのは理由がある。我々が既に知っている標準入出力に対する処理を、ファイルに対する処理に拡張しようという狙いがあるのである。それだけでなく、プログラミング言語によっては、ストリームが入出力されるデータを蓄えておいたり、インターネットを通してサーバーとの通信機能を担ったりすることもある²。これらはすべてデータを入出力するという共通の機能を持ち、それらを同一のインターフェースでまとめたのがストリームなのである。例えば「標準出力に Hello World を出力する」という処理は、ほぼ同様にして「ファイルに Hello World を出力する」という処理に書き換えることが可能となる。

まとめると、ストリームという概念を用いて、それにファイルとか標準入出力を割り当てることにより、プログラマが面倒をみるべき処理を統一化している、と考えることができる。この後、ファイル入出力のために用いる関数を紹介するが、これらはむしろストリームを操作する関数と考えたほうがよい。

【C 言語における代表的なストリーム】

ファイルを扱う場合、ファイルを開いてストリームとするのであるが、実際には標準入出力などがストリームである場合がある。これらの、C 言語において代表的である (ファイルを開かずに使うことができる) ストリームを紹介しておこう。

C 言語の代表的ストリーム

stdin 標準入力 (standard input) を表しており、入力専用のストリームである。

stdout 標準出力 (standard output) を表し、出力専用のストリームである。

stderr 標準エラー出力 (standard error output) を表し、出力専用のストリームである。

標準エラー出力はコンソール上での出力結果が標準出力と見た目に区別できないため、混同されることもあるが、本来は区別されるべきものである。一般に、これら 3 つを総称して標準入出力 (standard input and output) という。

12.1.3 リダイレクトの意味

リダイレクトについては、既にその方法のみを示した。だが、これがどういう動作かについては詳述しなかった。ストリームを理解したなら、リダイレクトの動作はすぐに理解

²C++ や Java などのオブジェクト指向言語を扱うときに役に立つ。

できるので、ここで説明しておこう。

標準入出力ストリームに対する操作は通常、コンソール (画面) に対する処理と見なされる。よって stdout に文字列を出力した場合、画面にその文字列が出力されることになるし、stdin からの入力にはコンソールを通してキーボードから入力することに対応する。しかし、標準入出力を別の対象と対応付けることもできたほうが便利だろう。この、標準入出力を他の対象に置き換える動作のことをリダイレクト (redirect) と呼んでいるのである。復習も兼ねて、リダイレクトの方法とその置き換え対象を示しておく。

リダイレクトとその置換対象

実行ファイル名 (またはコマンド) の後ろに...

- < (ファイル名) と付す：指定したファイルが stdin に割り当てられる。
- > (ファイル名) と付す：指定したファイルが stdout に割り当てられる。
- 2> (ファイル名) と付す：指定したファイルが stderr に割り当てられる。

12.2 ストリームの取り扱い

ファイル/ストリームについての基礎知識を学んだ所で、いよいよ実際にファイルを取り扱っていくことにしよう。

12.2.1 ファイルの開閉と入出力

ファイル関連の取り扱いは「文法」と言うより「語法」ないし「イディオム」と言う方が正確な部分が多い。それ故、理論を説明せずとも、ソースを追っていくほうが理解しやすいと思われる。ここでもまずはソースを見て、それから説明を加えていこう。

【ファイルの行数】

入力される名前のテキストファイルを開き、そのファイルの行数を数えて出力するプログラムを作成する。

【解説】

ファイルの行数は、動的配列の準備や、統計的処理などで重要になる。このソースを例に、方法を理解しておきたい。

また、このソースはファイル名として入力される 1 行の文字数が 1024 文字以上になる場合、正しく動作しない。

リスト 12.1: ファイル行数カウント

```
1 #include<stdio.h>
2
3 int main(void){
4     char temp[1024],filename[256];
5     FILE *fp;
6     int line;
7
8     fscanf(stdin,"%255s%c",filename);
9     if((fp=fopen(filename,"r"))==NULL){
10         fputs("File_open_error!\n",stderr);
11         return -1;
12     }
13     for(line=0;fgets(temp,sizeof(temp),fp)!=NULL;line++);
14     fclose(fp);
15     fprintf(stdout,"file_%s_has_%d_lines.\n",filename,line);
16     return 0;
17 }
```

【ファイルを開く方法】

ファイルを読み書きする場合、まずは「ファイルを開く」という作業を行う必要があります、そのためには開いたファイル（厳密にはファイルのアドレス）を保持する変数が必要になる。リスト 12.1 では、*l.5* でファイルを開くためのポインタ（ファイルポインタ (file pointer)）を準備している。ここで用いている型 `FILE` は `stdio.h` 内で定義されている構造体で³、これへのポインタを用いてファイルを扱うのが一般的である。

保持するためのポインタを宣言したら、今度はファイルを開き、そのアドレスをポインタに代入してやれば良い。ファイルを開くためには `fopen` 関数を用いる。

ファイルを開く方法

ファイルを開く際には

`fp=fopen(ファイル名を表す文字列, 権限を表す文字列)`

のように行う。ファイル名を表す文字列は `Path` として指定したり、文字列リテラルとしてソースに埋め込んでも問題ない。

翻ってリスト 12.1 を見てみると、*l.9* でファイルを開いている。だが、返却値が `NULL` ポインタである場合には、例外処理を行なっている。これは、`fopen` 関数がファイルを開

³現実のプログラミングにおいては、`FILE` 構造体の中身を知る必要は全くないし、むしろこれが構造体かどうかさえわからなくてよい。`FILE *` という 1 つのデータ型と考え、ブラックボックスとして利用すればよいだろう。

くの失敗した場合に、NULL ポインタを返却するためである。このように、ファイルを開く場合には、その失敗時の例外処理を必ず記しておくべきである。

ファイルを開く際に、第1引数が文字列として与えられる点を利用すれば、一連の(似たような名前の)ファイルに対し、一括処理を行うことも可能である。これには、`sprintf` 関数などを利用すると良いだろう。このように、`fopen` 関数の第1引数が文字列であることは、様々な工夫によってプログラムを便利に記述できるという事でもある。

問題となるのは第2引数の「権限を表す文字列」であろう。ここには、ファイルの開き方(読み込み用か書き込み用か、テキストファイルとして扱うかバイナリファイルとして扱うか)を表す文字列を指定する。ここに指定できる文字列を表 12.1 に示す。

表 12.1: `fopen` 関数に指定できるファイルモード

文字列	意味	備考
<code>r</code>	テキストの読み込み	ファイルが存在しなければ失敗する
<code>w</code>	テキストの書き込み	ファイルが存在すれば既存の内容が削除される
<code>a</code>	テキストの書き込み	ファイルが存在すればファイル末尾から書き込む
<code>r+</code>	テキストの読み書き	ファイルが存在しなければ失敗する
<code>w+</code>	テキストの読み書き	ファイルが存在すれば既存の内容が削除される
<code>a+</code>	テキストの読み書き	ファイルが存在すればファイル末尾から書き込む
<code>rb</code>	バイナリの読み込み	ファイルが存在しなければ失敗する
<code>wb</code>	バイナリの書き込み	ファイルが存在すれば既存の内容が削除される
<code>ab</code>	バイナリの書き込み	ファイルが存在すればファイル末尾から書き込む
<code>rb+</code>	バイナリの読み書き	ファイルが存在しなければ失敗する
<code>wb+</code>	バイナリの読み書き	ファイルが存在すれば既存の内容が削除される
<code>ab+</code>	バイナリの読み書き	ファイルが存在すればファイル末尾から書き込む

なお、表 12.1 は、よく使うので暗記しておくの良い。この表を暗記するのは一見大変なように思えるが、次のようにまとめるとわかりやすいだろう。

`fopen` のモード

- `r`, `w`, `a`^a という文字がファイルの読み書きの別を決める。
- `+` が付けば、読み書き両方が可能となるが、ファイルが存在するか否かによる処理は、もともなった `r`, `w`, `a` の指定によるものを受け継ぐ。
- `b` が付けば、バイナリファイルの操作となる^b。 `b` を付けなければ、テキストファイルの操作となる。

^aそれぞれ `read`(読み込む), `write`(書き込む), `append`(追加する) の頭文字である。

^bUNIX 系 OS ではテキストとバイナリの違いが曖昧であるため、これを付さなくともバイナリファイルを開くことができる。

なお、ファイルのモードは、どのようなファイルを扱うかによって適宜決めればよい。だが、便利だからといって無闇矢鱈に`+`をつけて読み書き可能にするべきではない。というのは、読み込みだけであれば他のプログラムからそのファイルへのアクセスが行われていても処理可能なことがあるためである。これは別のプログラムがそのファイルに書き込みを行っていたとしても、読み込みだけであれば二重に書きこむなどのエラーが生じない

ためである。

ファイルを開いた後、ファイルポインタはファイルの先頭を指している。これに対し、多くの関数では、読み書きした分だけファイルポインタが後ろにずらされ、順次読み書きできるようにしている。もちろん、このファイルポインタを任意に前後させるような関数も存在する(後述)。

【ファイルを閉じる】

開いたファイルは、必ず閉じなければならない⁴。何故閉じないといけないのか、という点になるが、これはファイルを同時に二重に開くと問題が発生するためである。この問題を防ぐため、ファイルに書き込んでいる間には、排他制御という、他からの書き込みを禁止する制御が行われる。この為、ファイルを閉じないと、排他制御がいつまでも終わらない(つまり、他のプログラムがいつまでたってもそのファイルにアクセスできない)ことになる。そのために、ファイルを閉じるという作業が重要なのである。ファイルを閉じるためには、fclose 関数を用いる。

ファイルを閉じる方法

ファイルを閉じる際には

```
fclose(閉じたいファイルポインタ);
```

のように記す。

実際、リスト 12.1 では、l.14 でファイルを閉じている。ここで、終了直前ではなく使い終わった段階で閉じているのは、他ファイルからのアクセスを少しでも早い段階から可能にしたり、ファイルの閉じ忘れを防いだりするためである。

以上に説明したファイル処理の流れは、次のようにまとめられる。

ファイルの処理を行う方法

1. FILE *型の変数を用意する。
2. fopen 関数によってファイルを開く。この時、ファイルが開けなかった場合の処理を例外処理として記述しておく。
3. ファイル処理を行う。
4. fclose 関数によってファイルを閉じる。

ファイルに対する処理の前後には、ファイルの開閉(=ストリーム割り当て)が必ず必要になる。標準入出力以外の入出力先を扱う場合、ファイルに限らず、ストリーム割り当て

⁴動的配列の際にも同様であったが「使ったものはきちんとしまふ」事である。なお、ファイルについては、プログラムが正常終了すれば通常 OS が自動で閉じてくれるため、動的メモリ確保に比べれば問題になりづらい(動的確保領域は正常終了しても解放されない)。だが、その分ファイルを閉じるのを忘れやすいとも言えるので注意しよう。

が必要になる。実際、ネットワークを介して動くプログラムなどでは、ネットワークの接続先への出力をストリームに割り当てて利用する場合もある。

【ファイルへの入出力】

ファイルへの入出力を行う際には、通常の入出力関数に入出力ストリームをつけた関数を用いる。これらの関数は全て `stdio.h` に収められており、ファイルの扱い/入出力関連は `stdio.h` で賄えるようになっている。入出力については、リスト 12.1 でも何種類か見せているが、具体的には次のような関数がある。

ストリームへの入力関数

- `fgets` 関数: 第 1 引数に入力先文字列へのポインタを、第 2 引数に入力最大文字数を、第 3 引数に入力ストリームをとり、データを読み込む。区切り文字は改行で、改行も読み込まれる。
- `fscanf` 関数: `scanf` 関数の引数に先行する引数として入力ストリームをとり、`scanf` と同様の形式で入力を行う。
- `getc` 関数/`fgetc` 関数: 唯一の引数である入力ストリームから一文字読み込み、それを `int` 型として返却する。

ストリームへの出力関数

- `fputs` 関数: `puts` 関数の引数リストの後に出力ストリームをつけた形で記し、文字列を出力する。`puts` と違い、最後の `NULL` 文字を自動で改行に置換しない。
- `fprintf` 関数: `printf` 関数の引数に先行する引数として出力ストリームをとり、`printf` と同様の形式で出力を行う。
- `putc` 関数/`fputc` 関数: 第 1 引数の文字を第 2 引数の出力ストリームに出力する。

この時、注意しなければならないのは入力関数の返却値である。`fgets` の返却値は `char *` 型であり、ファイル終端に来到ると `NULL` ポインタを返す (リスト 12.1 の l.13)。一方で `scanf` 系関数の返却値は `int` 型で、`EOF` というマクロ (`stdio.h` で定義されている) の値 (一般には -1 であることが多い) を返す。

以上までに出てきた入出力関数を、以下、系統別にまとめておこう。まずは入力である。

`scanf` 系関数

- 何れも返却値は `int` 型で、ストリーム終端に達した場合は `EOF` を返す。
- 基本となる `scanf` に対し、何種類かの接頭辞がついて意味を表す。`f` はファイル、`s` は文字列、`w` はワイド文字列、`v` は可変引数リスト等。
- 通常、入力元は第 1 引数として与える。

gets 系関数

- 何れも返却値は `char *` 型で、ストリーム終端に達した場合は `NULL` ポインタを返す。
- `gets` 関数は C11 においてなくなっており、バッファオーバーランを防げないので使用すべきではない。
- 基本的には文字列の入力に用いる。

getchar 系関数

- 何れも一文字読み込んで、それを返却値としている。従って、ストリーム終端かどうかは後述の `feof` 関数などを用いて別途確認が必要。
- 引数は入力ストリームのみで、基本となる `getchar` 関数は引数を取らない。

続いて出力をまとめる。出力は、入力にそれぞれ対応させて理解すると良いだろう。

printf 系関数

- 基本となる `printf` に対し、何種類かの接頭辞がついて意味を表す。`f` はファイル、`s` は文字列、`w` はワイド文字列、`v` は可変引数リスト、`n` は文字列の長さ指定等。
- 通常、出力先は第 1 引数として与える。`snprintf` 系列については、出力先・出力バイト数・書式文字列の順になる。

puts 系関数

- `puts` 関数に限り、終端の `NULL` 文字を改行文字に置換して出力する。
- 基本的には文字列の出力に用いる。

putchar 系関数

- 何れも一文字出力するための関数。
- 引数は出力ストリームのみで、基本となる `putchar` 関数は引数を取らない。

以上の入出力関数とストリームを自在に用いられれば、C プログラミングにおけるテキストの基本的な入出力はマスターしたと言えるだろう。

12.2.2 ファイル・ストリームを取り扱う関数

ファイルやストリームを取り扱う関数は多くあり、標準ライブラリ以外にも多数見られる。以下、標準ライブラリにおいて定義されている関数を紹介する。なお、何れも `stdio.h` に収録されている。

- `int feof(FILE *stream)`: ファイルポインタが終端に達していれば非0を、そうでなければ0を返す。
- `long ftell(FILE *stream)`: 現在のファイルポインタが先頭から何バイトの部分であるかを返却する。
- `int fseek(FILE *stream, long offset, int whence)`: 第1引数のファイルポインタを第3引数の基準の場所から第2引数で示す `offset` バイト進んだ場所に動かす。基準としては、ファイルの先頭を示す `SEEK_SET`、現在の位置を示す `SEEK_CUR`、終端を示す `SEEK_END` などがある。
- `void rewind(FILE *stream)`: 引数のストリームを先頭にセットする。 `fseek` 関数を用いても同等の機能が実現できる他、ファイルを一度閉じて開きなおしても同じ結果になる。
- `FILE *tmpfile(void)`: 一時ファイルを作成してそのファイルストリームを返す。
- `int remove(const char *fn)`: `fn` の示すファイルを削除する。
- `int rename(const char *old, const char *new)`: `old` で示すファイル名を `new` で示すファイル名に変更する。
- `int fflush(FILE *stream)`: 引数のストリームにおいて、何らかの理由で出力されずに残っている文字類を強制的に出力させる。
- `FILE *freopen(const char *fn, const char *mode, stream)`: `mode` に従って `fn` の示すファイルを再オープンして `stream` にストリームを返す。

12.3 バイナリファイルの入出力

ここまではテキストファイルを扱ってきたが、コンピュータで扱うファイルはテキストファイルだけではない。ここでは、バイナリファイルについて理解を深めよう。

【ビット反転プログラム】

入力される名前のバイナリファイルを読み込み^a、その全ビットを反転させることによって暗号化するプログラムを作成する。二度反転すると元に戻るため、これは復号化プログラムも兼ねている。

^aとはいえ、最初に記したとおり、テキストファイルとバイナリファイルは解釈の違いであるので、テキストファイルをバイナリファイルとして読み込ませても何ら問題ない。テキストファイルといえど、その実態はビット列であることに相違ないのだから。

【解説】

- 大きなファイルにも対応できるように作成しているが、ソース中の 2^n の倍数に大きな意味はない。
- 適当なファイルについて (テキストファイルでも良い)、一度これをかけると読めなくなるはずである。二度かけると元に戻る。
- ghex などのバイナリエディタを用いてビットの状態を観察すると、反転していることがわかるだろう。

リスト 12.2: ビット反転プログラム

```
1 #include<stdio.h>
2 #include<stdlib.h>
3
4 #define MIN(x,y) (((x)>(y))?(y):(x))
5 typedef unsigned int u_int;
6
7 int main(void){
8     char filename[256];
9     FILE *fp,*tmpfp;
10    u_int tmp,count,i,*all;
11
12    scanf("%255s%c",filename);
13    if((fp=fopen(filename,"rb"))
14       ==NULL){
15        puts("file_open_error!");
16        return -1;
17    }
18    tmpfp=tmpfile();
19    for(count=0;!feof(fp);count++)
20    {
21        fread(&tmp,sizeof(u_int),1,fp);
22        fwrite(&tmp,sizeof(u_int),1,
23              tmpfp);
24        if(freopen(filename,"w",fp)==
25           NULL){
26            fclose(fp);
27            fclose(tmpfp);
28            return -1;
29        }
30
31        do{
32            tmp=MIN(count,65536);
33            all=(u_int *)calloc(tmp,sizeof
34                                (u_int));
35            fread(all,sizeof(u_int),tmp,
36                  tmpfp);
37            for(i=0;i<tmp;i++) all[i]=~
38                all[i];
39            fwrite(all,sizeof(u_int),tmp,fp);
40            count-=tmp;
41            free(all);
42        }while(count!=0);
43        fclose(fp);
44        fclose(tmpfp);
45        return 0;
46    }
```

以下ではバイナリの入出力について記すが、リスト 12.2 では、随所に「ファイル・ストリームを扱う関数」を用いているので、先の説明と対比して確認されたい。

【バイナリ入出力関数】

バイナリファイルといえど、ファイルである以上、その扱いは基本的にテキストファイルと同じである。唯一違うのは入出力で、バイナリファイルはデータ類を一気に入力/

出力できるという点がテキストファイルと異なる。この、バイナリファイルからの入出力に用いるのが `fread/fwrite` 関数である。

バイナリファイルの入出力

バイナリファイルから入力を行う場合は、`fread` 関数を用い

```
fread(入力先ポインタ, 1つあたりのサイズ, 個数, 入力ストリーム)
```

を実行する。出力の際も `fread` が `fwrite` に代わる以外は同様である。

上記のバイナリ入出力関数によって数値列 (文字列) を配列に読み込んだら、後はこれまでに学んだ方法で処理をすれば良いのである。リスト 12.2 では、その入出力が 2 回ある。これは、第 1 回の入出力である `l.20-21` で 1 個ずつデータを読み込むことによって個数を数え、第 2 回の入出力である `l.32-38` では 65536 個ずつ一括処理したものである。

なお、ここでは大きなファイルでも何とかできるように、数を数えつつ一時ファイルに保存した後に処理を施したが、

```
fseek(fp, 0, SEEK_END);  
filesize=ftell(fp);
```

などとして、ファイルサイズを先に取得してから処理を施す方法もある⁵。興味があれば、これを用いて書き換えてみるのも良いだろう。

12.4 コマンドライン引数

実際のプログラムやコマンドでは、その実行の後ろに引数を取ることができる。これをコマンドライン引数 (command-line argument) と呼ぶ。C 言語においては、コマンドライン引数は `main` 関数の引数として実装され、ほぼその定形が決まっている。

12.4.1 `main` 関数の引数

やはり、最初に例示から入ろう。

【`cat` コマンドの実装】

`cat` コマンドは、その引数として与えられるテキストファイルを順次出力するコマンドである (引数は可変引数で、その個数は幾つでも良い)。このコマンドを C を用いて実装してみる (但し、オプションは一切ないものとする)。

⁵但し、`ftell` の返却値は `long` 型であるので、2GB 以上のファイルは扱えない。そのような場合には、`fgetpos` 関数などの他の関数を使うか、処理系によって定義されている固有の関数類を使わなければならない。これは `ftell` に限らず、いくつかの標準関数に共通した問題である。

【解説】

与えられたファイルを読み込み、それを順に出力しているだけである。なお、可読性向上のため、ファイルの出力は関数化した。

リスト 12.3: 簡易版 cat コマンド

```
1 #include<stdio.h>
2
3 void outfile(FILE *fp);
4
5 int main(int argc,char *argv[]){
6     int i;
7     FILE *fp;
8     for(i=1;i<argc;i++){
9         if((fp=fopen(argv[i],"r"))==NULL){
10             fputs("File_open_error!\n",stderr);
11             return -1;
12         }
13         outfile(fp);
14         fclose(fp);
15         putchar('\n');
16     }
17     return 0;
18 }
19
20 void outfile(FILE *fp){
21     char str[65536];
22     while(fgets(str,sizeof(str),fp)!=NULL)
23         fputs(str,stdout);
24 }
```

リスト 12.3 のような、ファイルに対して操作を行うプログラムは、コマンドライン引数にして作る場合が多い。これを実装しているのが 1.5 の main 関数の引数である。

main 関数の引数

一般的に、main 関数が引数を取る場合 (コマンドライン引数を取る場合)

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

と書きだす。但し、第 2 引数については等価な `char **argv` などと書き換えても良い。名称についても、慣例的に `argc,argv` と決まっている。

この形式によって書かれたソースにおいて、`argc` にはコマンドライン引数として与えられた引数の個数が格納される。引数そのものは、スペースを区切りとしており、`argv[0],argv[1],...` に文字列として格納されている。注意しなければならないのは、実行ファイル名そのものも `argc` や `argv` に含まれる、という事である。例えば


```
./a.out test.txt test2.txt
```

などとしてプログラムを実行した場合、argc は 3 になり、argv[0] は ./a.out に、argv[1] は test.txt に、argv[2] は test2.txt になる。

コマンドライン引数を用いた場合でも、リダイレクトは通常通り行うことができるので、安心して使って良い。

12.4.2 第3のコマンドライン引数

UNIX 系 OS や Windows では、main 関数に第3の引数が存在するものがある。これを利用する場合は通常

```
int main(int argc, char *argv[], char *envp[]){
```

のように main 関数を書きだす。この、第3引数について、少し説明を加えておこう。

この第3引数はセットされている環境変数を取得し、それが各要素に格納される。そして、最後の要素には NULL ポインタが入れられている。従って、NULL ポインタをターミネータとして、環境変数の個数を計算することができるのである。

本講の要点

本講では、ファイル及びストリームの扱いについて学んだ。また、関連する事項として、main 関数の引数についても説明した。

ファイルの扱い

- ファイルにはテキストファイルとバイナリファイルがあり、その解釈の仕方に応じた処理を行う必要がある。
- リダイレクトは標準入力・標準出力・標準エラー出力を置き換える動作である。
- ファイルを扱うための型として FILE 型があり、通常 FILE *型にストリームポインタを開いて用いる。
- ファイルを扱う際には、まず fopen 関数を用いてファイルを開き、その後処理を施して、fclose 関数によって閉じる。
- ファイルに対して入出力を行う場合、ストリームを指定して入出力を行う関数を用いる。
- ファイルを扱う関数のほとんどは stdio.h に収録されている。
- ファイルを取り扱う操作の大半はイディオムであり、関数などを調べて用いる。

main 関数の引数

- コマンドライン引数をとりたい場合、main 関数を

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

の形で書きだす。

- 上記の形で記したソース中において、argc にはコマンドライン引数の個数が、argv[] には各コマンドライン引数が格納される。
- argc, argv は、各々第 0 引数としての「実行ファイル名」を含んでいる。
- main 関数には第 3 引数として char *envp[] を付すこともでき、これは環境変数のリストを表す。
- 環境変数の個数は引数としては現れてこないが、ターミネータとして NULL ポインタがあるので、これを用いて取得できる。
- main 関数の引数に用いる名前は、慣例的にここで書いた名前 (argc, argv, envp) を用いることになっている。

第13講 データ構造の基礎

本講では、アルゴリズムと共にプログラミングを構築する上で非常に重要な要素となるデータ構造について述べる。いくつかのプログラミング言語のチーフデザイナーとして世界的に知られている Niklaus Wirth は、自著のタイトルを”Algorithms+Data Structures=Program”¹とし、データ構造の重要性をタイトルからもわかるようにしたとされる。同書は情報科学の世界の古典的名著とされるが、今なお学べる所が多い本であり、これは同時に、データ構造の重要性が古今変わっていないことを示しているともいえよう。

なお、ここからの3講はC言語の文法ではなく、これまでの文法を用いて実装できる「応用例」を示す。コードが少なくなり、理論説明が多くなるが、これらの実装を自力で行ったり姉妹書の「実習編」を用いたりして、実際に動かしてみたい。

13.1 データ構造とは

多数のデータを扱うとき、これまでに学んできたデータ型の中では配列や構造体を使うだろう。だが、これら単体ではデータ間の適切な関係を表すことができるとは言い難い。たとえば配列だけでは、頻繁に末尾以外への挿入/削除が起こるような処理は時間がかかってしまうため、別の整理方法を考えなくてはならない。このような「多数のデータを組織的に扱うための整理の方法」をデータ構造 (data structure) と呼ぶ。

本講では、基本的なデータ構造のいくつかを扱い、プログラミングでの実装の幅を広げていく。なお、データ構造はここに挙げたものがすべてではなく、まだまだ多くのものがある。それ故、より多くを学びたい人は、他の参考書により学んでいただきたい。

13.2 基本的なデータ構造

まず、スタック・キューという、データ構造の最初に学ぶ「定番」を学ぼう。

13.2.1 スタック

スタック (stack) は棚とも表現されるデータ構造である。この構造は **FILO** (First In Last Out) あるいは **LIFO** (Last In First Out) などと形容され、「最初にこの構造に入ったものが最後に出てくる」ないし「最後にこの構造に入ったものが最初に出てくる」構造である。

¹邦訳は「アルゴリズムとデータ構造」(浦 昭二他訳、近代科学社、1990)

具体的には、スーパーのかごを思い浮かべるとよいだろう。スーパーのかごが山積みになっているとき、必要になればその上側からかごを取る。逆に、必要なくなったらかごを一番上に返す。この「山積み」がスタックである。

「スタック」というと、メモリ領域にも「スタック領域」があった。そして、たとえば再帰処理のメモリの様子を見てみると、確かに「後側に開かれた関数が先に返却値を返す」というスタック構造になっていた。このことからわかるとおり、スタックは再帰処理を用いて代替できる場合もあるし、組み合わせると便利な場合も多い。

では、実際にスタックを用いてみよう。ここでは、スタックを使って回文を判定するプログラムを作ってみよう。

【回文の判定】

スタックを用いて、1000 文字以下の文字列の回文を判定するプログラムを作成する。ここで入力される文字列は英小文字のみからなるものとする。

【解説】

回文の判定そのものはスタックを用いなくとも実装可能である (先に述べた「再帰処理による代替」も可能である)。だが、ここであげる方法を応用すると、文字列解釈の際の括弧の対応なども実装できるので、回文はその基礎として組むとよい。
ここで用いたスタックは、擬似的にその動作を模倣するものであるが、動作概要をつかむには十分だろう。

リスト 13.1: 回文判定 (スタック利用)

```
1 #include<stdio.h>
2 #include<string.h>
3
4 int main(void){
5     char str[1024],stack[512];
6     int i,j,l,flg=0;
7
8     scanf("%1000s%c",str);
9     l=strlen(str);
10    for(i=0;i<l/2;i++){
11        stack[i]=str[i];
12    }
13    for(j=l/2+1%2;j<l;j++){
14        if(str[j]!=stack[l-j]){
15            flg=0;
16            break;
17        }
18        flg=1;
19    }
20    puts(flg?"YES":"NO");
21    return 0;
22 }
```

リスト 13.1 では、1.10-11 でスタックに文字を積み、1.12-13 で LIFO の順に文字を取り出していることがわかるだろう。

ここでは擬似的なものとしたが、実際には、スタックに要素を積むための関数と、スタックから要素を出す関数、記憶領域の 3 つを用意するのが一般的である。

13.2.2 キュー

キュー (queue) とは、待ち行列とも表現されるデータ構造で、スタックに対し **FIFO** (First In First Out) ないし **LILO** (Last In Last Out) と形容される。この形容からわかるとおり「最初にこの構造に入ったものが最初に出てくる」「最後にこの構造に入ったものが最後に

出てくる」構造である。

同じくスーパーでたとえるなら、こちらはレジの行列 (待ち行列) であろう。ある一つのレジでは、先に並んだ人が先に会計を済ませる。この行列こそキューである。

それでは、実際にキューを用いた例を見てみることにしよう。

【キューを用いた Honor の方法】

(1 次以上の) 多項式の次数、各次の係数が入力されるとき、区間 $[-10,10]$ で 0.125 刻みで多項式を計算し、出力するプログラムを作成する。

【解説】

Honor の方法は多項式の計算を高速化する方法である。3 次多項式の場合

$$a_3x^3 + a_2x^2 + a_1x + a_0$$

を順に計算すると掛け算 6 回、足し算 3 回が必要になる。だが、これを

$$((a_3x + a_2)x + a_1)x + a_0$$

として計算すると、掛け算 3 回、足し算 3 回になり、計算回数が減る。多項式を幾度も計算する場合などにはこの違いがばかにならなくなってくるので、普段の計算でもこの方法を使う癖をつけておこう (その場合、キューを使わなくてもよいことも多い)。

なお、先のスタックと同じく、これも動作を理解するための擬似的な実装である。

リスト 13.2: Honor の方法 (キュー利用)

```
1 #include<stdio.h>
2
3 double func(double x,double
4             queue[],int n);
5
6 int main(void){
7     int n,i;
8     double coef[128],x;
9     scanf("%d",&n);
10    for(i=0;i<=n;i++){
11        scanf("%lf",&coef[i]);
12
13    for(i=0;i<=160;i++){
14        x=0.125*i-10;
15        printf("%.3f\t%f\n",x,func(x,coef,n));
16    }
17    return 0;
18 }
19
20 double func(double x,double
21             queue[],int n){
22     double tmp=queue[0]*x+
23             queue[1];
24     int i;
25     for(i=2;i<=n;i++){
26         tmp*=x;
27         tmp+=queue[i];
28     }
29     return tmp;
30 }
```

リスト 13.2 において、1.9-10 の入力 is キューへのインプットである。そして、これから使っているのは関数 `func` であり、計算が前側から行われている点に注目されたい。

キューの実装についても、スタック同様、キューに要素を入れるための関数と、キューから要素を出す関数、記憶領域の 3 つを用意するのが一般的である。

なお、ここでは実装しなかったが、キューの中が常にソートされた状態であり、昇順/降

順に出ていくものをプライオリティーキュー (Priority Queue) と呼ぶ。また、両端から挿入/取り出しが可能なキューもあり、これは両端キュー (double-ended queue) ないしデック (deque) と呼ばれる。興味があれば調べてみていただきたい²。

13.3 リスト

リスト (list) はリンクリスト (linked-list) ととも呼ばれるデータ構造で、データと順序を持つ。配列との違いとして、配列はメモリ上に整然と並んでいるのに対し、リンクリストはポインタを用いて順番を示し、これにより挿入/削除を簡単にしたものである。

13.3.1 片方向リスト

リストは、データ部と次のデータを示すポインタからなる自己参照構造体 (self-referential structure) を用いて実装することが多い。自己参照構造体は、次のように定義できる。

自己参照構造体の定義

自己参照構造体を定義する場合、自身へのポインタをメンバに持たせ

```
struct tag{
    type member;
    :
    struct tag *next;
};
```

のように行う。

ここで説明した自己参照構造体のように、次の要素を示すポインタとデータ部からなる構造体を定義し、これにより「次は何か」を示して並べたリストを片方向リスト (singly-linked list) と呼ぶ。図 13.1 に片方向リストのイメージを示す。

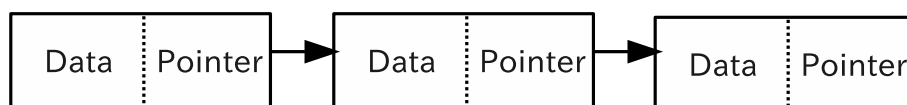


図 13.1: 片方向リスト

自己参照構造体を複数個用意した後、先頭となるものをひとつ定める。そして、順序付けをポインタを用いて行う。最後の要素のポインタを NULL ポインタか先頭要素にしておけば、ターミネータとなる。なお、最後の要素のポインタを NULL にしたものは線形リスト (liniarly-linked list) と、先頭要素へのポインタにしたものは循環リスト (circularly-linked list) と呼ばれている。

²これらのデータ構造の利用については、「プログラミングコンテストチャレンジブック」(秋葉, 岩田, 北川 2010 マイナビ) に詳しい。

【リストの特徴】

リストに要素を挿入したり、リストから要素を削除するときには、ポインタ部を書き換えることで実現できる。そのため、配列のように多数の要素の書き換えを行う必要がない。これはリストの利点の一つである。

その一方で、メモリ上に連続に配置されているわけではなく、要素番号を用いたアクセスなどができない(ある要素を見つけるために先頭からたどっていく必要がある)ため、配列に比べてアクセスにかかる時間が長くなる。

以上のことからわかるとおり、リストは配列に対し、要素の増減に強く、アクセスに弱いという事になる。そのため、よく増減するようなデータを扱う際などに使うと良い。また、リストはこの後に学ぶグラフのうち、単連結有向グラフの一種でもあるため、グラフの基礎としても使うことができるだろう。

13.3.2 双方向リスト

先に学んだ片方向リストでは、次の要素を参照するのは容易であるが、1つ前の要素を参照するのは大変である。そこで、参照のためのポインタを2つに増やし、「次の要素を指すポインタ」と「前の要素を指すポインタ」を準備してやれば、前の要素を参照するのが簡単になるだろう。このように、次の要素と前の要素の両方の参照を持つリストを双方向リスト (doubly-linked list) と呼ぶ。図 13.2 に双方向リストのイメージを示す。



図 13.2: 双方向リスト

双方向リストは片方向リストに比べて要素間の行き来が簡単であり、挿入/削除などの容易さもほとんど変わらない。そのため、リストを使う必要性が出てきた場合には双方向リストを使うほうが楽な場合が多いだろう。

13.3.3 ループのチェック

リストにおいて要素間にループがあるかどうかを調べる必要が出てきた時、どのように行えばよいだろうか。片方向リストを例に考えてみよう。

一つには、通った要素のログをとっておき、それを参照するという方法がある。だが、これは随分余分にメモリ領域を食ってしまう。あるいは、通った要素に対して、「通ったかどうかのフラグ」をつけ、それが既にオンになっているかどうかをチェックする、という方法もあろう。しかし、これだと、循環のチェック後にフラグをリセットするなどの手間がかかるし、何よりリストに用いた構造体を変更しなければならない(あるいは、リストに用いた構造体をネストとして持つ構造体を作らなければならない)。これらの煩わしい手段を何とかして回避できないものか。

よく知られている方法として、2つのポインタを使う方法がある。リストに用いられている構造体を指すポインタを用意し、共に最初は先頭要素を指しておくものとする。それから、片方のポインタは1つずつ、他方のポインタは2つずつ要素をすすめていく。すると、適切に作られた線形リストならば後者のポインタが先に NULL ポインタになるはずである。これが NULL ポインタにならず、途中で2つのポインタが出会ってしまう(同じ値を持つ)ようであれば、リストのループが検出されたことになる。この方法は循環リストに使えないように思えるが、循環リストであれば2つのポインタが再び出会う時は先頭要素であるはずである。そのため、この方法を用いればリストが適切に構築されているかどうか、簡単にチェックすることができるのである。

13.4 木(Tree)構造

木(tree)構造は、「前(親)の要素をひとつだけ持ち、次(子)の要素を複数持つことができる構造」である。各要素のことをノード(node)と呼ぶ。例えば、会社の部署やファイル・ディレクトリの構造などが木構造である。但し、1つの木構造には唯一の「親ノードを持たないノード」が存在し、これを根(root)と呼ぶ。逆に、子を持たないノードは葉(leaf)と呼ばれる。また、親ノードと子ノードを結ぶ線を枝(branch)と呼ぶ。図 13.3 に木の例を示す。

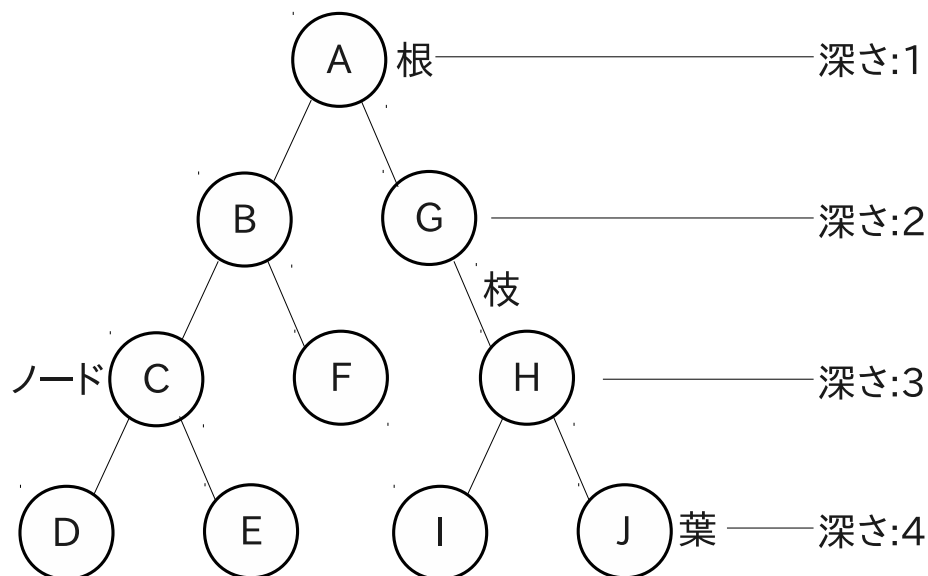


図 13.3: 木(二分木)の例

図 13.3 において、「深さ」と記したのは、根を 1 としてそこから何本の枝でいけるかを足したものであり、そのノードがどの階層に属するかを示している。また、この図を見るとわかるとおり、木はその一部分を取り出しても(例えば、ノード B 及びそれ以下に属するノードのみを取り出しても)木構造になっていることがわかる。この、部分を取り出した木のことを部分木と呼ぶ。木は、このような再帰的性質故、再帰処理と相性が良い。また、木構造を扱う場合、便宜上左側を先に、右側を後にして説明や実装を行う場合が多く、ここでもこの流儀に乗っ取る。

ここでは、まず木構造の基本となる実装を述べた後、その操作について説明し、そこから派生される幾つかの(基本的な)木構造について触れることとする³。

13.4.1 二分木

木の構造の基本を学ぶのに、1つの要素の持つ子要素が最大2個までである二分木(binary tree)を扱う。図13.3の木も二分木である。要素間の兄弟関係などを利用すれば(深さなどは変わるものの)任意の木を二分木に変換できることが知られているため、二分木について学ぶことは大きな意味を持つ。

二分木のうち、各ノードの子ノードがちょうど2個(葉は0個)であるようなものを全二分木(full binary tree)と、更に全ての葉が同じ階層にある全二分木を完全二分木(complete binary tree)と呼ぶ。図13.3の木は全二分木ではない(例えばノードGの子は1つである)が、ノードB以下の部分木は全二分木である。

二分木は、片方向リストと似たような形で、データ部に子ノードへのポインタ2つを加え

```
struct tree_node{
    :
    struct tree_node *left;
    struct tree_node *right;
};
```

のような構造体を用意すれば実装できる⁴。もしも親ノードへの参照が必要な場合は、双方向リストと同様、親ノードへのポインタを追加しても良い。

13.4.2 木の探索

木の全てのノードを調べたい場合がしばしば存在する(走査(traversal))。このとき、もれなく重複なく調べる方法として、深さ優先探索と幅優先探索が知られている。

【深さ優先探索 (DFS)】

深さ優先探索 (Depth first search, **DFS**) は、根からスタートし、葉に行き着くまで順に枝を辿り、そこまで行き着いたら戻って別の葉を探索し…という方法である。図13.3において深さ優先探索をした場合、その経路はアルファベット順になる(左側優先の場合)。

DFSは、「左側の部分木を見る」「右側の部分木を見る」「根を見る」の3つを再帰的に行うことで実装可能である。この時、部分木を見るより前に根を見るDFSを行きがけ順

³木構造には、ここで紹介する木構造以外にも、BIT(Binary Indexed Tree)・赤黒木・フィボナッチヒープ・セグメント木・スタープロット木・スプレー木・B+木等、挙げていくと枚挙に暇がないほどの種類がある。

⁴この構造体と同様の形式の構造体を双方向リストの実装の際にも用いた。だが、表しているものは全く違う。このように、データ構造とはあくまでもデータの整理の方法であるので、同じ型のものを用いても様々な実装を行うことができる。また、配列を用意し、その番号をポインタ代わりに使うなど、様々な実装を考えることもできるだろう。

(preorder traversal) と呼ぶ。その他の順番に対しても名前がついており、左側を見た後右側を見る前に根を見るものを**通りがけ順** (inorder traversal) と、左右を見た後に根を見るものを**帰りがけ順** (postorder traversal) と呼ぶ。

【幅優先探索 (BFS)】

DFS が左右の木をある種非対称に見ていったのに対し、**幅優先探索** (Breadth first search, **BFS**) は対称的に見ていく方法である。BFS ではキューなどを用いて、同じ階層にあるノードを見ていく。図 13.3 の場合、A,B,G,C,F,H,D,E,I,J のように見ていくことになる。

キューを用いるとはどういうことか、図 13.3 を例にもう少し説明を加えておこう。まず根である A を見る。そしてこの時、「探索すべきもの」として、二つの子ノード B,G をキューに入れる。次いで、キューの先頭にある B の要素を見る。そして、やはり同様に、子ノード C,F をキューに入れる。次に G を見て…と同様のものを繰り返せば良い。

以上に説明した探索方法は、後で説明するグラフにも利用されるので、以下にまとめておこう。

DFS と BFS

DFS 根から始まり、そこから行くことができるノードが存在する限り経路をたどる。行くことができなくなったら戻って、戻ってきたノードの別の分岐をたどっていく。一般に再帰を用いて実装される。

BFS 根から始まり、そこから直接行くことができるノードをキューに入れる。そして、キューの先頭ノードに対し、同様にして、直接いけるノードをキューに入れる。これを繰り返して探索を行う。

13.4.3 二分ヒープ

二分木に対して、次の二つの制約を課したものを**二分ヒープ** (binary heap) あるいは、単に**ヒープ**と呼ぶ⁵。

(二分) ヒープの制約

- 親ノードはそのいずれの子ノードに対しても、一定の大小関係 (等号含む) を持つ。すなわち、「親ノード \geq 子ノード」か「親ノード \leq 子ノード」のいずれか一方が、全ての親子関係について成立する^a。 (**heap property**)
- 構築された木は完全二分木か、最下階層の一段を完全二分木に加えた形になる。この時、最下階層は左側から埋まる。 (**shape property**)

^aもちろん、木の各要素は数値とは限らないので、大小関係はデータに応じてプログラマが定めなければならない。これは後に扱うソートでも同じ事である

⁵一般の木に対して同様の制約を課したのもヒープ (heap) と呼ばれる。

図 13.4 にヒープの例を示す。ここでは、大小関係として、アルファベット順に先にあるものを小さいとした。

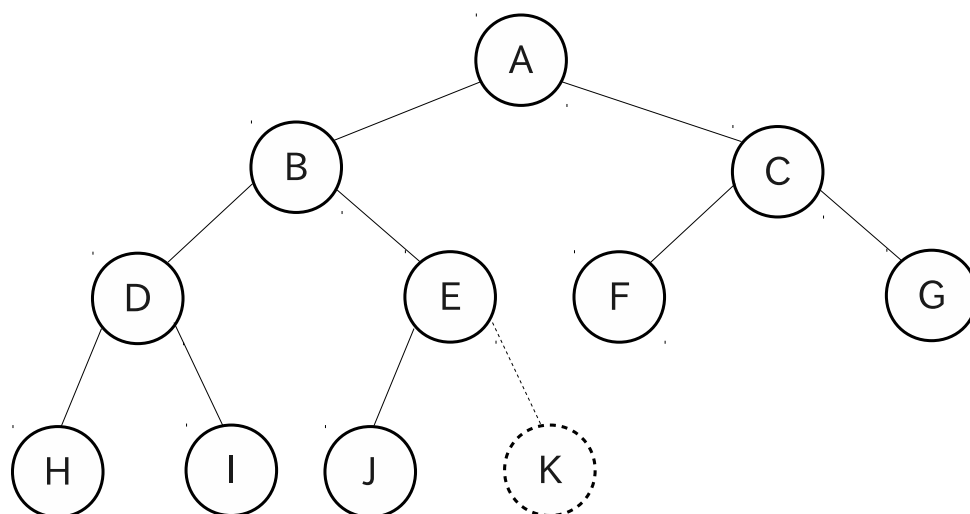


図 13.4: ヒープの例 (大小関係:アルファベット順に先に来るものが小さい)

なお、ヒープは親と子の大小関係については制約を設けているが、子同士の関係には言及していない点に注意しよう。すなわち、左側の子と右側の子の大小関係はどうでもいいのである。これは、二分ヒープに限らず、一般のヒープに言えることである。

図 13.4 のヒープは、確かに二つの制約を満たしている (そして同時に、偶然にも、左側の子ノードは右側の子ノードよりも小さい)。この関係性により、ヒープはソートや最小値/最大値を求める際によく用いられる。以下、この構築方法について記す。

【上方移動によるヒープへの要素挿入】

ヒープに新たな要素を加えることを考えよう。この時、図 13.4 のノード K のように、今埋まっていない階層の一番左側に新たなノードを追加する。これにより、shape property が保たれる。だが、単純に追加しただけでは heap property が保たれない。

そこで、親ノードと比較し、順序が正しくない場合に親ノードと交換を行う上方移動 (shift up) を行う。もしも、親ノードとの交換が起きた場合、再帰的にこれを実行し、交換が起きなくなるまで交換を繰り返す。これにより、追加ノードはヒープの適切な位置に挿入される。

何もない状態から、逐次上方移動によって要素を追加/挿入していけば、ヒープを構築することができる。

【下方移動によるヒープへの変換】

先程、最初のノードから順に上方移動で順次要素を追加していけばヒープを構築できると記した。だが、これでは追加の度に再帰を行わなければならない、煩わしい。

そこで下方移動 (shift down) という操作を行い、全データの割り当てられた二分木をヒープへ変換する方法が知られている。

下方移動によるヒープへの変換

前提として、全データを shape property を満たすように二分木に割り当てておく。

1. ヒープを後方 (階層が深い側、同階層では右側) から順に見ていき「葉でないノード」を見つける。このノードを以下親ノードとする。
2. 見つかったノードに対し、子ノードと値を比較し、それらの最小値が親ノードの値となるように (必要に応じて) 交換を行う。
3. 再び、最初のステップに戻り、探索を続ける。これを親ノード=根となるまで続ける。

ここで紹介した上方移動/下方移動は、各々DFS/BFS と似た様相を呈している。木への操作の基本はDFS/BFSにあるため、再帰やキューといった道具を適切に使えるようにしておくが良い。

13.4.4 二分探索木

ヒープがソートに便利な木であったのに対し、二分探索木 (Binary search tree) はその名の通りサーチによく用いられる他、様々に応用がきく構造である。これは、二分木に対し、次の制約を課した木である。

二分探索木の制約

任意の親ノードに対して

- その左側の部分木の全要素が、注目している親ノードの値と比べ小さいか等しい。
- その右側の部分木の全要素が、注目している親ノードの値と比べ大きい。

という条件が成立すること。なお、ここでは等しいノードを左側に入れるとしたが、右側に入れても構わない (但し、木全体で左側に入れるか右側に入れるか統一されていなければならない)。

上記の制約のため、所定の値を検索する際に素早く検索できるのが二分探索木の特徴である。また、これに対して通りがけ順DFSで値を表示させた場合、昇順ソートされて値が出力される (もちろん、右側を先に持ってくれば降順ソートも可能である)。

【二分探索木の構築】

二分探索木に要素を追加するにはどうすればよいだろうか。これは、DFS 同様に上側から順に見ていけば良い。ここで、注意しなければならないのは、ヒープの追加は下から上に木をたどっていったが、二分探索木では上から下にたどる、という点である。以下、具体的に手順を示そう。

二分探索木への要素の追加

1. 最初に根ノードを「現在のノード」として以下の操作を行う。
2. 追加したい値が現在のノードよりも小さいか等しいならば左側へ、大きいならば右側へ進む。
3. 進んだ先にノードが存在しなければそこに追加したい値を新たなノードとして追加して終了する。ノードが存在する場合、そのノードを現在のノードとして、前項に戻る。

これにより、最初に適当な根ノードを決め、これに対して順次要素を追加していけば二分探索木が構築できる⁶。同様にして、二分探索木を用いてのサーチを行うこともできる。

【ノードの削除】

ここまではノードの追加/木の構築について話してきたが、必要なくなった要素は木から削除しなければならない。ヒープの場合、削除したい場所に最後のノードを移動し、大小関係について整合性を取れば良い(上方/下方移動)のであるが、二分探索木からの削除は少し面倒である。

二分探索木からのノードの削除

1. 最初に根ノードを「現在のノード」として以下の操作を行う。
2. 現在のノードと、削除を行う値を比較する。この時、削除する値が現在のノードと等しければステップ4に進む。等しくない場合、削除する値が現在のノードよりも小さければ左側に、そうでなければ右側に移動し、次項へ進む。
3. 進んだ先のノードを現在のノードとして、前項へ戻る。
4. 現在のノードが子ノードを持たない場合はそのノードを削除して終了する。子ノードを持つ場合、削除した後に次項に進む。
5. 子ノードを1つだけ持つ場合は削除した場所を、子ノードによって置き換えて終了する。2つ持つ場合は削除したノードの右側部分木の最小値のノードで置き換える。この時、置き換え元(最小値)のノードに右側子ノードが存在するならば、右側子ノードを置き換え元ノードに置き換える(以下の関係は保つ)。

13.5 グラフ

リンクリストは「前のノードが1つ・次のノードが1つ」というデータ構造であった。木は「前のノードが1つ・次のノードが複数」というデータ構造であった。この自然な拡

⁶なお、この方法で構築していく時、例えば昇順/降順データが与えられたら、二分探索木は線形リストのような形になってしまう。これを防ぐために用いられるのが赤黒木などの平衡二分探索木である。本書では紙数の都合上扱わないが、興味があれば学んでみて欲しい。

張として、「前のノードが複数・次のノードが複数」というデータ構造を考えることができる。あるいは、前後の区別をなくし「隣り合ったノードが複数」という事もできる。路線図やロードマップ・地図のようなものがこのようなデータ構造をなしている。前後の区別をするならば、例えばロードマップの場合、一方通行の道路があったらそこには指向性がある。前後の区別をしないものとしては、例えば地図が考えられる。地図を見てみると兵庫県は京都府・大阪府・和歌山県・鳥取県・岡山県・徳島県・香川県と接しているが、これは接していることだけに意味があり、兵庫県から京都府へ方向付けする必要性はない。ここでは、「複数のノードが隣り合っているデータ構造」であるグラフを扱う。

13.5.1 グラフの基礎概念

グラフ (graph) とは、ノード (node, 路線図でいう所の駅に当たるもの。日本語では節点ないし頂点と呼ぶ。) とそれらをつなぐエッジ (edge, 日本語では辺。路線図でいう所の駅の間の線路に当たるもの。) からなるデータ関係をいう。図 13.5 にグラフの例を示す。

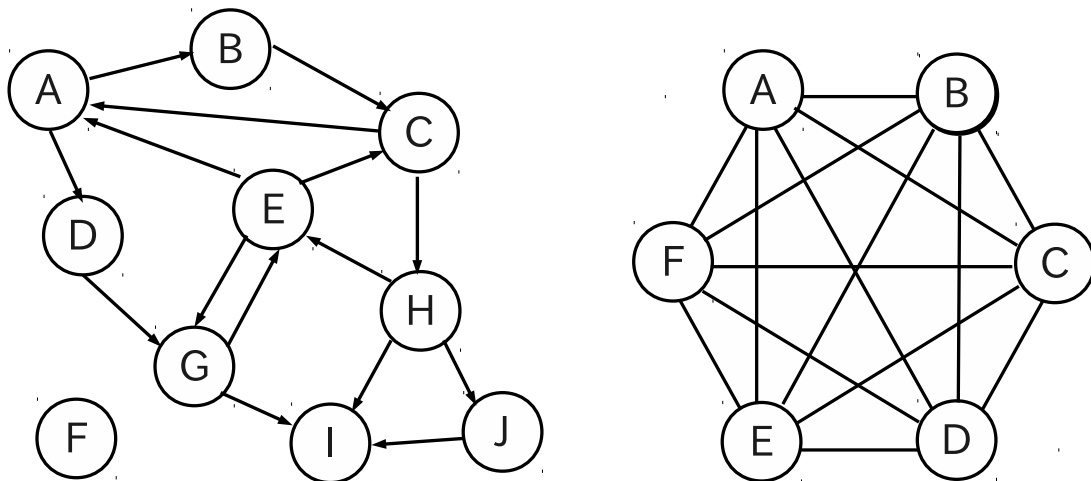


図 13.5: グラフの例 (左:有向グラフ・右: 無向グラフ)

図 13.5 の左側のように、辺に指向性があるものを有向グラフ (digraph) と呼ぶ。これに対し、図 13.5 の右側のような、辺に指向性がないものは無向グラフ (undirected graph) と呼ばれる。

グラフの各々の辺には方向づけを行うことができるが、それに加え重みを付けることができる。例えば路線図では、A-B 間は 3 分、A-D 間は 2 分、B-D 間は 4 分などと所要時間を付けることができるが、この所要時間を辺毎にのせていけば、これが辺の重みになる。ここで挙げた路線図の場合は向きに寄って時間が変わったり一方通行だったりすることはないだろうが、ロードマップの場合は混雑の度合いによって所要時間が $A \rightarrow B$ は 30 分、 $B \rightarrow A$ は 25 分などのように指向性を持つ場合があり、このような場合には有向グラフを用いることになる。ここでは重みを所要時間とした為、重みは正または 0 だろうが、実際には負の値を取る重みも存在する⁷。

⁷負の値を取る重みなど、理論はともかく実際には何の役に立つのか? と思うかもしれない。だが、辺の途中でものの授受などがある場合は正負両方が必要だろう。例えば実生活でも、どこかに行く経路をたどる

【グラフの種類】

グラフには、その特徴に応じて幾つかの種類がある。ここではそれを紹介しておく。

無向グラフにおいて、任意の2ノード間に経路が存在するグラフのことを**連結グラフ** (connected graph) と呼ぶ。グラフを扱う場合、連結性を確かめてから使う場合が多い (チェックについては次項に述べる)。また、連結グラフでなくとも、適切な部分グラフに分ければ、連結グラフに分けて扱える。

図13.5右のように、全てのノードの間にエッジがあるようなグラフを**完全グラフ** (complete graph) と呼ぶ。自己ループや多重辺 (ある2ノード間に複数の同方向エッジが存在するもの) を含まない**単純グラフ** (simple graph) を扱うアルゴリズムを考える時、完全グラフはエッジの数が最大化されたグラフとして扱えるため、「エッジの多い場合の極端な例」としてしばしば用いられる。ノード数 V の完全グラフに対して、そのエッジ数 E は

$$E = \frac{V(V-1)}{2} \quad (13.1)$$

により計算することができる。

また、閉路 (ループ) を持たないグラフにも特別な名前が付いている。閉路を持たない連結グラフは、書いてみればわかるが、前述の木である。木をグラフとしてみた時、完全グラフとは逆にエッジ数が一番少ないため、「エッジの少ない場合の極端な例」として使うことができる。ノード数 V の木に対してそのエッジ数 E は

$$E = V - 1 \quad (13.2)$$

である。

一方、閉路を持たない非連結グラフは、木が多く集まった形をなすことから**森** (forest) と呼ぶ。

以下では、無向単純グラフを基本に説明を行うこととし、単に「グラフ」と書いた場合は無向単純グラフを指すこととする。有向グラフの場合にも多少工夫することで同様のアルゴリズムを適用できる場合が多い。

13.5.2 グラフの実装と連結性チェック

ここでは、グラフを実装し、その連結性をチェックすることについて説明する。これは、グラフに関する諸問題の基礎となる部分である。

【隣接行列/接続行列による表現】

グラフを実装する場合、リストや木で用いたような構造体表現も可能であるが、重みやノード毎に個数が違う等の面倒な部分もある (有向グラフの場合は指向性の表現も面倒)。

時、途中で銀行のある道でお金をおろすならば、その道ではお金がプラスになる。だが、お金を払わなければいけない道 (有料道路など) があればお金はマイナスになるだろう。

エッジ数が少ない場合は構造体表現も楽であるが、増えてきた場合 (特に完全グラフの場合) は構造体表現は面倒である。そこで、グラフのノードを配列を用いて用意しておき、エッジに対し隣接行列 (Adjacency matrix) と呼ばれる行列を用いることが多い。

隣接行列は二次元配列を用いて実装される。隣接行列の $[i][j]$ 成分をノード i からノード j への辺の重みとする。辺が存在しない場合は、それを表す値を代入しておく。重みがないグラフについては、 $1/0$ を辺の有無に対応させて用いる場合が多い。

また、別の実装として、ある点からある辺が出ているかどうかについて接続行列 (incidence matrix) を用いることもある。これは $[i][j]$ 成分をノード i からエッジ j が出ているかどうかに対応させた行列である。単純グラフを扱う場合、この行列では1列にちょうど2個の1 (ノードからエッジが出ている値) が見つかるはずである。

【グラフの連結性チェック】

グラフが連結であるかどうかは、アルゴリズムが適用できるかどうかを始めとして、様々な場面で重要である。ここではグラフの連結性を調べる方法について述べる。

グラフが連結であるということは、先に述べたとおり「任意の2ノード間に経路が存在する」ということである。ここで、無向グラフにおいてはある1ノードから全てのノードに到達可能である場合、他のノードからも全てのノードに到達可能であるという点に着目する。すると、ある1点から全ての点にエッジをたどって到達可能であるかどうかを調べれば良い、という結論に至る。

では、到達可能かどうかはどのように判定すればよいか。これは、実際にある1ノードから「もれなく・重複なく」たどってみれば良い。DFSないしBFSの出番である。各ノードが到達可能かどうかを表す一連のフラグ配列を用意し、これに可能であるかどうかを調べていけば良いのである。DFSを再帰処理で組むのは変わらないが、BFSの場合、キューを用いずとも、フラグ配列を順に見て行って、まだ見ていないノードを見れば良い。

同様に、連結性のチェックに限らず、グラフを探索する場合にはDFSかBFSのいずれかを用いるのが基本となる。

この後に話す最短路問題・閉路問題では連結グラフを前提とし、特段の記述がない場合非連結グラフは扱わない。

13.5.3 最短路問題

最短路問題 (Shortest path problem) とは、重み付きグラフにおいて、あるノードから別ノードへの道のうち重みを最小化するものを求める問題である。この内、ある2点間の最短路を求める問題は**2頂点对最短路問題**と、単一の始点から全ての点への最短路を求める問題は**単一起点最短路問題**と、全ての点対について最短路を求める問題を**全点对最短路問題**と呼ぶ。ここでは、図13.6のグラフを例に、単一起点最短路問題と全点对最短路問題のアルゴリズムを紹介する⁸。

⁸2 頂点对最短路問題については、単一起点最短路問題のアルゴリズムを用いて求めれば良い

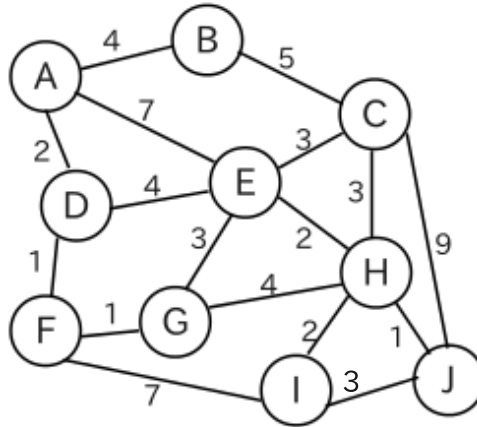


図 13.6: 重み付き無向グラフ

なお、以下では最短路のコストを求めているが、最短路そのものを求める (経路復元) 場合も同様のアルゴリズムで直前の頂点を記憶しておけば良い。もちろん、コストを求めてから BFS/DFS などをして求められる。

【Bellman-Ford 法】

図 13.6 の始点 A からの単一起点最短路問題を考えよう。ノード i, j を結ぶエッジの重みを $C_{i,j}$ で表し⁹、A からノード k への最短距離 (最小重み) を D_k として、次のような等式が成立する。

$$D_i = \min_j (D_j + C_{i,j}) \quad (13.3)$$

木の場合、式 (13.3) を初期条件 $D_A = 0$ の下で順次計算していけば全ての点への最短路が求められる。しかし、グラフに閉路が含まれる場合などは「順次」の順序を決めることができない。そこで、 $D_A = 0, D_{i(\neq A)} = \infty$ として、式 (13.3) を適用し、値を更新していく。この手法を **Bellman-Ford 法** と呼ぶ。手順をまとめると次のようになる。

Bellman-Ford のアルゴリズム

1. 初期化 : スタートノードの値 (最小コスト候補) を 0、他のノードの値を ∞ (充分大きな値) に設定する。
2. 各エッジに対して式 (13.3) を適用して、その隣点からの最短距離更新がないかをチェックし、現在の「仮の最短距離」よりも小さければ更新する。
3. 前ステップを、更新がなくなるまで繰り返す。この反復は高々頂点数-1 回で終わる。

ノード数 V 、エッジ数 E のグラフに対して、この時間計算量は $O(VE)$ である。なお、時間計算量については次講か姉妹書の実習編を参照されたい。

⁹無向グラフの場合、 $C_{i,j} = C_{j,i}$ であるが、有向グラフでは区別する。また、ノード i, j 間にエッジが存在しない場合、 $C_{i,j}$ は充分大きな値とする (実装では大きな値だが、理論上は $C_{i,j} = \infty$ としても良い)。

負の重みを持つエッジ (有向グラフの場合は負の閉路) が存在した場合、最短経路は存在しない (その閉路を繰り返し通ることで無限に小さくできるため)。しかし、このケースでは、Bellman-Ford 法が頂点数-1 回の反復で終了しないため、それにより負閉路を検出することが可能である。これは、次に述べる Dijkstra 法にはない利点である。

【Dijkstra 法】

Bellman-Ford 法では、一度エッジを計算しても、その点への最短距離が定まるかどうかはわからなかった。そのため、更新を何度も繰り返した。だが、図 13.6 のノード D への最短経路は、ノード A から出るエッジの重みなどを考慮すれば、 $D_D = 2$ であると直ちにわかる。同様の論法で、 $D_F = 3, D_B = 4, D_G = 4$ と、順次最短経路を確定させていくことができる。そして、確定したノードへのエッジについては、もうこれ以上考慮に入れる必要はなくなる。これにより、Bellman-Ford 法から無駄を省いた方法が次の **Dijkstra 法**¹⁰ である。

Dijkstra のアルゴリズム

1. 初期化：スタートノードの値を 0、他のノードの値を未定義（または ∞ ）に設定する。
2. 確定ノードをピックアップできなくなるまで（＝変化がなくなるまで）次の 3,4 のループを繰り返す。但し、最初の確定ノードはスタートノードとする。
3. 確定ノードから伸びているエッジをそれぞれチェックし、式 (13.3) により現在の最小コストを計算し、そのノードの現在値よりも小さければ更新する。
4. まだ確定されていないノードのうち、最小の値を持つノードを見つけ、確定ノードとする（確定フラグを立てる）。

Dijkstra 法を図 13.6 に適用した場合、確定ノードは A,D,F,(B,G),E,H,(C,J),I の順に定まる。但し、括弧でくくったノードの順番は入れ替わることがある。

Dijkstra 法は「現在の最小値はこれ以上更新されない」という考えに従うため、更新が起こりうる負のエッジを持つグラフ (有向・無向を問わない) については適用できない。また、ノード数 V 、エッジ数 E のグラフに対して、Dijkstra 法により最短経路を求めるための時間計算量は $O(V^2)$ である。ただし、これは確定ノードを決定する際に、ループを回して全てのノードをチェックするからであり、ここをヒープ (またはプライオリティキュー) を用いて管理するとより速くなる。以下、これについて述べる。

スタートノードを根としたヒープを考えよう。ここでは簡単のため、図 13.4 の各ノードの文字と、図 13.6 の各ノードの文字が対応していることにし、ヒープの各ノードには D_i の値を割り当てておくことにする。

ノード A に隣接しているノードを考えれば、 D_D, D_B, D_E の更新が起こる。この更新を

¹⁰ この Dijkstra (ダイクストラ) は構造化プログラミングの提唱者として知られているエドガー・ダイクストラその人である。

行った後、ヒープのノード A を削除する (削除方法は既に述べた)。そして根ノードとなったノードを確定ノードとし、同様の処理を施していけば良い。まとめると、次のようになる。

Dijkstra のアルゴリズム (ヒープ利用)

1. 初期化：スタートノードの値を 0、他のノードの値を未定義（または ∞ ）に設定する。
2. ヒープ初期化：最短距離の値についてのヒープを構築する。この際、ヒープのデータは、元のグラフのどのノードに対応しているかという事と、最短距離の値である。
3. ヒープのノードが尽きる (または 1 つになる) まで、次の 4~6 の操作を繰り返す。
4. ヒープの根ノードに対応するグラフのノードを確定ノードとする。
5. 確定ノードから伸びているエッジをそれぞれチェックし、式 (13.3) により現在の最小コストを計算し、そのノードの現在値よりも小さければ更新する。
6. ヒープから根ノードを削除し、更新した値を用いてヒープを再構築する。

この方法を用いると確定ノードを速く決められる事になるため、時間計算量が $O(E \log V)$ となる。これは、Bellman-Ford 法に比べて速い方法である。そのため、最短経路を定めることが可能な無向グラフや、負エッジを有しない有向グラフについては Dijkstra 法が用いられることが多い。

なお、全てのエッジのコストが等しい場合には、単に BFS を用いてエッジ数を記録していったほうが簡単に経路を求められる。

【Warshall-Floyd 法】

全点对最短経路問題を解く場合には、各ノードに対する単一始点最短経路問題を Bellman-Ford 法や Dijkstra 法により解いてもよいが、一般化した **Warshall-Floyd 法** を用いることが多い。この方法は、先の 2 つのアルゴリズムと違った漸化式を用いる (漸化式を作って計算する、動的計画法 (Dynamic Programming, DP) の一種である)。

頂点に 0-indexed により番号をつける。第 k 番までの頂点及びノード i, j そのものを用いて構築可能な、ノード i, j 間の最短経路を $D_{i,j,k}$ と表すと、次の関係式が成立する。

$$D_{i,j,k+1} = \min(D_{i,j,k}, D_{i,k+1,k} + D_{k+1,j,k}) \quad (13.4)$$

従って、この漸化式に対し、適切な初期値を定めて計算を実行すれば良い。初期値は

$$D_{i,j,0} = \min(C_{i,j}, C_{i,0} + C_{0,j}) \quad (13.5)$$

とすれば良い (但し、 $C_{i,j}$ はノード i, j を結ぶエッジの重みで、存在しなければ ∞)。これをコードにまとめると次のようになる。なお、実際の実装の際には、 $D_{i,j,k}$ のうち、 k を落としてしまっても良い。

Warshall-Floyd のアルゴリズムの実装

i, j, k は int 型とし、式 (13.4) に示したのと同じ意味を持つものとする。 V はグラフの頂点数で、 $D[i][j]$ は $D_{i,j,k}$ を表すものとする。また、 $C[i][j]$ は隣接行列である。

リスト 13.3: Warshall-Floyd のアルゴリズム

```
1 #define MIN(x,y) (((x)<(y))?(x):(y))
2
3 // initializing
4 for(i=0;i<V;i++) for(j=0;j<V;j++) D[i][j]=MIN(C[i][j],C[i][0]+C[0][j]);
5
6 //calculation
7 for(k=1;k<V;k++) for(i=0;i<V;i++) for(j=0;j<V;j++)
8     D[i][j]=MIN(D[i][j],D[i][k]+D[k][j]);
```

ここに示したコード断片により、 $D[i][j]$ にノード i, j 間の最短距離が入る。

ノード数 V 、エッジ数 E のグラフに対して、Warshall-Floyd 法は $O(V^3)$ で動作する。Bellman-Ford 法を全ノードに用いた場合 $O(V^2E)$ 、Dijkstra 法の場合 $O(VE \log V)$ であることを考えると、とりわけ、エッジ数が多いグラフに対して速いことがわかる。また、比較の実装が簡単なのも利点である。

さらに、Warshall-Floyd 法は、負エッジを持つ有向グラフにも適用できる他、負閉路 (無向グラフの場合負エッジ) の検出も可能である ($D[i][i]$ が負になる)。

13.5.4 閉路問題

グラフ理論の古典的な問題として閉路問題を紹介する。

【頂点の次数】

閉路問題の前に、ノードの次数という概念を導入する。各ノードに対し、それに接続しているエッジの数を次数と呼んでいる。有向グラフの場合、ノードに入ってくる辺の数を入次数と、ノードから出ていく辺の数を出次数と呼ぶ。

【オイラー閉路問題と一筆書き】

閉路問題は大きくわけて2つにわけられる。まず、全ての辺を1度ずつ通るような閉路であるオイラー閉路 (Eulerian cycle) について説明する。

与えられたグラフがオイラー閉路であるかどうか (このようなグラフをオイラーグラフ (Eulerian graph) と呼ぶ) は、次の定理により判定することができる。

オイラーグラフ

与えられたグラフがオイラーグラフであることは、全ノードの次数が偶数であることと同値である。

オイラー閉路問題を言い換えると、スタート地点とゴール地点が同じ一筆書きでそのグラフを書くことができるかどうか¹¹の判定である。一方、現在よく考えられる一筆書きには、例えば数字の9のように、最初の点と最後の点が等しくないような場合もある。このような「一筆書き問題」の判定も考えてみよう。

一般に、一筆書き可能なグラフのことをオイラー路 (Eulerian path) と呼ぶ。オイラー閉路は当然オイラー路でもある。オイラー路でありオイラー閉路でないようなグラフ (準オイラーグラフ (semi-Eulerian graph)) は、次のように判定できる。

準オイラーグラフ

与えられたグラフが準オイラーグラフであることは、全ノードのうちに次数が奇数であるものがちょうど2つ存在する事と同値である。

以上から、一筆書き可能かどうかは、グラフにおいて全ノードのうち、次数が奇数であるものが存在しないか2つ存在するかによって判定可能である。なお、オイラーグラフの始点 (= 終点) はどこでもよいが、準オイラーグラフの始点と終点は、次数が奇数である2つのノードである。

【ハミルトン閉路問題と巡回セールスマン問題】

オイラー閉路が全辺を通る問題だったのに対し、ハミルトン閉路 (Hamiltonian cycle) は全ての頂点を1度ずつ通るような閉路の事を言う。閉路でないが全ての頂点を1度ずつ通るような路をハミルトン路 (Hamiltonian path) と呼ぶのは、オイラー路と同様である。

与えられたグラフが、ハミルトン閉路を含む場合、そのグラフをハミルトングラフ (Hamiltonian graph) と、ハミルトン路を含むがハミルトン閉路を含まないグラフを準ハミルトングラフ (semi-Hamiltonian graph) と呼ぶ。オイラーグラフはグラフそのものであったが、ハミルトングラフは含むかどうかである点に注意されたい。

あるグラフがハミルトングラフであるかどうか判定するハミルトン閉路問題を効率よく解くアルゴリズムはない。ハミルトン路を含むかどうかの判定は、それ以上に難しいとされる。さらに、ハミルトン閉路問題に重みをつけた、「ハミルトン閉路のうち最短路を求めよ」という問題は巡回セールスマン問題 (Traveling Salesman Problem, **TSP**) として知られる難問である。ハミルトン閉路問題は Class-NP (後述) に属するが、ハミルトン路問題や巡回セールスマン問題は Class-NP にすら属さない (丸つけすら難しい) 可能性のある問題であり、オイラー閉路同様に古典的であるが、難易度は雲泥の差である。

ここまで、グラフについて古典的な問題を扱ったが、グラフの問題には、彩色問題・マッチング・フロー問題・最小全域木など、まだまだ多くの問題が存在する。詳しくはグラフ理論の専門書¹²を参照されたい。

¹¹ この問題の最も古典的な例としてケーニヒスベルグの橋が知られている。

¹² 例えば、「シリーズ 情報科学の数学 グラフ理論」(恵羅, 土屋 1997 産業図書) などがある。

本講の要点

本講では、データを適切に整理して扱うために、基本的なデータ構造を学んだ。

基本的なデータ構造

- スタックは LIFO、キューは LILO のデータ構造である。

リンクリスト

- リンクリストは、ポインタを用いてデータの順序関係を示し、データの挿入や削除を容易にしたものである。
- リンクリストの各要素は最大 1 つまでの前要素/次要素を持つ。
- リンクリストを実装する場合には自己参照構造体を用いる。
- リンクリストに循環が存在するかどうかは、2 つのポインタを「おいかけっこ」させることにより判定できる。

木

- 木構造は前要素を 1 つ以下、次要素を複数持つデータ構造である。
- 任意の木は、前要素 1 つ以下、次要素 2 つ以下の二分木に変換できる。
- 木の探索 (走査) 方法は DFS と BFS が知られている。
- ノード間あるいは木全体に制約をかけることで、ヒープや二分探索木といった様々な木を実装できる。

グラフ

- グラフは前要素・次要素ともに複数持つデータ構造である。前後の区別をなくしたものもある。
- グラフが連結しているかどうかは様々な問題の基礎になる。連結の判定は BFS や DFS による。
- 重み付きエッジを持つグラフに対し、そのノード間の最小重みを求める問題を最短路問題と呼んでいる。最短路問題には Bellman-Ford 法・Dijkstra 法・Warshall-Floyd 法などの方法が知られている。
- 閉路問題は大きくオイラー閉路とハミルトン閉路に分かれる。前者は次数を数えることで判定可能であるが、後者は簡単には解けない。

第14講 ソートとサーチ

ソートは、ある程度慣れたプログラマであれば常識として知っておいても良いアルゴリズムで、活用される場面も少なくない。また、アルゴリズムの解析においても、典型的かつ動作が見やすくイメージしやすいため、非常によく用いられる例である。本講では様々なソートを知り、アルゴリズムの解析手法についても学ぶと共に、これと関連の深い配列の探索についても学ぶ。ソート、サーチの世界はそれだけで分厚い本が書けるほどに奥深く¹とても1講で解説しきれるものではないが、ここで素地だけでも学んでおくことが後々役立つと考えている。

14.1 ソート総論

個々のソートを見ていく前に、ソート全体について説明する。

14.1.1 ソートとは何か

ソート (Sort) は、日本語では「整列」や「並び替え」と訳され、データをある一定の規則に従って並び替えることを言う。ソートの方法は多数存在し、それぞれに特徴がある。

この章のソートの部分では、代表的なソートのアルゴリズムのみを紹介し、紙数の関係上実装は行わない。そのため、これまでの復習として自力で実装していただきたい。なお、特別に断らない限り、ソート対象を要素数 n の `int` 型配列、ソート規則を昇順 (小さい数が前) として説明を行う。

14.1.2 ソートの解析～バブルソートを例に～

アルゴリズムの解析について、本書ではここまで全く書いてこなかったので、ここでバブルソートを例に解析法を紹介しておく。なお、演習編では最初にアルゴリズムの解析について掲載したので、合わせて参照されたい。

バブルソート (Bubble Sort) は、配列を前から順に見ていって、その大小関係がひっくり返っている場所をスワップすることで整列を行う方法である。例えば `[4,1,3,2,3]` のバブルソートを考えると、`[1,4,3,2,3]→[1,3,4,2,3]→[1,3,2,4,3]→[1,3,2,3,4]→[1,2,3,3,4]` というソート手順になる。小さい要素が泡のように上 (前) にやってくることから、この名前がついた。

¹具体的には「The Art of Computer Programming Volume 3」(D.E.Knuth 著、有澤 誠他訳、ASCII、2006)。ソート・サーチについてはこの本1冊で万全であると言って過言でない名著。

【時間計算量と空間計算量】

さて、バブルソートの特徴として、配列を一度スキャンし終わると、最大の要素が配列の末尾に来ることが挙げられる。そのため、今度は配列より1小さい数の比較ですむ。これを繰り返していくと、比較回数は $\frac{n(n+1)}{2}$ 回であり、交換回数は最大 $\frac{n(n+1)}{2}$ 回であることがわかるだろう。このことから、バブルソートのアルゴリズムの計算回数は、 n の二次関数に比例して大きくなっていくことがわかる。つまり、バブルソートにかかる時間は、十分 n が大きい時、「高々」 n^2 に比例する程度である。これを $O(n^2)$ のアルゴリズムと記す。

ここで用いた O による記法をランダウの漸近記法 (Landau asymptotic notation) とよぶ。なお、ここでは「高々」の評価としたが、同じく n が十分大きい時ちょうど $f(n)$ に比例するようなら $\Theta(f(n))$ と、少なくとも $g(n)$ に比例するようなら $\Omega(g(n))$ と記す。以下では原則 O を用いる。

また、これらの O, Θ, Ω は何れも使われるメモリの量を記すときにも使える。バブルソートの場合、元の配列以外で余分に使うメモリはせいぜい交換用の変数ぐらいで、 n によらないので $O(1)$ のメモリ量である。

ここに述べた、計算回数のオーダーは一般に時間計算量 (time complexity) と、メモリ量のオーダーは空間計算量 (space complexity) と呼ばれている。また、 $O(1)$ は定数オーダーと、 $O(\ln n)$ は対数オーダーと、 $O(n^k)$ は多項式オーダー²と、 $O(e^n)$ は指数オーダーと呼ばれる。時間計算量にこれらのオーダーが用いられた場合「 $\bigcirc\bigcirc$ オーダーの時間」と呼ぶのは煩わしいので「指数時間」などと呼ばれる。

オーダーについて注意しなければならない点として、十分大きい値での比例関係での等号なので定数倍やよりオーダーの小さい項があったとしてもこれらは同じオーダーである、という点が挙げられる。例えば、 $O(n^3)$ と $O(5n^3)$ と $O(3n^3 + 2n^2 + \log n)$ は同じオーダーである。

ここで紹介した時間計算量や空間計算量は決して理論だけのものではない。実装方法を決める際に、求められている仕ように対して時間計算量/空間計算量を見積もることで、適切な実装を選ぶ一助とすることができる。主要な処理のオーダーを知っておけば、全体の見通しを立てやすい (例えば、`strlen` 関数は、引数の文字列長 l に対し、各回の呼び出し～返却まで $O(l)$ がかかることが知られている)。また、計算回数を大雑把でよいので見積もれば、処理にかかる時間等も把握しやすい³。

²ある問題に対し、その解を多項式時間で求められる問題を **Class-P** (P:Polynomial time) の問題と呼ぶ。また、ある問題の解の候補が与えられた時、それが解として正答かどうかを判定するのに多項式時間で判定できるものを **Class-NP** (NP:Non-deterministic Polynomial time) の問題と呼ぶ。わかりやすく言うと、多項式時間で解ける問題が Class-P、多項式時間で丸付けのできる問題が Class-NP である。アメリカ・クレイ研究所のミレニアム懸賞問題の中には、これにまつわる問題として **P 対 NP 問題** がある。この問題は、Class-NP に属する問題は必ず Class-P に属するか? という問題である (逆に Class-P の問題が Class-NP に属することは直感からも明らかだろう)。興味があれば調べてみて欲しい。

³現在の PC なら、大体 1000 万回～1 億回の処理で 1 秒である (1 回の処理にかかる時間や PC 性能により変動)。もちろん、計算時間を小さい n について測定し、そこから大きい n についての時間を推定するという手法も有効である。

【ソートの安定性】

バブルソートでは、最初に2つあった3の順序関係が入れ替わっていない。このように、同じ大きさの値が配列にある時、その順序関係が入れ替わらないソートを安定なソート (Stable sort) と呼び、逆に入れ替わるソートを不安定なソート (Unstable sort) と呼ぶ。単なる値の整列だけならば気にしなくてよいが、構造体や文字列をソートする場合には安定性が問題になってくる場合がある。ソートによっては、不安定であっても比較的簡単に安定に組みなおすことが可能なものもある。

【バブルソートのまとめ】

以上に説明したバブルソートについてまとめておこう。なお、バブルソートは速度も遅く、これといった優位性も無いので、実装練習以外に使われることは稀である。

バブルソート

【手順】

1. 配列を順にスキャンし、途中、大小関係が逆転している部分があれば、その2要素を交換する。
2. 第 k 回目のスキャンにおいて第 $n - k + 1$ 要素までスキャンを終えたら^a、1に戻る。但し、ソートが完了しているか $k = n$ であれば終了する。

【性質/特徴】

安定性 安定

時間計算量 $O(n^2)$

空間計算量 $O(1)$

^a効率化を考えて第 $n - k + 1$ 要素までとしたが、第 n 要素まで見るとする方が一般的である。もちろん、第 n 要素まで見るほうが遅い。

以下に紹介するソートでは、簡単な例示や特徴的な性質の説明などを主眼とし、時間計算量などはその結果のみを示すに留める。

14.2 基本的なソート

まず、あまり高速ではないがオーバーヘッドが小さく、比較的簡単に組めるソートを紹介する。小規模データや、後で扱う高速なソートと組み合わせて用いられることが多い。

14.2.1 挿入ソート

挿入ソート (insertion sort) ないし単純挿入法は配列を順に見ていき、途中で大小関係の逆転している場所を見つけたら、後側の要素をそこまでの適切な場所⁴に「挿入」する

⁴この探索には、後で説明する二分探索を用いると良い。二分探索を用いた挿入ソートを特に二分挿入ソート (binary insertion sort) と呼ぶこともある。

方法である。ほぼソートされた配列に強く、ソート済み配列への要素追加などの際に高速である。

挿入ソート

【手順】

1. 配列を順にスキャンしていく。
2. 途中、大小関係が逆転している部分があれば、その後ろ側の要素を、既にスキャンを終えた部分のうち適切な位置に挿入する。この時、配列をひとつずつ順繰りにずらさなければならない点に注意。
3. 配列の終端まで来たら終了する。

【性質/特徴】

安定性 安定

空間計算量 $O(1)$

時間計算量 平均 $O(n^2)$ /最良 $O(n)$

付記 ほぼソート済み配列に対し高速

この手順に従って、 $[4,1,3,2,3]$ をソートすると、 $[1,4,3,2,3] \rightarrow [1,3,4,2,3] \rightarrow [1,2,3,4,3] \rightarrow [1,2,3,3,4]$ となり、バブルソートよりも手順が少ないことがわかるだろう。

14.2.2 選択ソート

選択ソート (selection sort) は「王様探し」などとも呼ばれるソートで、配列中から最小値を探し、それを先頭に持っていく方法である。挿入ソートと異なるのは、最小値を先頭に挿入して残りの要素を順繰りにずらすのではなく、元々先頭にあった要素と交換する点である。交換回数が最大 $n-1$ 回である分、バブルソートより高速である。一般に選択ソートといった場合は不安定であるが、安定に実装するのも難しくない。

選択ソート

【手順】

1. 配列を順にスキャンするが、第 k 回目のスキャンでは第 k 要素からスキャンをはじめめる。 $k = n$ ならば終了する。
2. 最後までスキャンすることで最小要素を見つける。
3. 最小要素と第 k 要素を交換し、最初のステップに戻る。

【性質/特徴】

安定性 不安定 (安定も可)

時間計算量 $O(n^2)$

空間計算量 $O(1)$

この手順に従って、 $[4,1,3,2,3]$ をソートすると、 $[1,4,3,2,3] \rightarrow [1,2,3,4,3] \rightarrow [1,2,3,3,4]$ となる (但し、交換が行われないスキャンを省いた)。

14.2.3 シェーカーソート

シェーカーソート (shaker sort) ないしカクテルソート (cocktail sort) は、バブルソートに近いソートで「第 k 回目のスキャンを終えると後側 k 項が整列されている」という性質を使って改良したものである。まず、配列を前から見ていき、一番後ろまでバブルソートの要領で交換する。その後、後ろまでたどり着いたら、今度は反対向き (後ろから前) に向かって配列を見ていき、同様の要領で交換を行う。第 k 回目の往復では、第 k 要素から第 $n - k + 1$ 要素の間を往復することになる。

このソートも挿入ソート同様にほとんどソートされた配列に対して高速である。但し、実装の長さや平均速度の面から、挿入ソートのほうが一般的である。

シェーカーソート

【手順】

1. このステップを行うのが第 k 回目であるとき、第 k 要素から配列をスキャンし、逆順箇所を交換する。
2. 第 $n - k + 1$ 要素にたどり着いたら、今度は配列を逆に見て、逆順箇所を交換する。
3. 第 k 要素まで戻ってきたら、ステップ 1 に戻る。これを、往復区間がなくなる^a まで交換が起こらなくなるまで繰り返す。

【性質/特徴】

安定性 安定

空間計算量 $O(1)$

時間計算量 $O(n^2)$

付記 ほぼソート済み配列に対し高速

^a毎回 2 ずつ往復区間が減り、ソートは 1 要素以下で終わりであるため、 $k = \lceil \frac{n}{2} \rceil$ 回目の往復が終わったら、ソート完了とできるという事がわかる。但し、 $\lceil \cdot \rceil$ はガウス記号。

この手順に従って、 $[4,1,3,2,5,3]$ をソートすると、 $[1,4,3,2,5,3] \rightarrow [1,3,4,2,5,3] \rightarrow \dots \rightarrow [1,3,2,4,3,5] \rightarrow [1,3,2,3,4,5] \rightarrow [1,2,3,3,4,5]$ となる。

14.2.4 ノームソート

ノームソート (gnome sort)⁵ は単純に実装できる割に、最良だと $O(n)$ の安定ソートである。基本的にはバブルソート同様、順番にスキャンして逆順の部分交換する。但し、交換が起きた時、1 つ前に戻ってもう一度比較する、という手順が入る。

⁵gnome は庭師を示し、庭師が鉢植えを並べ替える方法に由来する。

ノームソート

【手順】

1. 配列を順にスキャンする。
2. 順序が逆の場所を見つけたら交換を行い、続きのスキャンを1つ前の要素から行うようにする。最後まで辿り着けば終了。

【性質/特徴】

安定性 安定

空間計算量 $O(1)$

時間計算量 平均 $O(n^2)$ /最良 $O(n)$

付記 一重ループで実装可能

この手順に従って、 $[4,1,3,2,3]$ をソートすると、 $[1,4,3,2,3] \rightarrow [1,3,4,2,3] \rightarrow [1,3,2,4,3] \rightarrow [1,2,3,4,3] \rightarrow [1,2,3,3,4]$ となる。

14.2.5 シェルソート

シェルソート (Shell Sort)⁶は挿入ソートを改良したものである。「大雑把に見ていってだんだん細かくする」という考えと「ソート済み配列に対して挿入ソートは高速である」という考えが元となっている。まず、大雑把に間隔を開けて挿入ソートを行う。その後、この間隔を少しずつ小さくしていくという手法である。

シェルソート

【手順】

1. 適当な間隔 h を定める。通常、これは2の累乗数にすることが多い。
2. 間隔 h をあけてとった数列に対して、挿入ソートを適用する。なお、このような間隔 h の数列は配列中に複数存在する為、挿入ソートも複数回行われる。
3. 間隔 h を適度に狭めて (2の累乗数にしている場合は半分にする事が多い) 先と同様の操作を行う。
4. $h = 1$ になるまでこれを繰り返してソートすることができる。

【性質/特徴】

安定性 不安定

時間計算量 $O(n \log^2 n)$

空間計算量 $O(1)$

なお、このソートで間隔 h を $h_{n+1} = 3h_n + 1$ という漸化式から生み出される数列 $\{1, 4, 13, 40, \dots\}$ にすると、時間計算量が $O(n^{1.25})$ になることが知られている。

⁶シェル (Shell) は開発者の名前 Donald L. Shell からとっており、貝殻とは関係ない。

シェルソートはここまでで紹介したソートに比べ、やや速く、中規模データの並び替えに便利である(ここまでのソートは小規模向け、この後のものは大規模向けが基本である)。中規模向けであることから、後述の「高速なソート」に、挿入ソートなどと共に組み合わせられることが多い。

数列 $[8, 3, 1, 2, 6, 5, 6, 4]$ を間隔 $4, 2, 1$ の順に変化させてシェルソートしてみよう。まず、間隔 4 なので、 $[8, 3, 1, 2, 6, 5, 6, 4] \rightarrow [6, 3, 1, 2, 8, 5, 6, 4]$ となる。ここから、間隔 2 と間隔 1 を順に適用して $[1, 2, 6, 3, 6, 4, 8, 5] \rightarrow [1, 2, 3, 4, 5, 6, 6, 8]$ となる。

14.3 高速なソート

比較ソート(個々の項目を比較演算で大小判定することを基本とするソート)の限界速度は $O(n \log n)$ であることが理論的に知られている。比較ソートは汎用性が高くメモリもそれほど多く食うわけではないため、 $O(n \log n)$ のソートは実用的である。これよりも速いソートはデータの特異性や特別なハードウェアなどが必要な場合があるが、もしも適用できるならば有効である。ここでは、 $O(n \log n)$ の比較ソートを中心に、それ以外のソートとしてバケットソートを紹介する。

14.3.1 マージとマージソート

マージソートはD&Cの一種で、配列の各部分をソートし、それらに対してマージと呼ばれる操作を行うことによって全体をソートしようとするものである。

【マージ】

マージ(merge)とは、2つのソート済み配列を統合し、1つのソート済み配列にする操作である。2つの配列をキューに見立て、先頭要素同士を比較して、小さい側を統合後の配列にいれればよいので、次のように非常に簡単に実装できる。

【マージの実装】

ポインタ $p1, p2$ によって示される2つの int 型配列をマージし^a、 $p3$ に示される配列に格納する関数を作成する。 $p1, p2$ の配列は各々 $n1, n2$ 個の要素を持つものとし、 $p3$ の要素数は $n1+n2$ 以上あるものとする。

^aここでは int 型昇順という縛りをつけて組んだが、より一般化することもできる。実際、`stdlib.h` に収録されているソートの関数(後述)は、汎用ポインタ・要素あたりのサイズ・個数などを用いて一般化されている。規則についての一般化は簡単には思いつかないかもしれない。だが、順序付けを関数によって行うものとし(一般に比較関数(comparison function)と呼ばれる)、その関数のポインタを引数に取れば、汎用化できる(前述のソートの関数も引数に比較関数をとっている)。

リスト 14.1: マージ

```

1 void merge(int *p1,int *p2,int *p3,int n1,int n2){
2   int i,p1_suf=0,p2_suf=0,tmp=n1+n2;
3   for(i=0;i<tmp;i++){
4     if(p1_suf>=n1) *(p3+i)=*(p2+p2_suf++);
5     else if(p2_suf>=n2) *(p3+i)=*(p1+p1_suf++);
6     else
7       *(p3+i)=(*(p1+p1_suf)<*(p2+p2_suf))**(p1+p1_suf++):*(p2+p2_suf++);
8   }
9 }

```

要素数 n_1, n_2 の 2 つの配列をマージする場合、それにかかる時間は $O(n_1 + n_2)$ である。

【マージソート】

2 つのソート済み配列をマージすれば、それによってできる統合後の配列はソート済みである。従って、ある配列を前半と後半の 2 つに分けて、その各々をソートした後にマージする、という戦略を考えることができる。つまり、配列を分割し、統治 (ソート) しているのである。これを再帰的に適用していくのがマージソート (merge sort) である。

ここで、何故分割統治するのかという事を考えてみよう。1 段だけの分割を考える。 $2n$ 要素の配列を挿入ソートで整列する場合、挿入ソートは $O(n^2)$ であるため、その比較回数は定数 C を用いて $4Cn^2$ である⁷。だが、前半と後半に分け、各々 n 要素を挿入ソートするとすれば、ソートにかかる比較回数は $2Cn^2$ ですむ。マージは $O(n)$ であるので、 n が十分大きいとき、分割してソートした後にマージする方法は、単純にソートする方法のおよそ半分の計算回数になり⁸、高速化できるのである。これは、マージソートのみにとどまらず、D&C を用いるソート全体に共通していることである。

このように、分割は計算回数を減らすことができ、しかも再帰的に適用可能である。マージソートの場合、各部分列に対して再帰的にマージソートを適用し、要素数 1 の配列ができたならそれを「ソート済み配列」として戻ってマージすれば良い。この時、部分列の数は再帰 1 段につき 2 倍になり、これらをマージして戻すのに n 回程度の比較がある。このことから考えてわかるとおり、 $O(n)$ のマージを $\log_2 n$ 回程度行う必要があるので、マージソートは $O(n \log n)$ で動作する。この過程において配列要素の値が関係していないことからわかるとおり、マージソートはどのような配列に対しても $O(n \log n)$ で動作してくれ

⁷ここでは均等分割しているが、奇数の場合は前半ないし後半の要素数を 1 だけ大きくすれば良い。できるだけ均等にすると理由としては $n_1 + n_2 = N$ ($n_1, n_2, N \in \mathbb{N}$) に対して $n_1^2 + n_2^2$ を最小化する問題を考えてみれば良い。

⁸ n が小さい場合、「ソートする配列が半分になって比較の減った分」に対する「マージに必要なになった比較の増えた分」の割合が大きくなるので、高速化が望めるかどうか怪しくなる。このような場合の対策については後述する。同じ事で、3 分割以上の分割をした場合も、マージに必要な比較が増えてしまうため、小さい n に対して高速化が望めるかどうかは怪しい。

る。但し、マージのためのサブ配列が $O(n)$ 程度必要である。以上をまとめると次のようになる。

マージソート

【手順】

1. 配列の要素数が1であればそれをソート済みとして終了する。
2. 配列を前半と後半に (ほぼ) 均等に分割する。
3. 配列の前半・後半に各々マージソートを再帰的に適用する。
4. 配列の前半・後半をマージし、ソート済み配列を作って終了する。

【性質/特徴】

安定性 安定

時間計算量 $O(n \log n)$

空間計算量 $O(n)^a$

^a $O(1)$ で動作する In-place マージソートというアルゴリズムもある。

【少要素数への対応】

先の脚注でも説明したが、マージソートを始めとする D&C 型ソートは、少要素数の場合に遅くなりがちである (ないし、高速化される見込みがなくなる)。仮に 4 要素の配列に対してマージソートを適用してみると、 $n \log n = 8$ となってしまう、定数によってはマージソートのほうが $O(n^2)$ のソートよりも遅いことも十分あり得る。しかし、分割統治でアルゴリズムを組んでいる限り、このような少要素数のソートが部分問題として出てきてしまう。そこで、これに対する対応を考えてみる。

マージソートを再帰的に適用するのは、要するに部分列をソートするためであった。そこで、部分列が十分小さい (高速化が見込めない) 時には、それを別のアルゴリズムによってソートすれば良い。例えば、要素数が 8 以下ならば (ソート済み or ほぼソート済みの可能性が高いため) 挿入ソートを適用するなどの方法がある (不安定でも良いなら、もう少し多い段階でシェルソートを用いても良い)。

従って、先にまとめたマージソートの第 1 ステップを「要素数が一定数以下であれば挿入ソートを適用して終了する」に変更すれば、少要素数の場合の問題が解消され、高速化が見込めるだろう。なお、この「小規模の時挿入ソート、中規模の時シェルソートで部分問題を解く」というテクニックは、マージソートにとどまらないテクニックである⁹ので、他の D&C ソートの実装の際にも使ってみて欲しい。

⁹同じ事で、D&C アルゴリズムに対しては、それを再帰適用して部分問題が十分小さくした後、部分問題を別の方法で解いて統合していくことが多い。この理由も本節で述べたのと同じものである。このことから、マージソートは D&C アルゴリズムの縮図と言ってよいだろう。

14.3.2 クイックソート

クイックソートもマージソート同様 D&C の一種であり、最悪は $O(n^2)$ だが、平均的には $O(n \log n)$ の高速なソートで、比較ソートの中ではトップレベルの平均速度である。

【クイックソートのアルゴリズムと乱択】

クイックソートは、まず適当な値 (ピボット (pivot)) を定め、これより小さい値を前側に、大きい値を後側に集める。その後、ピボットの部分を切れ目にして分割し、これを再帰的に適用する (分割区間は半分ずつとは限らない)。ピボットは通常、配列中の要素から定められる。

クイックソートにおいて重要なのはピボットの選択である。仮に、各部分列において最大値/最小値ばかりをピボットに選んでしまった場合、これは最悪のケースとなり $O(n^2)$ の計算回数が必要になる。そこで、乱数を用い、選ぶピボットの規則をなくしてやることにより、最悪のケースを回避しやすくする方法が知られている (乱択クイックソート)¹⁰。他にも最悪のケースを回避する方法は幾つもあるが、実装の際には、ひとまず乱択クイックソートを作ることができればよいだろう。以下、乱択クイックソートについて記す。

(乱択) クイックソート

【手順】

1. 要素数が 1 であるならば確定したものとして終了する。
2. 乱数によってピボットを 1 つ選ぶ。
3. 左から順に見て、ピボット以上のものを見つけたらその位置を i とする。同様に、右からも見て、ピボット以下のものを見つけたらその位置を j とする。
4. i が j より左ならば、その 2 つの要素を入れ替えて、探索に戻る (i は右に、 j は左に一つ進める)。そうでない場合は次に進む。
5. i の左側を境界に分割を行って 2 つの領域に分け、そのそれぞれに対して本アルゴリズムを再帰的に適用する。

【性質/特徴】

安定性 不安定

空間計算量 $O(n)$ ($O(\log n)$ にもできる)

時間計算量 平均 $O(n \log n)$ / 最悪 $O(n^2)$

なお、ピボットを選ぶ方法としては、乱択で 3 つにとってその中央値を取るなどの方法も考えられる。そのため、先の手順の 2 は必要に応じて変更すれば良い。

¹⁰この方法も決して万能ではない。1 が 10000 個、10000 が 10000 個、2~9999 の値が 500 個というような配列のソートでは思ったより速度がでないこともありうる。理想的には、各部分列で中央値を求めてピボットに用いれば良いのであるが、中央値を求めるには $O(n)$ にかかるため、高速化のために中央値を用いたはずがそれほどの効果が見込めない (あるいは逆に遅くなる) こともありうるのが難点である。

クイックソートも D&C であるので、マージソート同様、少要素部分列に対しては挿入ソートを適用するように組み替えて高速化をはかることができる。

【qsort 関数と比較関数】

C 言語にはクイックソートに由来した¹¹、ソートを行う関数 `qsort` が `stdlib.h` に用意されている。これを使ったソートの例を見ておこう。

【qsort 関数によるソート】

`qsort` 関数を用いて `int` 型配列を昇順整列する。

リスト 14.2: `qsort` 関数の例

```
1 #include<stdio.h>
2 #include<stdlib.h>
3
4 #define SIZE_AR(x) ((sizeof(x))/(sizeof(x[0])))
5
6 int compare(const void *p1, const void *p2);
7
8 int main(void){
9     int data[5]={1,9,2,0,-4},i;
10    qsort(data,SIZE_AR(data),sizeof(int),compare);
11    for(i=0;i<SIZE_AR(data);i++)
12        printf("%d\n", data[i]);
13    return 0;
14 }
15
16 int compare(const void *p1, const void *p2) {
17     int n1,n2;
18     n1=*((const int *)p1);
19     n2=*((const int *)p2);
20     return n1-n2;
21 }
```

リスト 14.2 の l.10 において、`qsort` 関数を用いている。この関数は第 1 引数にソートしたい配列へのポインタを、第 2 引数に整列したい要素の数を、第 3 引数に 1 要素あたりの要素のサイズをとってソートする。第 4 引数の比較関数 (comparison function) への関数ポインタは、ソート順序の規則を表す。

`qsort` 関数 (及び `bsearch` 関数) の比較関数には、次のような条件が要請される。

¹¹ 名前の由来がクイックソートであるだけで、実際に中のアルゴリズムがクイックソートである必要はない。極端な話、バブルソートやストージソート、ボゴソートなどの実用に向かないレベルのソートであっても仕様は満たしていることになる。とはいえ、主要なコンパイラは十分に速いソートで実装しているので心配ない。

比較関数の条件

- 引数が2つで、共に `const void *` 型であること。
- 返却値が `int` 型であること。
- 第1引数と第2引数を比較し、第1引数が第2引数より前に来るべき時は負値を、第1引数が第2引数より後にくるべき時は正值を、2つを等しいとみなす場合には0を返すようにする^a。

^a`strcmp` 関数が同じような形の返却値を取るので、参考にとすると良い。但し、`strcmp` 関数そのものは引数の型の要件を満たさない。

上記の条件を満たす比較関数を適切に定義することで、`qsort` 関数を用いて任意のデータを整列できる (構造体を含む)。

14.3.3 ヒープソート

親ノードが子ノードより小さいという関係性で構築されたヒープにおいては、ルートノードは最小値を示している。そして、ヒープの再構築にかかる時間計算量は $O(\log n)$ である。これを利用して、ヒープを用いて選択ソートを行う¹²ソートのことをヒープソート (heap sort) と呼ぶ。不安定であり、平均的にはクイックソートよりも遅いが、空間計算量は $O(1)$ であり、最悪時間計算量 $O(n \log n)$ なので、使い勝手のいいソートだと言える。

ヒープソート

【手順】

1. 配列内の全要素を用いてヒープを構築する。
2. 構築されたヒープのルートノードの値を配列に格納する。
3. ルートノードを削除し、要素が残っていなければ終了する。残っている場合、ヒープを再構築し、前項に戻る。

【性質/特徴】

安定性 不安定

時間計算量 $O(n \log n)$

空間計算量 $O(1)$

残りの要素が少ない場合、挿入ソートを使って高速化することもある。

14.3.4 コムソート

コムソート (comb sort) は挿入ソートをシェルソートに変えたのと同様の方法で、バブルソートを「大雑把に見てから細かく」適用したものである。理論上 $O(n^2)$ だが、実際には $O(n \log n)$ で動く。速度の割に実装が簡単でコードも短いので、比較的使いやすいソートである。但し、不安定である点には注意されたい。

¹²二分探索木や平衡二分探索木を用いる方法もある。

シェルソート同様に、ある一定の間隔を空け、その間隔毎にバブルソートを行う。第 k 回目の間隔 h_k は漸化式

$$h_k = \left\lceil \frac{h_{k-1}}{1.3} \right\rceil, \quad h_0 = n \quad (14.1)$$

によって計算される (但し、 $\lceil \cdot \rceil$ は Gauss 記号)。以上、まとめて次のようになる。

コムソート

【手順】

1. $k = 1$ とし、式 (14.1) により h_k を計算する。
2. 配列の第 i 項と第 $i + h_k$ 項 (但し、 $i = 0, 1, \dots, n - h_k$) を比較し、前者のほうが大きい場合は入れ替える。
3. $h_k \neq 1$ であるならば k を 1 増やして、 h_k を計算し、前項に戻る。 $h_k = 1$ ならば、正しくソートされるまでバブルソートを繰り返して終了する。

【性質/特徴】

安定性 不安定

時間計算量 $O(n \log n)$

空間計算量 $O(1)$

14.3.5 ストランドソート

ストランドソート (strand sort) はマージを利用するソートであるが、マージソートとは違い D&C には属さない。性能はマージソートに劣るが、実装は楽である。

このソートは昇順部分配列の抽出とマージを繰り返すだけである。手順を示そう。

ストランドソート

【手順】

1. 元の配列の他に、サブ配列と結果配列を用意しておき、次の 2 つの操作を元の配列の要素がなくなるまで繰り返す。
2. 配列を前から見ていき、昇順関係が崩れないようにサブ配列に要素を移す。(抽出)
3. サブ配列を結果配列にマージし、前項に戻る。(マージ)

【性質/特徴】

安定性 安定

空間計算量 $O(n)$

時間計算量 平均 $O(n \log n)$ /最悪 $O(n^2)$

手順を聞くだけではわかりづらいので、配列 $[3, 7, 1, 9, 5, 7, 4, 6, 8, 2]$ をストランドソートした場合の変化を表 14.1 にまとめた。

表 14.1: ストランドソートの変化 (操作の後の数字は、その実行が何回目かを示す)

操作	元の配列	サブ配列	結果配列
初期状態	[3,7,1,9,5,7,4,6,8,2]	\square	\square
抽出 1	[1,5,7,4,6,8,2]	[3,7,9]	\square
マージ 1	[1,5,7,4,6,8,2]	\square	[3,7,9]
抽出 2	[4,6,2]	[1,5,7,8]	[3,7,9]
マージ 2	[4,6,2]	\square	[1,3,5,7,7,8,9]
抽出 3	[2]	[4,6]	[1,3,5,7,7,8,9]
マージ 3	[2]	\square	[1,3,4,5,6,7,7,8,9]
抽出 4	\square	[2]	[1,3,4,5,6,7,7,8,9]
マージ 4	\square	\square	[1,2,3,4,5,6,7,7,8,9]

ストランドソートはコムソートと並んで実装が楽なので、そこそこ高速な安定ソートが欲しい場合などにさっと実装できるのが利点である。

14.3.6 イン트로ソート

gcc の C++ のソートに使われているソートアルゴリズムが、ここで紹介するイントロソート (intro sort) である。これは、クイックソートの弱点をヒープソートでカバーしたソートアルゴリズムである。

クイックソートの弱点として、よく工夫された並びに対しては、実行時間が $O(n^2)$ になってしまうという点が挙げられる。実際、これを突いた Dos 攻撃も存在する。そこで、通常はクイックソートを使うことにしておき、再帰回数が増えた場合、その各部分列に対するソートをヒープソートに切り替える。更に、ヒープソートの最終段階において、挿入ソートを用いる。このように、クイックソート・ヒープソート・挿入ソートを組み合わせることで高速なソートを組むことができる。これがイントロソートである。

イントロソートは平均/最悪時間計算量 $O(n \log n)$ で、空間計算量 $O(\log n)$ の、不安定ソートである。

14.3.7 ティムソート

Python や Java7、あるいは C++ の Stable_sort などで行われる高速ソートアルゴリズムがティムソート (Tim sort) である。マージソートを基本に行っているが、Top-Down ではなく、Bottom-Up に使う部分が違う。

配列を先頭から順に見ていき、最初の 2 項を含む広義単調増加部分ないし狭義単調減少部分を見つける。狭義単調減少部分の場合、これを逆転させて狭義単調増加部分にする。この単調増加部分を為す要素の数が一定個数 S 未満の場合、それ以降の部分に対して二分挿入ソートを行い、各分割が一定の大きさ S になるようにする。この時、「一定の大きさ」 S は、配列の個数 n に対し、 $2^k S = n$ (k は自然数) という関係を満たし、かつ挿入ソートが最速である程度に小さい (概ね 32 個未満) ことが望ましい。条件を満たす S が複数ある場合、最大のものを選ぶ。

これによって配列が大きさ S の分割になったら、隣り合ったメモリ領域をマージする。この時、全要素をマージするのではなく、前側分割において後側分割の先頭より小さいものと、後側分割において前側分割の終端より大きいものを除いた部分列をマージすることで、マージ速度をあげる。実際には、この部分列の作成も二分探索等を用いて高速化する。

以上、マージソートに対して各種の高速化を用いたものがティムソートである。もともと並んでいるデータに対して $O(n)$ 、平均/最悪の場合 $O(n \log n)$ で動作する安定ソートで、いいことづくめのようなのであるが、残念ながら空間計算量 $O(n)$ を要する他、実際の実装もやや難しい。紹介程度に捉えておくと良い¹³。

14.3.8 バケットソート

バケットソート (Bucket sort) は非常に別名が多いソートで、分布数えソート (Distribution counting sort) やビンソート (bin sort) とも呼ばれる。これは比較ソートではなく、また別の手段による方法である。

試験の得点で順位を決める場合、分布を求めて順位を決定するだろう。この、分布を数えて順位付けを行い、その順位付けに応じてソートを行う方法が、バケットソートである。

バケットソートを行う場合は、要素が有限種類に限られなければならない。要素が m 種類であるならば、空間計算量は $O(m)$ となる。但し、この条件がクリアできた場合、計算時間 $O(n)$ で安定という大きな恩恵を得ることができるソートでもある。

バケットソート

【手順】

1. 要素の取りうる値 m に対し、 m 個のキューを用意しておく。この時、キューは要素順に対応するようにしておく。
2. 配列を順に見て、各要素を対応したキューに挿入する。
3. キューから要素を取り出していく。取り出し終わったら次のキューに移る。

【性質/特徴】

時間計算量 $O(n)$

空間計算量 $O(m)$

安定性 安定 (キューを用いた場合)

条件 要素の定義域が有限個の定値

この手順において「キューは要素順に対応するように」というのは、例えばテストの例の場合、100 点のキューを `queue[100]`, 99 点のキューを `queue[99]`, ... などとしておく、ということである。この条件を満たすように定めれば、第3ステップが「100 点のキュー

¹³安定ソートが必要な場合、実用的には部分列が十分小さくなった時に二分挿入ソートを使うようにしたマージソートで十分速く、実装も楽である。主に「既に組まれているものを使う」ぐらいであろう。

の要素を全て取り出して配列に格納する」「99 点のキューの要素を全て取り出して配列に追加する」… となり、たしかに $O(n)$ になるのである。

ここに述べたとおり、バケットソートは性能が良いソートであるが、メモリとのトレードオフになる。そのため、例えば電話番号の整列など、存在するかしないかだけの場合にはビットを立てるなどの方法でキューを表現してメモリを節約したほうが良い。

また、この方法を用いて大雑把に分割してソートを行う方法もある (キューの代わりにプライオリティーキューを用いる)。ここで紹介した基本形を元に、様々な工夫を加えて使えるソートである。

14.4 サーチとその手法

サーチ (search) は、必要な値を配列内から検索するために用いられる。サーチもソート同様、非常に多くのアルゴリズムが知られているが、ここでは紙面の都合上、リニアサーチとバイナリサーチを紹介するに留める。以下、 n 項の `int` 型配列に目標の整数が含まれているかどうかを判定し、含まれているときにその位置を見つける方法¹⁴について記す。

14.4.1 リニアサーチ

リニアサーチ (linear search) は誰しもがわかる方法で、配列を前から順に見ていく方法である。日本語では線形探索などと呼ばれる。時間計算量 $O(n)$ で動作する。

この方法は他のサーチの方法と違って前提条件がないため、少ない回数の探索などの場合には便利である。しかし、同じ配列を何度も探索する場合、前提条件を満たすようにしてから他のサーチアルゴリズムを使ったほうが効率が良いだろう。

14.4.2 バイナリサーチ

バイナリサーチ (binary search) は日本語では二分探索とも呼ばれるサーチで、配列がソートされていることを前提とし、 $O(\log n)$ で要素を見付け出す (あるいは無い事を示す) アルゴリズムである。

【bsearch 関数】

バイナリサーチそのものの説明の前に、関数を紹介しておこう。stdlib.h には、バイナリサーチに由来した¹⁵、bsearch 関数がある。この関数は、第 1 引数に、探したい値へのポインタを取る。残りの引数は qsort と同様である (但し、第 2 引数の配列は、第 5 引数の比較関数の規則において昇順ソート済みである必要がある)。これにより、bsearch 関数は、一致する要素があればそれへのポインタを、存在しなければ NULL ポインタを返す。注意すべき点として、複数の同じ要素がある場合、そのどれが返されるかは不明である (が、乱択では無いので一定ではある) という点である。

¹⁴このように、あるかないかの判別までするのがサーチの問題の一般的な形である。

¹⁵qsort 同様、バイナリサーチのアルゴリズムで組まれているかどうかは保証されていない。

ひとまず多数の回の探索を行う！という場合には、配列を qsort した後 bsearch を使うのが定番である。(リニアサーチと時間計算量を比較してみよ。)

【バイナリサーチのアルゴリズム】

それでは、バイナリサーチについて見ていくことにしよう。

ソートされているならば、配列の真ん中の値を取ることで中央値がわかる。探している値がこの中央値よりも小さければ、その値は前半にしかない。逆に、この中央値よりも大きければその値は後半にある。そこで、その半分の部分列について、中央値と探している値を比較してやる。これを繰り返していくと、探索範囲が順次半分になるため、 $O(\log n)$ 程度で配列の中に要素があるかどうかを判定できる。これがバイナリサーチのアルゴリズムの考え方である。

バイナリサーチのアルゴリズム

1. 「今から注目する部分列」を配列全体にしておく。
2. 「今から注目する部分列」の中央値と探したい値を比較する。一致していれば終了し、一致していなければ次項へ進む。
3. 探したい値が中央値より小さいならば「今から注目する部分列」を前半分に縮小する。逆に、大きい場合は後半分に縮小する。要素数が 0 になった場合は見つからなかったとして終了し、そうでない場合は前項に戻る。

バイナリサーチのアルゴリズムで重要なのは、これは配列の探索にかかわらず用いられる、という事である。例えば大小関係によって整数を当てるゲームでは、その整数の範囲に対して二分探索を行うことで解を求めることができる。

【範囲なしのバイナリサーチ】

バイナリサーチを座標などに用いる場合 (例えば、どこまで池かを知りたい時など)、無限に広い座標では「中央値」を定めることができない。そこで、ひとまず元となる 1 点を決めておき、そこから 1 移動した点、1+2 移動した点、1+2+4 移動した点、... と探索を行なっていく。そして、その結果が異なるようになった 2 点の間でバイナリサーチを行う。先の池の例であれば、点 0 が池で、1,3,7,15 の距離にある点も池であるが、31 の距離にある点が池でなかった場合、距離 15 から距離 31 の間の 16 の距離をバイナリサーチすることで池の端を見つけることができる。

この他、範囲の有無にかかわらずバイナリサーチを用いるアルゴリズムは多い。例えば、次講で学ぶ、方程式の解を求める二分法はまさしくバイナリサーチである。このように、バイナリサーチの考えは各所で出てくるため、ぜひ理解しておいていただきたい。

本講の要点

本講義では、ソートを中心に学び、それと関連の深いサーチについても触れた。

ソート

- ソートは与えられたデータを一定の規則に従って整列する問題である。
- ソートの規則は比較関数によって与えられる場合が多い。qsort 関数に対する比較関数については、strcmp と似たような条件が定まっている。
- ソートの性能を評価するための指標としては、時間計算量・空間計算量の他、同一値の複数要素の順序が変わるかどうか (安定性) などがある。
- 今回紹介したソートは次のとおりである。

名称	計算時間量 (平均/最悪)	空間計算量	安定性
バブル	$O(n^2)$	$O(1)$	安定
挿入	$O(n^2)$	$O(1)$	安定
選択	$O(n^2)$	$O(1)$	不安定
シェーカー	$O(n^2)$	$O(1)$	安定
ノーム	$O(n^2)$	$O(1)$	安定
シェル	$O(n \log^2 n)$	$O(1)$	不安定
マージ	$O(n \log n)$	$O(n)$	安定
クイック	$O(n \log n)/O(n^2)$	$O(\log n)$	不安定
ヒープ	$O(n \log n)$	$O(1)$	不安定
コム	$O(n \log n)$	$O(1)$	不安定
ストランド	$O(n \log n)/O(n^2)$	$O(n)$	安定
イントロ	$O(n \log n)$	$O(\log n)$	不安定
ティム	$O(n \log n)$	$O(n)$	安定
バケット	$O(n)$	$O(m)$	安定

サーチ

- サーチとは、与えられたデータの中に探したいデータが存在するか判定し、存在するならどこに存在するか求める問題である。
- リニアサーチは、前から順に要素を探索していく方法である。
- バイナリサーチは、ソート済みの配列に対してその中央値と探索値を比較し、範囲を狭めていく方法である。
- バイナリサーチの考え方は様々なアルゴリズムに応用される。

第15講 数値計算の基礎

コンピュータは電子”計算機”であるから、計算を行うという目的に使うことも当然多い。だが、コンピュータでは、極限などのような数学概念を扱うことができない(多項式の足し算などを行う”数式処理ソフトウェア”もあるが、これは数式を文字列として解釈し、その文字列に一定の処理を行なっているだけである)。そこで、式を扱うのではなく、数値そのものと四則演算を用いて計算を行うことになる。この計算方法が数値計算法(numerical method)である。この章では、分野を問わずよく使われるであろう基礎的な数値計算法を紹介する。より多くの数値計算法を知りたい場合は、巷間に出まわる多数の参考書¹を参照されたい。

15.1 1元方程式を解く

まず、方程式 $f(x) = 0$ の解を数値的に計算する初歩的な方法を学ぼう。以下、 $f(x) = 0$ は区間 $[a, b]$ において唯一の実解を持つという前提で説明を記す。但し、実際には解を複数持ってもよい。つまり、図 15.1 のような状況を思い描けば良い。

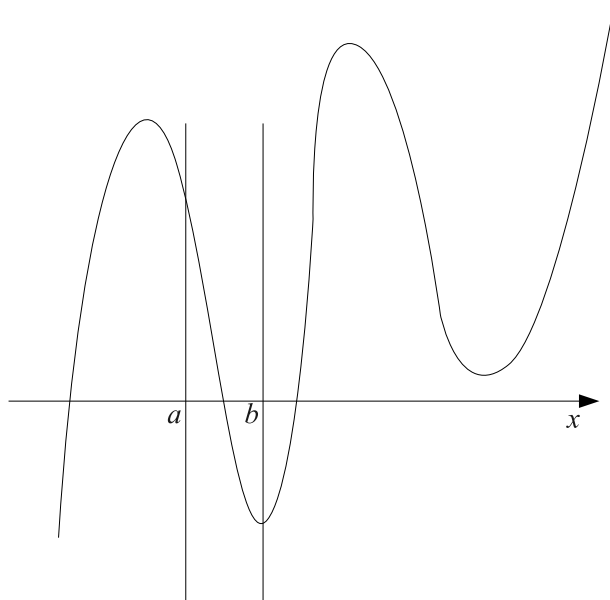


図 15.1: 方程式 $f(x) = 0$ の様子

複数の解を持つ場合は、区間を変えて解く事になる。つまり、このあと考えていくのは「区間内の1つの解を出す」方法である。

¹一冊だけ挙げると、「Numerical Recipes in C」(W.H.Press et.al 著, 丹慶他訳, 1993, 技術評論社) などが有名。

15.1.1 逐次探索法

方程式の解の存在がわかっているのであれば、「下手な鉄砲も数撃ちゃ当たる」という方針で、とにかく色々な値を代入してみれば良い。そのうち、方程式を満たす値が見つかるはずである。これを探索する方法の一つが逐次探索法 (serial search) である。

区間 $[a, b]$ の間を幅 h で見ていく。つまり、 $a, a+h, a+2h, \dots, b$ と代入して「解に近いもの」または解を探す。解が直接見つければもちろんそれで良い。解そのものでなくとも、解の周辺では符号が変わることから、 $f(a+kh) \cdot f(a+(k+1)h) < 0$ となるような場所が見つかるはずである。そこで、この区間 $[a+kh, a+(k+1)h]$ を、幅 h をより小さくして再度探索する。これを繰り返していき、望みの精度になったら (小さい正数 ϵ に対して、 $|f(x')| < \epsilon$ となるような x' が見つければ) それを解として出力すれば良い。

これは、解をリニアサーチしているのと同じ事である。計算時間がかかるので、通常はこのアルゴリズムが単独で用いられることはない。

15.1.2 二分法

先のように、解をリニアサーチしても良いのだが、区間が $[a, b]$ に定まっていることを利用し、この間をバイナリサーチして解を見つけることもできる。この方法を二分法 (bisection method) と呼ぶ。但し、 $f(a) \cdot f(b) < 0$ が条件である。以下、手順を示しておく。

二分法

1. $f\left(\frac{a+b}{2}\right)$ を計算する。これが十分 0 に近ければ解として終了する。そうでなければ次項に進む。
2. $f(a), f(b)$ のうち、先に計算した値と同符号の側の区間を縮める。例えば、 $f(b)$ が $f\left(\frac{a+b}{2}\right)$ と同符号であれば、 b を $\frac{a+b}{2}$ で置き換える。置き換えた後、前項に戻る。

【二分法+逐次探索法】

先に示した方法では $f(a) \cdot f(b) < 0$ が満たされない時などに使えない。そこで、前準備として逐次探索法を行い、適切な区間を定めてから二分法に切り替えることで高速に解を求めることもできる。

逐次探索付き二分法

1. 区間 $[a, b]$ に対して逐次探索を行い、 $f(x') \cdot f(y') < 0$ となるような x', y' を見つける。
2. $f\left(\frac{x'+y'}{2}\right)$ を計算する。これが十分 0 に近ければ解として終了する。そうでなければ次項に進む。
3. $f(x'), f(y')$ のうち、先に計算した値と同符号の側の区間を縮め (置き換え) て、前項に戻る。

【二分法/逐次近似法の利用と注意】

ここまでの議論では解が区間 $[a, b]$ に唯一存在することを仮定しているが、実際にこれがわかっていることは稀だろう。従って、ここまでに紹介した方法はどちらかというと区間 $[a, b]$ を知る (解を囲い込む) ために用いられる。これらの方法を適用すれば、 $f(x)$ の符号の変わり目を知ることができ、中間値定理からその解の存在を伺える。

だが、逐次探索法・二分法共に解が充分近い場合はこれらを分離できないことがある。この場合、単純に区間幅を縮めても良いのだが、速度が遅くなり誤差も出やすくなるので、トレードオフとなる。他の方法の場合、これらの近い複数解のうち一つを見つけ出すことができるのに対し、逐次探索法や二分法ではうまくいかないのである。

逐次探索法・二分法には更に大きな欠点がある。それは、重解を検知できないことである。 $(x - \pi)^2 = 0$ という方程式が重解 π を持つことは直ちにわかるが、この例のように無理数の重解を持つ場合、符号の変わり目がなく、「たまたま調べた場所が 0 になる」ことも少ないため、解が無いという事になってしまうのである。

その一方、これら 2 つの方法は急激な変化が起こる関数 (例えば、 \sin) のような形をした関数) に対しても対応するという利点がある。

以上のように、方程式の解法は一つに絞るのではなく、様々な種類のものを状況に応じて使い分ける (時に併用する²⁾) べきなのである。

15.1.3 割線法

割線法 (secant method) は図 15.2 のように、仮の解 x_0, x_1 を定めておき、その 2 点を通る直線と x 軸との交点を定め、それを次の「仮の解」 x_2 として、どんどん解を改良していく方法 (総称して反復法 (iteration method) と呼ばれる) である。

【割線法の漸化式】

図 15.2 は人間の目で見れば何をしているか直ちにわかる。だが、コンピュータは図から理解できるわけではないため、何らかの数式で表してやらなければならない。つまり、それまでの仮の解から、次の仮の解を数式によって決定しなければならないのである。以下、この漸化式を導出してみよう。

今、2 つ前の仮の解が x_{n-2} であり、1 つ前の仮の解が x_{n-1} であったとしよう。この時、この 2 点を通る直線の方程式は

$$y - f(x_{n-1}) = \frac{f(x_{n-1}) - f(x_{n-2})}{x_{n-1} - x_{n-2}}(x - x_{n-1}) \quad (15.1)$$

²実際、Brent 法などでは何種類かの方法を状況に応じて戦略的に使い分けている。ソートでも、イントロソートは複数のソートを組み合わせていた。このように、アルゴリズムは状況に応じて適切に選択される必要があるものであり、一つ知っていればそれでいいという事はないのである。

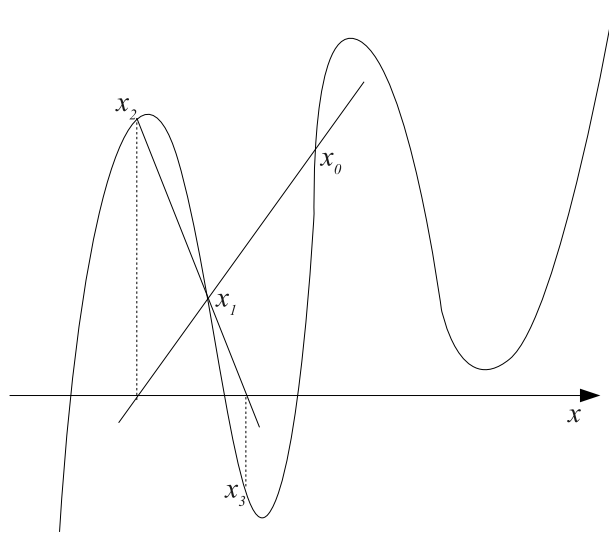


図 15.2: 割線法の様子

である。この直線と x 軸との交点が $(x_n, 0)$ なのだから

$$-f(x_{n-1}) = \frac{f(x_{n-1}) - f(x_{n-2})}{x_{n-1} - x_{n-2}}(x_n - x_{n-1}) \quad (15.2)$$

である。これを x_n について解いて

$$x_n = x_{n-1} - \frac{x_{n-1} - x_{n-2}}{f(x_{n-1}) - f(x_{n-2})}f(x_{n-1}) \quad (15.3)$$

を得る。後は、この漸化式を順次計算すれば良いのである。以下、これを手順化しておく。

割線法

1. 仮の解 x_0, x_1 を定める。また、 $n = 2$ とする。
2. 式 (15.3) を用いて x_n を計算する。
3. 小さい正数 ϵ について、 $|x_n - x_{n-1}| < \epsilon$ か、 $|f(x_n)| < \epsilon$ が満たされている場合はそれを解として終了する。そうでない場合、 n を 1 増やして前項に戻る。

【割線法の利用と注意】

割線法は必ずしも収束するとは限らない他、0 に近い極値がある場合、それに影響されやすいという欠点を持つ。しかし、大抵の場合には (初期値さえ適切に選べば) きちんと解を出してくれるので、比較的使いやすい。一度やって収束しなくても、初期値を変えたら収束するような場合もありうるためである。だが、割線法は、複数解がある場合にどれを出すかわからないため、少し厄介である。

ここでは 2 点からの直線近似としたが、3 点からの二次関数近似を利用して解を出す方法もある (Muller 法)。Muller 法は複素根も求められる優秀な方法であるが、その考えはここで紹介した割線法と原則同じである。そのため、割線法についてまず理解を深め、それから他の書籍などで Muller 法を理解されると良いだろう。

15.1.4 はさみうち法

割線法は解を囲い込めないという欠点があった。そこで、囲い込み区間を決定し、その間の解を求められるように割線法を改良したのがはさみうち法あるいはレギュラ・ファルシ法 (regula falsi method) である。

はさみうち法

1. 仮の解 a, b を定める。この時、 $f(a) \cdot f(b) < 0$ を満たすものとする。これから探し求めるのは区間 $[a, b]$ の実解である。
2. 式 (15.3) において、 $x_{n-1} = a, x_{n-2} = b$ とおいて x_{n+2} を計算する。
3. $f(x_{n+2})$ が十分 0 に近ければそれを解として終了する。そうでなければ、二分法同様に、 a, b のうち符号の同じ側を x_{n+2} で置き換え、前項に戻る。

はさみうち法は、割線法よりは収束が遅いが、解を囲い込めるのが利点である。初期値によっては二分法よりも遅いが、平均的にはこちらのほうがよく収束する。但し、重解に使えない点は同じである。

15.1.5 Newton 法

割線法では 2 点を取り、それを直線近似することにより解を得た。だが、関数を直線で近似するときには、二点の平均変化率で近似するより各点の接線で近似したほうが正確になるだろう。この考えに基づくアルゴリズムが **Newton 法** ないし **Newton-Raphson 法** と呼ばれるアルゴリズムである。重要なアルゴリズムであるので、まず使い方を述べた後、何種類かの導出を紹介し、その後実例となるソースを一つ見てみることにする。

【Newton 法のアルゴリズム】

Newton 法は、割線法と同様に仮の解を定めて、それを改良していく方法である。但し、割線法と違い、接線近似をするために仮の解は一つだけあれば良い。従って、Newton 法で解を改良するための漸化式は二項間漸化式であり、次のように与えられる。

$$x_n = x_{n-1} - \frac{f(x_{n-1})}{f'(x_{n-1})} \quad (15.4)$$

この漸化式には何種類かの導出方法があり、後にその内の 3 種類を示す。先にその手順を示そう (ほとんど割線法と同じである)。

Newton 法

1. 仮の解 x_0 を定める。 $n = 1$ としておく。
2. 漸化式 (15.4) を用いて x_n を計算する。
3. 小さい正数 ϵ について、 $|x_n - x_{n-1}| < \epsilon$ か、 $|f(x_n)| < \epsilon$ が満たされている場合はそれを解として終了する。そうでない場合、 n を 1 増やして前項に戻る。

漸化式 (15.4) は収束しない場合がある。例えば、 $x_{n+1} = x_{n-1}$ となる場合や、極値周辺に落ち込んだ場合などに弱い。従って、ある程度収束の状況を見ておき、適切に収束しない場合は初期値を変えるなどの工夫をしなければならない。幸い、Newton 法の収束はここまでで紹介した中でも一番速いので、やり直しても大した手間ではない。

また、関数の極値の大体の位置がわかっているならば、その x_n が極値の間から出た時にやり直す、などとすることで収束しやすくできる (上、区間を絞って解を見つけることもできる)。

【割線法の極限としての導出】

割線法は二点を通る直線によって計算を行う方法であった。この二点を近づけた時に得られる直線が接線になることは先刻承知であろう。従って、割線法の式 (15.3) の両辺について $x_{n-2} \rightarrow x_{n-1}$ の極限をとった

$$x_n = \lim_{x_{n-2} \rightarrow x_{n-1}} \left(x_{n-1} - \frac{x_{n-1} - x_{n-2}}{f(x_{n-1}) - f(x_{n-2})} f(x_{n-1}) \right) \quad (15.5)$$

を計算してみる事で、Newton 法を導出することができそうである。

ここで、

$$\lim_{x_{n-2} \rightarrow x_{n-1}} \frac{x_{n-1} - x_{n-2}}{f(x_{n-1}) - f(x_{n-2})} = \lim_{x_{n-2} \rightarrow x_{n-1}} \frac{1}{\frac{f(x_{n-2}) - f(x_{n-1})}{x_{n-2} - x_{n-1}}} = \frac{1}{f'(x_{n-1})} \quad (15.6)$$

である。このことから、式 (15.5) は

$$x_n = x_{n-1} - \frac{f(x_{n-1})}{f'(x_{n-1})} \quad (15.7)$$

となり、たしかに式 (15.4) と一致する。

以上のように、Newton 法は割線法の極限として見ることができる。逆に、微分を差分近似することで割線法に戻すことも可能である (これについては後に述べる)。

【接線近似からの導出】

先の方法では、平均変化率を求めてから極限を取ることにした。だが、最初から接線を計算すればそれで問題なく導出することができる。釈迦に説法かもしれないが、ここで接線を用いて導出しておこう。

点 $(x_{n-1}, f(x_{n-1}))$ を通る接線の方程式は

$$y - f(x_{n-1}) = f'(x_{n-1})(x - x_{n-1}) \quad (15.8)$$

である。この接線と x 軸の交点の x 座標が x_n であるので

$$-f(x_{n-1}) = f'(x_{n-1})(x_n - x_{n-1}) \quad (15.9)$$

となる。これを解くことで、式 (15.4) が導かれる。

先の割線法からの導出は割線法との関係がよく見える方法である。だが、このように接線近似を行ったほうが手早く導出できる。漸化式 (15.4) を忘れた時にはこの方法で導出するのが一番手っ取り早いだろう。

【解析的導出】

Newton 法は Taylor 展開を用いた微分近似によっても求められる。この方法は別の解法の導出の際にも用いられるため、ここで Newton 法を例にして紹介しておくことにする。

$f(x)$ を $x = x_n$ で Taylor 展開すると

$$f(x) = f(x_n) + f'(x_n)(x - x_n) + \cdots \quad (15.10)$$

となる。この右辺の二次以上の項を落として $f(x)$ を一次近似すると

$$f(x) \approx f(x_n) + f'(x_n)(x - x_n) \quad (15.11)$$

となる。ここから、右辺=0 の解を求めれば

$$x = x_n - \frac{f(x_n)}{f'(x_n)} \quad (15.12)$$

となる。後は、これを順次適用していけば解に近づいていくだろうことから、 x を x_{n+1} で置き換えれば、たしかに式 (15.4) を得られる。

このように、方程式を (必要に応じて近似を行い) 変形して左辺と右辺にわけ、右辺を既知項・左辺を未知項として漸化式を導出するのは、反復法において非常によく使われる導出法である。

途中、Taylor 展開で二次以上を落としたが、二次項まで残すとまた別の解法になる。

【数値微分との併用】

Newton 法の漸化式には、求めたい方程式の導関数がある。導関数が簡単に求まればいいのだが、そうは行かない場合もあるだろう。そこで、コンピュータを用いて数値的に微分を計算 (数値微分 (numerical differentiation)) してしまえば良いのでは? という考えに至る。

コンピュータで微分を計算する場合、基本となるのは差分近似である。以下、Taylor 展開を用いて、1 階微分の前進差分近似 (forward difference) と、中央差分近似 (centered difference) を導出してみよう。

Taylor 展開より、十分小さい h について、

$$f(x+h) = f(x) + hf'(x) + \cdots \quad (15.13)$$

である。ここまではどちらの微分近似の導出も同じである。

前進差分近似では、式 (15.13) の二次以上の項を落として、 $f'(x)$ について解く。すると、

$$f'(x) \approx \frac{f(x+h) - f(x)}{h} \quad (15.14)$$

となる。この時の打ち切り誤差は、二次以上の項を落として h で割っているので、 $O(h)$ である。

中央差分近似では式 (15.13) を用いて、 $f(x+h) - f(x-h)$ を計算する。これにより、右辺は奇数次の項しか残らない。

$$f(x+h) - f(x-h) = 2hf'(x) + \cdots \quad (15.15)$$

ここで、三次以上の項を落として $f'(x)$ についてとけば

$$f'(x) = \frac{f(x+h) - f(x-h)}{2h} \quad (15.16)$$

を得る。この誤差は $O(h^2)$ である。

今、式 (15.14) を用いて Newton 法の計算を行うことを考えよう。式 (15.4) に式 (15.14) を代入して

$$x_n = x_{n-1} - \frac{hf(x_{n-1})}{f(x_{n-1}+h) - f(x_{n-1})} \quad (15.17)$$

を得る。これは、微分を差分化しているので割線法と同様の形式である。つまり、Newton 法に前進差分を適用すると割線法に帰着するという事である。これは何も前進差分に限った話ではない。中央差分近似を始めとする他の方法を使ったとしても、割線法と本質的には同じアルゴリズムになる (平均変化率の計算が少し変わるだけ)。

ここで述べた方法により Newton 法に数値微分を組み合わせた場合、割線法を (任意の幅で) 用いる場合に比べて収束が遅い場合が多い為、これを使うぐらいなら最初から割線法で攻めたほうが良いだろう。しかし、割線法と Newton 法の関係が、差分と微分の関係であることを理解するのに重要なことであるため、ここで述べた。

なお、蛇足ながら、数値微分の差分近似方法はここに紹介した以外にも沢山あり、Taylor 展開から導出できる。例えば、 $f(x+h) + f(x-h)$ から二階微分の公式が導出できる他、 $f(x), f(x \pm h), f(x \pm 2h)$ を用いて一階微分を計算することにより、より精度の良い公式が導出できる。興味があれば計算してみたい。

【Newton 法の実例～平方根の計算～】

ここまでで、Newton 法の素性について学習したので、これを実際を使って正平方根を計算する関数を作成してみることにしよう。

ソースコードの前に、Newton 法の漸化式をどう書けばいいか考えておく。数 a の正平方根 \sqrt{a} を解に持つ方程式のうち、 x と a 及び定数だけで簡単に書けるものとして

$$x^2 - a = 0 \quad (15.18)$$

という方程式が挙げられる。これを Newton 法により解く。

$f(x) = x^2 - a$ として、式 (15.4) を書き下すと

$$x_n = x_{n-1} - \frac{x_{n-1}^2 - a}{2x_{n-1}} \quad (15.19)$$

である。初期値は、正の値ならば (その形状を推測して) 何でも良いが、ここでは決めやすく a としておこう。これにより、Newton 法を書く準備ができたので、以下に引数の正平方根を計算する関数を実装してみることにする。

【Newton 法による平方根の計算】

Newton 法を用いて引数の正の平方根を計算する関数 `newton_sqrt` を実装してみる。ここでは、収束の判定を、解候補の二乗と元の数の差が 10^{-6} 未満であるとし、マクロ `EPS` によって定めるものとした。なお、収束しない場合の処理などは行っていない。

リスト 15.1: Newton 法による開平

```

1 #include<math.h>
2 #define EPS 1E-6
3
4 double newton_sqrt(double a){
5     double x=a;
6     do{
7         x=(x*x+a)/(2*x);
8     }while(fabs(x*x-a)>=EPS);
9     return x;
10 }
```

このように、解を求めることができる方程式に対しても、その解の数値の計算のためなどの理由で、数値解法を用いる場合があるのである。

15.2 数値積分法

ある定積分を数値的に計算する。積分は面積であるから、その面積を何らかの近似で求められれば良い。この時、区間を大きく取ると近似しづらくなるので、

$$\int_a^b f(x) = \sum_{k=0}^{n-1} \int_{x_k}^{x_{k+1}} f(x) dx \quad (x_0 = a, x_n = b) \quad (15.20)$$

のように区間に分割して計算を行う。数列 $\{x_n\}$ を $x_{n+1} = x_n + h$ として決めれば、これは等分割になることがわかるだろう。ここでは、このような等分割の場合³に、右辺の積分をどのように近似するかを見ていくことにする。最後に、少し違ったアプローチでの積分方法を紹介する。

15.2.1 台形則

以前の講義で既に紹介したが、台形則ないし台形積分公式は、式 (15.20) 右辺の積分を台形によって近似し

$$\int_{x_k}^{x_{k+1}} f(x) dx = \int_{x_k}^{x_k+h} f(x) dx \approx \frac{h}{2} (f(x_k) + f(x_k + h)) \quad (15.21)$$

と近似する方法である。

³ガウス・ルジャンドルの積分公式など等分割にしない方法も存在する。

15.2.2 中点則

中点則 (midpoint integral) は、区間分割の近似を台形ではなく長方形で行う方法である。この時、長方形の幅が h であることはすぐにわかるが、高さには幾つかの選び方がある。 $f(x_k)$ や $f(x_k + h)$ を高さにする方法は高校などで区分求積法として説明されるが、中点則ではその名前通り、中点を高さにとる。つまり、 $f(x + 0.5h)$ を高さの基準として用いる。これにより、式 (15.20) 右辺の積分は

$$\int_{x_k}^{x_{k+1}} f(x)dx = \int_{x_k}^{x_k+h} f(x)dx \approx hf(x_k + 0.5h) \quad (15.22)$$

と簡単に近似できる。

通常、数値積分では台形則の方がよく用いられるが、このような方法があることも知っておくと良い。

15.2.3 シンプソン則

シンプソン則 (Simpson integral) は古典的な積分公式として知られている方法である。台形則よりも時間がかかるため精度が必要ない場合には使われず、精度が必要な場合はローンバーク積分やガウス・ルジャンドル積分の方が有用であるのでやはり使われないという、博物館行きになった感のある公式である。とはいえ、初歩として知っておく価値があるので、ここでその概要を紹介しておこう。

$x_k, x_k + 0.5h, x_k + h$ から、関数を二次関数で近似することを考える。ただし、この形式で書くのは面倒であるので、 $x'_- = x_k, x' = x_k + 0.5h, x'_+ = x_k + h$ および $h' = 0.5h$ とおいて議論を進める (こちらのほうが一般的な形になるため)。また、 $f'_- = f(x'_-), f'_+ = f(x'_+), f' = f(x')$ とする。

三点を通る二次関数 $g(x)$ は唯一に定まる。そこで、(簡単のため) 二次関数を

$$g(x) = a(x - x')^2 + b(x - x') + c \quad (15.23)$$

とおき、これが3点を通るように a, b, c を定めることとする。単純に各点を代入して解けば

$$(a, b, c) = \left(\frac{f'_- + f'_+ - 2f'}{2h'^2}, \frac{f'_+ - f'_-}{2h'}, f' \right) \quad (15.24)$$

となる。一方、

$$\int g(x)dx = \frac{a}{3}(x - x')^3 + \frac{b}{2}(x - x')^2 + cx \quad (15.25)$$

である (ただし、積分定数を省いた)。

式 (15.25) に式 (15.24) を代入して整理し、定積分の形に書き直すと

$$\int_{x'_-}^{x'_+} f(x)dx \approx \int_{x'_-}^{x'_+} g(x)dx = \frac{h'}{3}[f'_- + 4f' + f'_+] \quad (15.26)$$

を得る。これが、シンプソン則である。

シンプソン則を用いる場合は、先のように中点を用いる場合もあるし、3点ずつ適用するように変える方法もある。

15.2.4 モンテカルロ積分

モンテカルロ積分 (Monte Carlo integral) は、ここまでの方法とは変わった形で積分を行う方法で、とりわけ多次元積分や解析的に書きにくい領域の面積を求めるのに用いられる。なお、一般のアルゴリズムでモンテカルロ法 (Monte Carlo method) と言った場合、乱数を用いて計算を行うアルゴリズムで、それ故に結果が正しいとは限らないもの⁴のことを言う⁴。ここで挙げたモンテカルロ積分もモンテカルロ法の一種である。

積分したい領域を図 15.3 のように長方形で囲む。そして、囲んだ領域に対し、一様乱数を用いて⁵点を「撒く」。そして、このうち幾つの点が領域内に入っているか、その割合を出せば囲んだ領域の近似となるだろう。

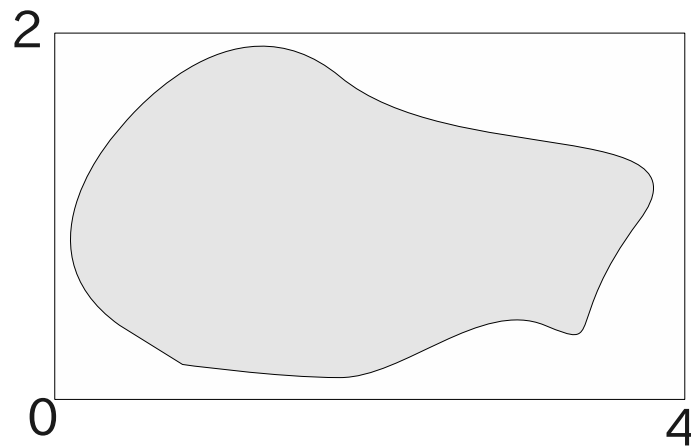


図 15.3: モンテカルロ積分したい領域の様子 (灰色領域)

この近似方法を手順にしてまとめておこう。

二次元のモンテカルロ積分

1. 計算したい領域を長方形で囲む。また、 $c = 0$ とする。
2. 次の 2 つの操作を n (任意の自然数値) 回繰り返す。
 - (1) 長方形内の点を一点、一様乱数を用いて選ぶ。
 - (2) 選んだ点が積分領域内に入っているかどうかを判定し、入っている場合は c を 1 増やす。
3. 長方形の面積 $\times c/n$ を計算し、それを積分値とする。

この方法は、3 次元の場合でも長方形を直方体に変えるだけで実行できるので、多次元積分に使いやすい。また、積分領域が解析的に書き表しにくい場合でも、領域として判定できさえすればいいので、台形積分が使えない場合などでも使えるという利点がある。

⁴これに対し、乱択クイックソートのように、乱数を使うがゆえに時間が一定に定まらない (解は正しく出る) ものをラスベガス法 (Las Vegas method) と呼ぶ。

⁵これがどの程度一様か、と言う所で正確性が定まる。計算向けには C 言語の標準関数の rand を用いるよりも、メルセンヌ・ツイスターと呼ばれる方法を用いたほうが良い。

モンテカルロ積分の精度は、 n が十分大きいならば、寧ろ一様性による。つまり $n = 2^{16}$ であるのを $n = 2^{20}$ に増やすよりも、乱数がより一様に出るようにしたほうが精度が良くなる。十分良い規則があるならば、点を選ぶのに乱数を用いず、規則性を用いて一様性を確保することもできる (準モンテカルロ積分 (semi-Monte Carlo integral) と呼ぶ)。だが、例えば細かい波状の境界の場合などは、適切な規則性が見当たらず、乱数を用いて選んだほうが適切に点を選べる場合もある。

15.3 連立方程式と行列

多元一次連立方程式を解く際に行列を用いる方法については既知であろう。この行基本変形をシステマティックに行うことによって、数値的に連立方程式の解を得ることができる。連立方程式を解くことは、数値計算の各所で必要になる操作であるため、ここでその基本を紹介しておく。また、その簡単な応用として、逆行列や行列式を求める方法についても触れる。なお、ここで紹介する方法についてのライブラリを作成するのは、配列・構造体・ポインタなどの良い復習になるので、できれば汎用ライブラリとしてまとめてみると良い。

15.3.1 Gauss 法

Gauss 法 (Gaussian Elimination) は、連立方程式を解く方法では最もポピュラーな方法であり、前進消去過程と後退代入過程からなる。

【前進消去過程】

連立方程式を表すため、以下、 n 行 $n+1$ 列からなる拡大係数行列 A を考える。もちろん、第 $n+1$ 列は連立方程式の定数ベクトルである。また、 $a_{i,j}$ と書いた場合、これは行列 A の i, j 成分を指すものとする。

拡大係数行列の下三角成分を 0 にする過程が前進消去過程である。前進消去過程では、第 k 行を用いて第 $k+1$ 行以降の各行の第 k 列成分を 0 にする。これを $k = 1$ から順に行うことで、下三角成分を 0 にすることができる。手順をまとめておこう。

Gauss 法の前進消去過程

1. $k = 1$ としておく。
2. $i = k+1, k+2, \dots$ 及び $j = k+1, k+2, \dots$ として、次の計算を行う (ただし、 $-$ は C 言語の `--` と同じ意味)。

$$a_{i,j} = \frac{a_{i,k}}{a_{k,k}} a_{k,j} \quad (15.27)$$

3. $k = n$ であれば終了する。そうでなければ、 k を 1 増やし、前項に戻る。

なお、実際の計算では、下三角が0になることはわかっているの、影響がある上三角部分だけ計算するようにしている。

以上により前進消去過程が行われたら、続いて後退代入過程を施し、連立方程式を解く。

【後退代入過程】

前進消去が行われた状態では、一番下(第 n 行)は単なる一次方程式になっている。そこで、これを解くことで、解のうちの一つを求めることができる。求めた解を、一つ上(第 $n-1$ 行)に代入して計算すれば、解成分がもうひとつ明らかになる。このように、下側から代入して解を求める…を繰り返すことで、連立方程式の解を求めることができる。これが、後退代入過程である。

Gauss 法の後退代入過程

1. $k = n$ としておく。また、 x_i は解ベクトルの第 i 成分とする。
2. 次の式により、 x_k を求める(ただし、 $k = n$ の時には総和部分は計算されない)。

$$x_k = \frac{1}{a_{k,k}} \left(a_{k,n+1} - \sum_{i=k+1}^n a_{k,i} x_i \right) \quad (15.28)$$

3. $k = 1$ であれば終了する。そうでなければ、 k を1減らし、前項に戻る。

なお、この方法を用いるときに、総和を逐次求めるのではなく、 x_k がわかった時にそこから上の行について減算を行なっておくという実装もできる。

実際には、ここまで述べてきた前進消去と後退代入をあわせて Gauss 法と呼んでおり、これらは一連の作業である。だが、分けたほうがこのあとの説明に便利であるため、2つに分けて説明した。

【部分ピボット選択】

前進消去過程の式(15.27)においては、割り算を伴うため、0による除算をしないように気をつけなければならない。だが、計算過程において $a_{k,k}$ が0になってしまう場合は十分にありうる。こんな時に行うのがピボット選択(pivot selection)である。ここで、ピボットというのは $a_{k,k}$ のように、行/列とも同じ番号の成分のことを言う。

もしもピボット $a_{k,k}$ が0になってしまった場合(あるいは絶対値が非常に小さい場合)、そこから下の行を見ていき、0でない成分 $a_{j,k}$ ($j > k$)を探し求める。適切な j が見つかったら、第 k 行と第 j 行を入れ替える⁶。このような行交換を行う方法が部分ピボット選択である。なお、行交換に加えて列交換も行う完全ピボット選択もあるのだが、ここでは名前だけの紹介に留める。

部分ピボット選択は、0による除算の回避のためだけにあるのではない。実際は、小さ

⁶蛇足ながら、浅いコピーを用いたほうが速度が速くて良い。

い値による割り算を行うと誤差が出たりする場合があるため、それを緩和する役目も持っている。一般には、同じ列の成分のうち、最も絶対値の大きい値をピボットに持ってくるのと良いとされる。これを含めて、前進消去過程を書き換えよう。

部分ピボット選択付き前進消去

1. $k = 1$ としておく。
2. $a_{i,k}$ ($i \geq k$) のうち、絶対値が最も大きい値を求める ($a_{m,k}$ とする)。求めた m に対し、 m 行と k 行を交換する。
3. $i = k + 1, k + 2, \dots$ 及び $j = k + 1, k + 2, \dots$ として、式 (15.27) の計算を行う。
4. $k = n$ であれば終了する。そうでなければ、 k を 1 増やし、前々項に戻る。

なお、部分ピボット選択の過程において、ある列で 0 の要素しか見当たらなくなった場合、その連立方程式には解が存在しないか、解が無数にある。この判定ができることも含めて、連立方程式を解く際には部分ピボット選択をつけておくと良い。

【LU 分解】

同じ係数行列でありながら、定ベクトルが異なる複数の連立方程式を解くことがある。このような場合に高速化を図る手法が **LU 分解** (LU decomposition) である。ここでは、数学的な議論は割愛し、プログラミングの観点から LU 分解を扱う。

同じ係数行列・異なる定ベクトルであれば、前進消去過程の大半 (定ベクトルに対する計算以外の部分) は無駄である。従って、何とかしてこの無駄な部分を省くことはできないか、と考える。

定ベクトルの計算に必要なのは、消去の時に、その行を何倍して引いたかという情報だけである。つまり、各過程における $\frac{a_{i,k}}{a_{k,k}}$ さえわかれば、後は定ベクトルを計算することができる。従って、 $\frac{a_{i,k}}{a_{k,k}}$ を記録しておこう、という事になる。この時、行列の下三角成分は無益に残っているだけなので、そこにメモすれば良い。これを手順化すると、次のようになる。

行列の LU 分解

以下、拡大係数行列ではなく、 $n \times n$ の係数行列を考えるものとする (ピボット選択は省いたが、省かずとも同じ手順である)。

1. $k = 1$ としておく。
2. $i = k + 1, k + 2, \dots$ 及び $j = k + 1, k + 2, \dots$ として、式 (15.27) の計算を行う。
更に、この時の $\frac{a_{i,k}}{a_{k,k}}$ の値を $a_{i,k}$ に保存しておく。
3. $k = n$ であれば終了する。そうでなければ、 k を 1 増やし、前項に戻る。

一度上記の手順を踏んで計算すれば、後は次の手順で (異なる定ベクトルに対しても) 解を求められる。

LU 分解後の解の計算

以下、定ベクトルの第 i 成分を b_i と記す。

1. $k = 1$ としておく。
2. $i = k + 1, k + 2, \dots$ とし、 b_i から $a_{i,k} \cdot b_k$ を引く。
3. $k = n$ ならば次項へ進む。そうでなければ k を 1 増やして前項に戻る。
4. 後退代入過程を行い、解を計算する。

「その方程式を解くのは一度限り！」という場合を除いては、この方法により計算すれば同じ計算を省略することが出来る。

15.3.2 Gauss-Jordan 法

Gauss-Jordan 法は Gauss 法よりも少し遅いが、間違えづらい方法で、特に逆行列への応用が効きやすい方法である。なお、ここでもやはり拡大係数行列を用いて説明を行う。

Gauss-Jordan 法は、係数行列に行基本変形を行い、単位行列に変換することによって連立方程式を解く方法である。早速手順を見てみよう (部分ピボット選択を含んでいる)。

Gauss-Jordan 法

1. $k = 1$ としておく。
2. $a_{k,k}$ が 0 である場合、 $a_{i,k} \neq 0$ ($i > k$) を満たす行を見つけ、第 i 行と第 k 行を交換する。
3. 第 k 行の全ての成分を $a_{k,k}$ で割っておく。これにより、 $a_{k,k} = 1$ となる。
4. $i \neq k$ なる全ての i に対し、第 i 行から ($a_{i,k} \times$ 第 k 行) を引く。
5. $k = n$ ならば終了する。そうでなければ k を 1 増やし、ステップ 2 に戻る。

この変形により、係数行列は単位行列になり、定ベクトル (拡大係数行列の第 $n + 1$ 列) は解ベクトルそのものになっている。

この方法は Gauss 法に比べて実装が楽なので、しばしば、連立方程式の解を出すアルゴリズムの試験のために用いられる。

15.3.3 行列式や逆行列への応用

Gauss 法や Gauss-Jordan 法は行基本変形を元になっているため、他の行列操作にも応用できる。ここでは、Gauss 法を行列式に、Gauss-Jordan 法を逆行列に応用してみよう。

【Gauss 法による行列式の計算】

行列式の計算を行う場合、第 1 列の成分のうち第 1 行以外を 0 にすれば、これが定数となって出てくる。すなわち、

$$\begin{vmatrix} a_{1,1} & \cdots & a_{1,n} \\ 0 & \cdots & a_{2,n} \\ \vdots & \ddots & \vdots \\ 0 & \cdots & a_{n,n} \end{vmatrix} = a_{1,1} \begin{vmatrix} a_{2,2} & \cdots & a_{2,n} \\ \vdots & \ddots & \vdots \\ a_{n,2} & \cdots & a_{n,n} \end{vmatrix} \quad (15.29)$$

とできる。ここで、 $a_{k,k}$ を用いてそこから下の成分を 0 にするのは、Gauss 法の前進消去過程そのものであることを思い出せば、前進消去の際にピボットを随時掛け算していくことによって行列式の値を計算することができることがわかるだろう。

【Gauss-Jordan 法による逆行列の計算】

係数行列と単位行列をくっつけた

$$\left[\begin{array}{ccc|ccc} a_{1,1} & \cdots & a_{1,n} & 1 & \cdots & 0 \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ a_{n,1} & \cdots & a_{n,n} & 0 & \cdots & 1 \end{array} \right] \quad (15.30)$$

を用意し、これに対して Gauss-Jordan 法を適用すると逆行列を得ることができる。これは、Gauss 法でも同じ事である。特段説明することはないだろう。

注意すべき点として、この方法で逆行列を求めてから連立方程式を解くと、速度が遅く、誤差も大きくなりがちである。これは、連立方程式に限ったことではない。例えば、 $x_{k+1} = A^{-1}x_k$ のような漸化式がある場合、 $Ax_{k+1} = x_k$ と変形して LU 分解を用いたほうが逆行列を計算するよりも (特に誤差の面で) 良い結果が得られる。従って、この方法で逆行列を得るのは、あくまでも逆行列そのものに興味がある場合と考えて良い。

実は、この方法は逆行列を得る方法としても最適のものではない。実際には、「第 i 成分が 1 で他は 0」という解ベクトルを $i = 1, \dots, n$ として用意し、LU 分解を用いて解いたほうが良い。実際に計算回数を見積もってみたい。

本講の要点

本講では、分野を問わずよく使われるであろう基礎的な数値計算法を学んだ。

一元方程式の解法

- 代数方程式の解を数値的に出す場合、一度に全ての解を出すことは通常難しいため、区間を絞って一つずつ出していくことになる。
- 区間をリニアサーチするのが逐次探索法、二分探索するのが二分法である。
- 割線法は曲線上の2点を取り、それらを結ぶ直線を引いて x 軸との交点を求めるという操作を繰り返すことで解を求める方法である。
- 割線法に、二分法と似たような区間制限をつけたのがはさみうち法である。
- 割線法の極限として、接線を用いて近似したのが Newton 法である。

数値積分法

- 数値積分を行う場合は通常、小区間に分割し、その各区間を近似的に計算して総和を求める形で積分する。
- 台形則は台形・中点則は中点の高さの長方形で区間を近似する。
- シンプソン則は関数上に3点を取り、そこを二次関数近似することによって積分を計算する。
- モンテカルロ積分は、領域の面積を求めるのに適した方法で、点を「撒いて」、それが求めたい領域に入った割合によって計算する方法である。

連立方程式と行列

- Gauss 法は下三角成分を0にする前進消去過程と、それによって解を順次求めていく後退代入過程からなる。
- Gauss-Jordan 法は係数行列を単位行列に変換することによって解を求める方法である。
- 同じ係数行列に対して前進消去過程を繰り返すのは無意味であるので、それをメモしておいて複数の定ベクトルに使いまわす方法が LU 分解である。
- 行基本変形のうち、行交換の過程のことを部分ピボット選択と呼んでいる。
- 連立方程式を解くアルゴリズムは、逆行列や行列式の値を求める際に容易に応用できる。

付録 A 簡易リファレンス

C 言語の簡易リファレンスを掲載しておく。紙面の都合上、記述は最小限度にとどめたので、利用方法については `man` コマンドはじめインターネット等で調べて頂きたい。なお、以下の書籍・サイトを参考にした。

- C 言語によるプログラミング [応用編] 第 2 版 (内田他著、Ohm 社刊)
- C 言語によるプログラミング [スーパーリファレンス編] (内田他著、Ohm 社刊)
- C 言語プログラミング (H.M.Deitel 他著、ピアソン・エデュケーション刊)
- プログラミング言語 C 第 2 版 (B.W.Kernighan 他著、共立出版刊)
- C リファレンスマニュアル第 5 版 (S.P.Harbison 他著、SiB access 刊)(C99)
- プログラミング言語 C の新機能 (<http://seclan.dll.jp/c99d/>)(C99)
- C 言語関数辞典 (<http://www.c-tipsref.com/>)(C99)

なお、末尾に C99 と付したものは C95, C99 で追加された機能について用いたものである。

特に断らない限り、大文字で書いているものはマクロであり、型・引数を書いているものは関数である。なお、ヘッダ内で返却値の型や引数の取り方が同じである関数については冒頭にそれを断った上で関数名のみを記している (`ctype.h` や `math.h` など)。また、便宜上、C89 の部分と C95/C99 の部分は分けて書き、C95/C99 で追加された関数やヘッダは C89 の後に追加した。

A.1 `assert.h`(プログラム診断)

- `NDEBUG`: 定義することで `assert()` を無効にする。
- `void assert(int expression)`: 実行時の条件チェック。`expression` が偽の際にファイル名と行番号を `stderr` に出力して `abort` する。

A.2 `ctype.h`(文字の分類)

本ヘッダで定義される関数はいずれも `int function(int argument)` の形である。以下、関数名のみ記している。また、主語「引数が」を省略している。

- `isalnum`: 半角英数字ならば真。
- `isalpha`: アルファベットならば真。
- `islower`: 英小文字ならば真。
- `isupper`: 英大文字ならば真。
- `isdigit`: 数字ならば真。
- `isxdigit`: 16 進数ならば真。
- `iscntrl`: 制御文字ならば真。
- `ispunct`: 区切り文字ならば真。
- `isspace`: 空白文字ならば真。
- `isgraph`: スペース以外の印字可能文字ならば真。
- `isprint`: スペースを含む印字可能文字ならば真。
- `tolower`: 英大文字であるならば対応する英小文字を返し、それ以外の場合は引数の値を返す。
- `toupper`: 英小文字であるならば対応する英大文字を返し、それ以外の場合は引数の値を返す。

以下は、C99 において追加された関数である。

- `isblank`: 行中空白ならば真。

A.3 `errno.h`(エラー)

- `EDOM`: 定義域エラー
- `ERANGE`: 値域エラー
- `errno`: エラー状態を保持する外部変数

A.4 `float.h`(浮動小数点数型属性検査)

いずれも環境依存のマクロである。

- `DBL_DIG`: `double` 型変数が 10 進数で表すことのできる精度の桁数
- `DBL_EPSILON`: `double` 型変数でのマシンイプシロンの値
- `DBL_MANT_DIG`: `double` 型変数の仮数部における基数 `FLT_RADIX` の桁数
- `DBL_MAX`: `double` 型変数の表現可能な最大値
- `DBL_MAX_10_EXP`: `double` 型変数の表現可能な指数最大値 (基数 10)
- `DBL_MAX_EXP`: `double` 型変数の表現可能な指数最大値 (基数 2)
- `DBL_MIN`: `double` 型変数の表現可能な正の最小値
- `DBL_MIN_10_EXP`: `double` 型変数の表現可能な指数最小値 (基数 10)

- DBL_MIN_EXP:double 型変数の表現可能な指数最小値 (基数 2)
- DBL_ROUNDSD:double 型変数の足し算に対する丸めモード
- FLT_DIG:float 型変数が 10 進数で表すことのできる精度の桁数
- FLT_EPSILON:float 型変数でのマシンイプシロンの値
- FLT_MANT_DIG:float 型変数の仮数部における基数 FLT_RADIX の桁数
- FLT_MAX:float 型変数の表現可能な最大値
- FLT_MAX_10_EXP:float 型変数の表現可能な指数最大値 (基数 10)
- FLT_MAX_EXP:float 型変数の表現可能な指数最大値 (基数 2)
- FLT_MIN:float 型変数の表現可能な正の最小値
- FLT_MIN_10_EXP:float 型変数の表現可能な指数最小値 (基数 10)
- FLT_MIN_EXP:float 型変数の表現可能な指数最小値 (基数 2)
- FLT_RADIX:float 型変数の指数表現の基数
- FLT_ROUNDSD:float 型変数の足し算に対する丸めモード
- LDBL_DIG:long double 型変数が 10 進数で表すことのできる精度の桁数
- LDBL_EPSILON:long double 型変数でのマシンイプシロンの値
- LDBL_MANT_DIG:long double 型変数の仮数部における基数 FLT_RADIX の桁数
- LDBL_MAX:long double 型変数の表現可能な最大値
- LDBL_MAX_10_EXP:long double 型変数の表現可能な指数最大値 (基数 10)
- LDBL_MAX_EXP:long double 型変数の表現可能な指数最大値 (基数 2)
- LDBL_MIN:long double 型変数の表現可能な正の最小値
- LDBL_MIN_10_EXP:long double 型変数の表現可能な指数最小値 (基数 10)
- LDBL_MIN_EXP:long double 型変数の表現可能な指数最小値 (基数 2)
- LDBL_ROUNDSD:long double 型変数の足し算に対する丸めモード

以下は C99 において追加されたマクロである。

- DECIMAL_DIG:—:浮動小数点数型で表現できる最大の 10 進桁数
- FLT_EVAL_METHOD:実際に浮動小数点数演算を行うときの範囲と精度を示す値

A.5 limits.h(整数型属性検査)

いずれも環境依存のマクロである。

- CHAR_BIT:char 型変数のビット数
- CHAR_MAX:char 型変数の表現可能な最大値
- CHAR_MIN:char 型変数の表現可能な最小値
- INT_MAX:int 型変数の表現可能な最大値
- INT_MIN:int 型変数の表現可能な最小値
- LONG_MAX:long 型変数の表現可能な最大値
- LONG_MIN:long 型変数の表現可能な最小値
- MB_LEN_MAX:多バイト文字の最大バイト数
- SCHAR_MAX:signed char 型変数の表現可能な最大値
- SCHAR_MIN:signed char 型変数の表現可能な最小値
- SHRT_MAX:short 型変数の表現可能な最大値
- SHRT_MIN:short 型変数の表現可能な最小値
- UCHAR_MAX:unsigned char 型変数の表現可能な最大値
- UINT_MAX:unsigned int 型変数の表現可能な最大値
- ULONG_MAX:unsigned long 型変数の表現可能な最大値
- USHRT_MAX:unsigned short 型変数の表現可能な最大値

以下は、C99 において追加されたマクロである。

- LLONG_MAX:long long 型変数の表現可能な最大値
- LLONG_MIN:long long 型変数の表現可能な最小値
- ULLONG_MAX:unsigned long long 型変数の表現可能な最大値

A.6 locale.h(地域情報管理)

- LC_ALL:全ての地域情報に対する検索・設定用定数
- LC_COLLATE:地域固有の文字の比較順序情報に対する検索・設定用定数
- LC_CTYPE:地域固有文字(多バイト文字等)に対する検索・設定用定数
- LC_MONETARY:地域固有の通貨文字に対する検索・設定用定数
- LC_NUMERIC:地域固有の小数点文字に対する検索・設定用定数
- LC_TIME:地域固有の時間表現文字列に対する検索・設定用定数
- NULL:NULL ポインタ
- struct lconv:地域固有の表現情報を格納する構造体
- char *setlocale(int category, const char *locale):地域情報を設定する。
locale が NULL の時は地域情報を検索する。
- struct lconv *localeconv(void):地域情報が格納された構造体 lconv へのポインタを返す。

A.7 math.h(数学関数)

gccでコンパイルする際には-lm オプションをつけ `gcc source.c -lm` とする必要がある。
以下特に断らない限り、型が省略されたものを double 型とする。

- HUGE_VAL:double 型変数の表現可能な最大値(パソコンにおける「十分大きい値」)。
C99 ではこれに F, L をつけたものもある。
- acos(x):逆余弦を返す。 $\cos^{-1}(x)$
- asin(x):逆正弦を返す。 $\sin^{-1}(x)$
- atan(x):逆正接を返す。 $\tan^{-1}(x)$
- atan2(y,x):点 (x, y) の方向角を返す。
 $\tan^{-1}(y/x)$ 。
- cos(x):余弦を返す。 $\cos x$
- sin(x):正弦を返す。 $\sin x$
- tan(x):正接を返す。 $\tan x$
- cosh(x):双曲余弦を返す。 $\cosh x$
- sinh(x):双曲正弦を返す。 $\sinh x$
- tanh(x):双曲正接を返す。 $\tanh x$
- exp(x):引数に対するネイピア数 $e = 2.718281828\dots$ を底とする指数関数の値を返す。 $e^x = \exp x$
- frexp(value, int *exp):value を $x \times 2^t$ の形に表し(但し $0.5 \leq x < 1$)、 t を *exp に格納した後、 x を返す。
- ldexp(x, int exp): $x \times 2^{\text{exp}}$ を返す。

- `log(x)`:自然対数を返す。 $\ln x = \log_e x$
- `log10(x)`:常用対数を返す。 $\log_{10} x$
- `modf(value, double* iptr)`:`value` の整数部を*`iptr` に格納したあと、`value` の小数部の値を返す。
- `pow(x,y)`: $x^y = \exp_x y$ の値を返す。
- `sqrt(x)`:正平方根値 \sqrt{x} を返す。
- `ceil(x)`:天井関数値 (小数点以下切り上げ値) を返す。 $\lceil x \rceil = [x] + 1$
- `floor(x)`:床関数値 (小数点以下切り捨て値) を返す。 $\lfloor x \rfloor = [x]$
- `fabs(x)`:絶対値を返す。 $|x|$
- `fmod(x,y)`: x の y による剰余を返す。

C99 では大幅に関数・マクロ等が増えた。以下、C99 で追加された分である。

- `INFINITY`:正または符号無し無限大を示す定数マクロ
- `NAN`:浮動小数点数型の NAN を示す定数マクロ
- `FP_INFINITE`:正または負無限大を表すマクロ
- `FP_NAN`:NAN を示すマクロ
- `FP_NORMAL`:正規化数 (正常に表される浮動小数点数) を示すマクロ
- `FP_SUBNORMAL`:非正規化数 (値が小さすぎて正常に表されない浮動小数点数) を示すマクロ
- `FP_ZERO`:0 を表すマクロ
- `FP_FAST_FMA`:`fma` 関数が有実であることを示すマクロで、`F`,`L` をつけたものもある (それぞれ `fmaf`, `fmal` に対応)
- `FP_ILOGB0`:`ilogb(0)` の返却値を示すマクロ
- `FP_ILOGBNAN`:`ilogb(NAN)` の返却値を示すマクロ
- `MATH_ERRNO`:整数定数 1 に展開されるマクロ
- `MATH_ERREXCEPT`:整数定数 2 に展開されるマクロ
- `math_errhandling`:`MATH_ERRNO`、`MATH_ERREXCEPT` もしくはこの二つのビット毎論理和に展開されるマクロ
- `fpclassify(x)`:引数の値をカテゴリに分類する関数マクロ
- `isfinite(x)`:引数の値が有限の値かどうかを判定する関数マクロ

- `isinf(x)`: 引数の値が無限大かどうかを判定する関数マクロ
- `isnan`: 引数の値が NaN (非数) かどうかを判定する関数マクロ
- `isnormal(x)`: 引数の値が正規化数かどうかを判定する関数マクロ
- `signbit(x)`: 引数の符号が負かどうかを判定する関数マクロ
- `isgreater(x,y)`: x が y より大きいかどうかを判定する関数マクロ
- `isgreaterequal(x,y)`: x が y より大きい、または等しいかどうかを判定する関数マクロ
- `isless(x,y)`: x が y より小さいかどうかを判定する関数マクロ
- `islessequal(x,y)`: x が y より小さい、または等しいかどうかを判定する関数マクロ
- `islessgreater(x,y)`: x が y より小さい、または大きいかどうかを判定する関数マクロ
- `isunordered(x,y)`: 引数が順序付け可能かどうかを判定する関数マクロ
- `acosh(x)`: 逆双曲線余弦値を返す。 $\cosh^{-1}(x)$
- `asinh(x)`: 逆双曲線正弦値を返す。 $\sinh^{-1}(x)$
- `atanh(x)`: 逆双曲線正接値を返す。 $\tanh^{-1}(x)$
- `exp2(x)`: 引数に対する 2 を底とする指数関数の値を返す。 $2^x = \exp_2 x$
- `expm1(x)` 引数に対して $e^x - 1$ の値を返す。
- `log2(x)`: 二進対数を返す。 $\lg x = \log_2 x$
- `log1p(x)`: 引数に対して $\ln(x + 1)$ を返す。
- `logb(x)`: 引数の浮動小数点数表現における指数部分を浮動小数点数形式の符号付き整数として返す。
- `int ilogb(x)`: 引数の浮動小数点数表現における指数部分を浮動小数点数形式の符号付き整数として返す。
- `scalbn(x, int n)`: $x \times \text{FLT_RADIX}^n$ を効率よく計算する。FLT_RADIX については `float.h` 参照。 `scalbln` という名前に変えたら、`n` が long 型になる。
- `cbrt(x)`: 3 乗根を返す。 $\sqrt[3]{x}$
- `hypot(x,y)`: 過度のオーバーフローやアンダーフローを避けて入力点の原点からの距離を計算する。 $\sqrt{x^2 + y^2}$
- `erf(x)`: 誤差関数値を返す。 $\frac{2}{\sqrt{\pi}} \int_0^x \exp(-t^2) dt$

- `erfc(x)`: 余誤差関数値を返す。 $\frac{2}{\sqrt{\pi}} \int_x^\infty \exp(-t^2) dt$
- `tgamma(x)`: ガンマ関数値を返す。 $\Gamma(x) = \int_0^\infty t^{x-1} e^{-t} dt$
- `lgamma(x)`: 引数に対して $\ln |\Gamma(x)|$ を返す。
- `nearbyint(x)`: 引数を現在の丸め方向にしたがって浮動小数点数形式の整数値に丸める。例外は発生しない。
- `rint(x)`: 引数を現在の丸め方向にしたがって浮動小数点数形式の整数値に丸める。例外が発生することがある。
- `long lrint(x)`: 引数を現在の丸め方向にしたがって `long` 型整数値に丸める。`llrint` とすれば、`long long` 型に丸める。
- `round(x)`: 引数を四捨五入する。
- `long lround(x)`: 引数を四捨五入して `long` 型にする。`llround` とすれば、`long long` 型にする。
- `trunc(x)`: 絶対値の小数点以下を切り捨てて返す。
- `remainder(x,y)`: IEEE60559 規格に則った剰余を返す。`remquo` も同様。
- `copysign(x,y)`: x の絶対値と y の符号を持った値を返す。 $\frac{y}{|y|} |x|$
- `nan(const char *p)`: 文字列を NaN に変換する。
- `nextafter(x,y)`: y 方向に見たとき、 x の次に表現可能な数値を返却する。`nexttoward` 関数はこの第 2 引数が `long double` になったもの。
- `fmax(x,y)`: 二つの引数のうち大きい方の値を返す。
- `fmin(x,y)`: 二つの引数のうち小さい方の値を返す。
- `fdim(x,y)`: $x > y$ なら $x - y$ を、そうでなければ 0 を返す。
- `fma(x,y,z)`: $x \times y + z$ を返却する。

また、ここには記さないが、C99 では上記に挙げた関数の関数名終端に `f` を付すと関数・引数とも `float` 型に、`l` を付すと `long double` 型になる (`double` 以外で特記しているものを除く)。C99 では `math.h` も `complex.h` も含め、`tgmath.h` をインクルードすることで `math.h` の `double` 型関数名で関数が見えるようになる。

A.8 setjmp.h(非局所分岐)

- `jmp_buf`: `longjmp` 時に必要な環境を格納するための配列型
- `void longjmp(jmp_buf env, int val)`: `env` に格納した環境を復元して `setjmp` の位置に戻る (第 2 引数は `setjmp` の返却値)。
- `int setjmp(jmp_buf env)`: `longjmp` 時のための環境を `env` に格納する。

A.9 signal.h(シグナル処理)

- `sig_atomic_t`:シグナルを保持する変数の型
- `SIG_DFL`:シグナルに対して処理系のデフォルトの処理を指示するマクロ
- `SIG_ERR`:シグナルの設定失敗を示すマクロ
- `SIG_IGN`:シグナルを無視することを示すマクロ
- `SIGABRT`:異常終了
- `SIGINT`:非同期割り込み
- `SIGFPE`:算術演算エラー
- `SIGSEGV`:メモリの不正アクセス
- `SIGILL`:不正命令
- `SIGTERM`:プログラム終了
- `(*signal(int sig,void(*func)(int)))(int)`:シグナル (`sig`) が発生した際の処理関数設定。返却値は成功時その関数へのポインタ、失敗時 `SIG_ERR`。
- `int raise(int sig)`:シグナル (`sig`) を発生させる。成功時は0,失敗時は非0を返す。

A.10 stdarg.h(可変引数)

- `va_list`:可変引数リストを扱う変数の型
- `type va_arg(va_list ap,type)`:可変引数リスト `ap` より `type` で示す引数を返す。
- `void va_end(va_list ap)`:可変引数リスト `ap` を処理して終了させる。
- `void va_start(va_list ap,paramN)`:`paramN` の次から始まる可変引数リスト `ap` の初期化を行う。

以下は C99 で追加された関数である。

- `void va_copy(va_list dst, va_list src)`:`va_start` で `va_list` を初期化したオブジェクト `src` のコピー `dst` を作成する。

A.11 stddef.h(共通定義)

- `NULL`:NULL ポインタ
- `ptrdiff_t`:2つのポインタの差を表現する型
- `size_t`:`sizeof` 演算子の返す符号なし整数型
- `wchar_t`:wide 文字の型
- `offsetof(type,member-designator)`:構造体 `type` 内での構造体メンバ `member-designator` のオフセットを返す。

A.12 stdio.h(標準入出力)

以下、特に断らない限り、streamはFILE *streamを示し、ストリームポインタとする。

- FILE: ファイルストリーム格納変数型
- fpos_t: ファイル内の位置を表す型
- size_t: sizeof 演算子の返す符号なし整数型
- _IOFBF: フルバッファリングでストリーム入出力を行う (setvbuf の mode)
- _IOLBF: ラインバッファリングでストリーム入出力を行う (setvbuf の mode)
- _IONBF: バッファリングせずにストリーム入出力を行う (setvbuf の mode)
- BUFSIZ: setbuf で使用するバッファサイズ
- EOF: ファイル終端
- FILENAME_MAX: ファイル名の最大長
- FOPEN_MAX: 同時オープン可能ファイル個数
- L_tmpnam: tmpnam でつくられる一時ファイル名の最大長
- TMP_MAX: tmpnam で作ることができる一時ファイルの最大個数
- NULL: NULL ポインタ
- SEEK_SET: ファイルの先頭
- SEEK_CUR: 現在のファイル位置指示
- stdin: 標準入力ストリーム
- SEEK_END: ファイルの最後
- stdout: 標準出力ストリーム
- stderr: 標準エラー出力ストリーム
- int remove(const char *fn): fn の示すファイルを削除する。
- int rename(const char *old, const char *new): old で示すファイル名を new で示すファイル名に変更する。
- FILE *tmpfile(void): 一時ファイルを作成してそのファイルストリームを返す。
- char *tmpnam(char *s): 一意な一時ファイル名を生成して s に格納し、その名前を関数値として返す。
- int fclose(stream): 引数のストリームをクローズする。
- int fflush(stream): 引数のストリームをフラッシュする。
- FILE *fopen(const char *fn, const char *mode): mode に従って fn の示すファイルをオープンしてストリームを返す。

- `FILE *freopen(const char *fn, const char *mode, stream)`: mode に従って fn の示すファイルを再オープンして stream にストリームを返す。
- `void setbuf(stream, char *buf)`: stream のバッファリングをバッファ buf を用いて行う。
- `int setvbuf(stream, char *buf, int mode, size_t size)`: stream のバッファリングをバッファ buf を用いて mode, size に従って行う。
- `int fprintf(stream, const char *format, ...)`: format に従い stream にデータを出力する。
- `int fscanf(stream, const char *format, ...)`: format に従い stream からデータを入力する。
- `int printf(const char *format, ...)`: format に従い stdout にデータを出力する。
- `int scanf(const char *format, ...)`: format に従い stdin からデータを入力する。
- `int sprintf(char *s, const char *format, ...)`: format に従ってデータを文字列 s に出力する。
- `int sscanf(char *s, const char *format, ...)`: format に従ってデータを文字列 s から入力する。
- `int vfprintf(stream, const char *format, va_list arg)`: format に従い stream に可変引数リスト arg の内容を出力する。
- `int vprintf(const char *format, va_list arg)`: format に従い stdout に可変引数リスト arg の内容を出力する。
- `int vsprintf(char *s, const char *format, va_list arg)`: format に従い文字列 s に可変引数リスト arg の内容を出力する。
- `int fgetc(stream)`: stream より 1 文字入力してその文字を返す。
- `char *fgets(char *s, int n, stream)`: stream より n 文字入力して s に格納し、s へのポインタを返す。
- `int fputc(int c, stream)`: c を stream に出力する。
- `int fputs(const char *s, stream)`: s を stream に出力する。
- `int getc(stream)`: stream より 1 文字入力してその文字を返す。
- `int getchar(void)`: stdin より 1 文字入力してその文字を返す。

- `char *gets(char *s)`:`stdin`より1行入力して `s` に格納し、`s` へのポインタを返す (改行コードが `NULL` 文字に置き換えられる)。
- `int putc(int c, stream)`: `c` を `stream` に出力する。
- `int putchar(int c)`: `stdout` に `c` を出力して `c` を返す。
- `int puts(const char *s)`: `s` を `stdout` に出力する。(`NULL` 文字が抜かれて改行コードが補われる。)
- `int ungetc(int c, stream)`: `stream` に `c` を戻し、`c` を返す。
- `size_t fread(void *ptr, size_t size, size_t nmemb, stream)`:
 `stream` から `size` 分のデータ `nmemb` 個を入力して `ptr` にセットする。
- `size_t fwrite(const void *ptr, size_t size, size_t nmemb, stream)`:
 `stream` に `size` 分のデータ `nmemb` 個を `ptr` から出力する。
- `int fgetpos(stream, fpos_t *pos)`: `stream` のファイル位置指示子の値を求めて `pos` にセットする。
- `int fseek(stream, long int offset, int whence)`: `stream` に対しファイル位置 `whence` と移動量 `offset` を変更する。
- `int fsetpos(stream, fpos_t *pos)`: `stream` のファイル位置を `pos` に変更する。
- `long int ftell(stream)`: 引数のファイルポジションインジケータの値を返す。
- `void rewind(stream)`: 引数のファイルポジションインジケータをファイル先頭にセットする。
- `void clearerr(stream)`: 引数の EOF 及びエラー指示子をクリアする。
- `int feof(stream)`: 引数のファイル終端指示子を調べ、EOF ならば非0を返す。
- `int ferror(stream)`: 引数のエラー状態指示子を調べ、エラーならば非0を返す。
- `void perror(const char *s)`: `errno` に対応するエラーメッセージを `stderr` に出力する。

以下は C99 において追加された関数である。

- `int snprintf(char *s, size_t n, const char *format, ...)`: `format` に従ってデータを `n` 文字分、文字列 `s` に出力する。返却値は出力文字数。
- `int vsnprintf(char *s, size_t n, const char *format, va_list arg)`:
 `format` に従い文字列 `s` に `n` 文字分可変引数リスト `arg` の内容を出力する。返却値は出力文字数。
- `int vscanf(const char *format, va_list arg)`: `scanf` の可変個数引数部分を可変引数リスト `va_list` に変更したもの。

- `int vsscanf(const char *s, const char *format, va_list arg):sscanf` の可変個数引数部分を `va_list` に変更したもの。
- `int vfscanf(stream, const char *format, va_list arg):fscanf` の可変個数引数部分を `va_list` に変更したもの。

A.13 stdlib.h(ユーティリティ)

以下特に断らない限り、`n` 及び `size` は `size_t` 型とし、`str` は `const char *` 型とする。

- `div_t:div()` の返す型
- `ldiv_t:ldiv()` の返す型
- `size_t:sizeof` 演算子の返す符号なし整数型
- `wchar_t:wide` 文字の型
- `EXIT_FAILURE`:プログラムの実行が失敗したことを示す
- `EXIT_SUCCESS`:プログラムの実行が成功したことを示す
- `MB_CUR_MAX`:その時点でのマルチバイト文字を表現するのに必要な最大バイト数
- `NULL:NULL` ポインタ
- `RAND_MAX`:疑似乱数の最大値
- `double atof(str):str` を浮動小数点数に変換して返す。
- `int atoi(str):str` を整数に変換して返す。
- `long int atol(str):str` を `long` 型整数にして返す。
- `double strtod(str, char **endptr):str` を浮動小数点数に変換し (先頭の空白文字はスキップされる)、浮動小数点より後ろの文字列へのポインタを `*endptr` に格納する。返却値は変換された浮動小数点数。
- `long int strtol(str, char **endptr, int base):str` を `base` によって指定された記数法にしたがって `long` 型に変換し (先頭の空白文字はスキップされる)、その数値より後ろの文字列へのポインタを `*endptr` に格納する。返却値は変換された数。
- `unsigned long int strtoul(str, char **endptr, int base):str` を `base` によって指定された記数法にしたがって `unsigned long` 型に変換し (先頭の空白文字はスキップされる)、その数値より後ろの文字列へのポインタを `*endptr` に格納する。返却値は変換された数。
- `int rand(void):0` から `RAND_MAX` の範囲で疑似乱数を発生させ、その値を返す。
- `void srand(unsigned int seed):` 引数を疑似乱数発生ルーチンの種として与える。

- `void *calloc(n,size)`:size バイト n 個分の動的メモリを割り当て、それを 0 クリアする。
- `void free(void *ptr)`:引数のポインタで与えられた動的メモリ領域を解放する。
- `void *malloc(size)`:size バイト分の動的メモリを割り当てる。
- `void *realloc(void *ptr,size)`:ptr で示される動的メモリを size バイト分の大きさに再割り当てする。
- `void abort(void)`:プログラムを異常終了する。
- `int atexit(void (*func)(void))`:プログラム終了時に実行する関数を登録する。
- `void exit(int status)`:引数を返却値としてプログラムを終了する。
- `char *getenv(str)`:環境変数 str に対応する文字列へのポインタを返す。
- `int system(str)`:str に示されるプログラム (コマンドライン形式) を実行する。
- `void *bsearch(const void *key,const void *base,n,size,
int (*compare)(const void *, const void *))`
:配列 (base、要素毎のサイズ size、要素数 n) 内のデータ中のキー key に一致するデータをバイナリサーチで検索し、その要素へのポインタを返す。比較は compare で示される比較関数によって行う。
- `void qsort(void *base,n,size,
int (*compare)(const void *, const void *))`
:配列 (base、要素毎のサイズ size、要素数 n) を compare で示される比較関数に従ってソートする。
- `int abs(int j)`:整数引数の絶対値を返す。
- `long int labs(long int j)`:long 型引数の絶対値を返す。
- `div_t div(int num,int denom)`:num/denom の商と剰余を div_t 型で返す。
- `ldiv_t ldiv(long int num,long int denom)`:num/denom の商と剰余を ldiv_t 型で返す。
- `int mblen(str,n)`:マルチバイト文字列 str を最大 n バイトまで検査して、次のマルチバイト文字のバイト数を返す。
- `int mbtowc(wchar_t *pwc,str,n)`:マルチバイト文字 str を最初から n バイトまで wide 文字 (pwc) に変換する。
- `int wctomb(char *s,wchar_t wchar)`:wide 文字 (wchar) をマルチバイト文字 s に変換する。

- `size_t mbstowcs(wchar_t *pwcs, str, n)`:マルチバイト文字列 `str` を最初から `n` バイトまで wide 文字列 (`pwcs`) に変換する。
- `size_t wcstombs(char *s, const wchar_t *pwcs, n)`:wide 文字列 (`pwcs`) を最初から `n` バイトまでマルチバイト文字列 `s` に変換する。

以下は C99 において追加された型/関数である。

- `lldiv_t`:`lldiv()` の返す型
- `void _Exit(int status)`:プログラムを正常終了する。
- `float strtodf(str, char **endp)`:`strtod` の float 版
- `long double strtold(str, char **endp)`:`strtod` の long double 版
- `long long int strtoll(str, char **endptr, int base)`:`strtol` の long long int 版
- `unsigned long long int strtoull(str, char **endptr, int base)`:`strtol` の unsigned long long int 版
- `long long int atoll(str)`:`atoi` の long long int 版
- `long long int llabs(long long int j)`:`abs` の long long int 版
- `lldiv_t lldiv(long long int num, long long int denom)`:`ldiv` の long long int 版

A.14 string.h(文字列操作)

以下特に断らない限り、`s1`,`s2`,`s` はいずれも `char *` 型とし、`m1`,`m2`,`m` は `void *` 型とする。但し、`const s2` 等と書いた場合は、`const char *` 型等、前に `const` を付すものとする。また、`n` は `size_t` 型とする。

- `size_t`:`sizeof` 演算子の返す符号なし整数型
- `NULL`:`NULL` ポインタ
- `void *memcpy(m1, const m2, n)`:`m2` を `m1` に `n` バイト分コピーする。
- `void *memmove(m1, const m2, n)`:`m2` を `m1` に `n` バイト分コピーする。(`m2` と `m1` が重なっても良い。)
- `int memcmp(const m1, const m2, n)`:`m1`,`m2` を `n` バイトまで比較し、一致したら 0, 一致しなければ非 0 を返す。
- `void *memchr(const m, int c, n)`:文字 `c` を `m` の最初の `n` バイト中から探し、あればその文字へのポインタを、なければ `NULL` を返す。

- `void *memset(m, int c, n)`: `m` を文字 `c` で `n` バイト分埋める。
- `char *strcpy(s1, const s2)`: `s1` に `s2` をコピーする。
- `char *strncpy(s1, const s2, n)`: `s1` に `s2` を最大 `n` バイトまでコピーする。
- `char *strcat(s1, const s2)`: `s1` の後ろに `s2` を連結する。
- `char *strncat(s1, const s2, n)`: `s1` の後ろに `s2` を最大 `n` バイトまで連結する。
- `int strcmp(const s1, const s2)`: `s1, s2` を比較し、一致したら 0, 一致しなければ非 0 を返す。
- `int strcoll(const s1, const s2)`: 地域情報を使い `s1, s2` を比較し、一致したら 0, 一致しなければ非 0 を返す。
- `int strncmp(const s1, const s2, n)`: `s1, s2` を `n` バイトまで比較し、一致したら 0, 一致しなければ非 0 を返す。
- `size_t strxfrm(s1, const s2, n)`: `s2` を地域情報にしたがって `s1` に最初の `n` バイトだけ変換する。
- `char *strchr(const s, int c)`: 文字 `c` を `s` 中から探し、あればその文字へのポインタを、なければ NULL を返す。
- `size_t strcspn(const s1, const s2)`: `s2` に含まれない文字だけで構成される文字列を `s1` から探し、その最初の部分の長さを返す。
- `char *strpbrk(const s1, const s2)`: `s2` 中の文字が `s1` に出てくる、その最初の文字へのポインタを返す。
- `char *strrchr(const s, int c)`: `s` 中で文字 `c` が現れる最後の位置のポインタを返す。
- `size_t strspn(const s1, const s2)`: `s2` に含まれる文字だけで構成される文字列を `s1` から探し、その最初の部分の長さを返す。
- `char *strstr(s)`: `s2` が `s1` に出てくる最初の文字へのポインタを返す。
- `char *strtok(s1, const s2)`: `s1` を区切り記号文字列 `s2` にしたがってトークンに分割する。`s1` は 2 回目以降の呼び出しにおいて NULL を指定する。トークンがあればそのポインタを、なければ NULL を返す。
- `char *strerror(int errnum)`: エラー番号 `errnum` を文字列に変換し、そのポインタを返す。
- `size_t strlen(const s)`: `s` の長さを返す。

A.15 time.h(時間)

- `clock_t:clock()` の返却値の型
- `time_t`:カレンダー時間の型
- `size_t:sizeof` 演算子の返す符号なし整数型
- `CLOCKS_PER_SEC:clock_t` における 1 秒間の数
- `NULL:NULL` ポインタ
- `clock_t clock(void)` プログラムの実行に要した経過時間を返す。
- `double difftime(time_t time1,time_t time2)`:`time1` と `time2` の差を秒で返す。
- `time_t mktime(struct tm *timeptr)`:ローカル時間 `timeptr` をカレンダー時間に変換して返す。
- `time_t time(time_t *timer)`:現在のカレンダー時間を `timer` にセットし、カレンダー時間を返す。
- `char *asctime(const struct tm *timeptr)`:ローカル時間 `timeptr` を文字列に変換して返す。
- `char *ctime(const time_t *timer)`:カレンダー時間 `timer` を文字列に変換して返す。
- `struct tm *gmtime(const time_t *timer)`:カレンダー時間 `timer` を世界標準時に変換して返す。
- `struct tm *localtime(const time_t *timer)`:カレンダー時間 `timer` をローカル時間に変換して返す。
- `size_t strftime(char *s,size_t max, const char *fm,const struct tm *tptr)`
:ローカル時間 `tptr` を表示形式 `fm` に従い、最大 `max` 文字まで変換し、`s` にセットする。

A.16 iso646.h(代替綴・C95)

本ヘッダは C95 において追加されたヘッダである。演算子をマクロを用いて記すためのヘッダであり、以下はいずれもマクロである。

- | | |
|---|---|
| • <code>and</code> :置き換える演算子は <code>&&</code> | • <code>bitand</code> :置き換える演算子は <code>&</code> |
| • <code>and_eq</code> :置き換える演算子は <code>&=</code> | • <code>bitor</code> :置き換える演算子は <code> </code> |

- `compl`:置き換える演算子は`~`
- `not`:置き換える演算子は`!`
- `not_eq`:置き換える演算子は`!=`
- `or`:置き換える演算子は`||`
- `or_eq`:置き換える演算子は`| =`
- `xor`:置き換える演算子は`^`
- `xor_eq`:置き換える演算子は`^=`

A.17 `wchar.h`(ワイド文字・C95)

本ヘッダは C95 において追加されたヘッダである。ひとまず、型とマクロを記す。

- `wchar_t`:wide 文字の型
- `wint_t`:`wchar_t` に加えて、拡張文字で表示されない値をひとつ以上示す広義整数型
- `mbstate_t`:マルチバイト文字からワイド文字への変換状態を示す型
- `size_t`:`sizeof` 演算子の返す符号なし整数型
- `NULL`:`NULL` ポインタ
- `WCHAR_MIN`:`wchar_t` 型の最小値
- `WCHAR_MAX`:`wchar_t` 型の最大値
- `WEOF`:ファイル終端を示す `wint_t` 型の値

次いで、他のヘッダの関数と直接には対応しない関数を示す。`c_(型)` は `const (型)` を示す。

- `fwide(FILE *stream, int mode)`:`stream` の入出力単位を `mode` が負の場合はバイト単位、正の場合はワイド文字単位に設定する。0 の場合は設定を変更しない。`mode` と同符号の値を返却する。
- `wint_t btowc(int c)`:引数の 1 バイト文字をワイド文字に変換して返す。
- `int wctob(wint_t c)`:引数のワイド文字を 1 バイト文字に変換して返す。
- `int mbsinit(c_mbstate_t *ps)`:引数の `mbstate_t` オブジェクトが初期変換状態を表すかどうかを判定し、表す場合は非 0、表さない場合は 0 を返す。
- `size_t mbrlen(c_char *s, size_t n, mbstate_t *ps)`:マルチバイト文字のバイト長を取得する。
- `size_t mbrtowc(wchar_t *c, c_char *s, size_t n, mbstate_t *ps)`:変換状態格納領域を `ps` とし、マルチバイト文字をワイド文字に変換する。
- `size_t wctomb(char *s, wchar_t wc, mbstate_t *ps)`:変換状態格納領域を `ps` とし、ワイド文字をマルチバイト文字に変換する。

- `size_t mbsrtowcs(wchar_t *p, c_char **s, size_t n, mbstate_t *ps)`: 変換状態格納領域を `ps` とし、`s` の示すマルチバイト文字列を `n` バイト分ワイド文字列に変換して `p` に格納する。返却値はエラーが出れば-1、処理成功時は変換に成功したヌル文字以外の文字の数である。
- `size_t wcsrtombs(char *s, c_wchar_t **src, size_t n, mbstate_t *ps)`: 変換状態格納領域を `ps` とし、`src` の示すワイド文字列を `n` バイト分マルチバイト文字列に変換して `s` に格納する。返却値はエラーが出れば-1、処理成功時は変換に成功したヌル文字以外の文字の数である。

以下に示す関数は `stdio.h`, `stdlib.h` 等の関数の `char` の部分を `wchar_t` に変更したものである。それ故、型や引数は省き、対応する関数を挙げておく。

- | | |
|---------------------------------------|-------------------------------------|
| ● <code>fwprintf:fprintf</code> に対応 | ● <code>wcstoul:strtol</code> に対応 |
| ● <code>fwscanf:fscanf</code> に対応 | ● <code>wscpy:strcpy</code> に対応 |
| ● <code>swprintf:snprintf</code> に対応 | ● <code>wcsncpy:strncpy</code> に対応 |
| ● <code>swscanf:sscanf</code> に対応 | ● <code>wmemcpy:memcpy</code> に対応 |
| ● <code>vfwprintf:vfprintf</code> に対応 | ● <code>wmemmove:memmove</code> に対応 |
| ● <code>vswprintf:vsprintf</code> に対応 | ● <code>wscat:strcat</code> に対応 |
| ● <code>vwprintf:vprintf</code> に対応 | ● <code>wcsncat:strncat</code> に対応 |
| ● <code>wprintf:printf</code> に対応 | ● <code>wscmp:strcmp</code> に対応 |
| ● <code>wscanf:scanf</code> に対応 | ● <code>wscoll:strcoll</code> に対応 |
| ● <code>fgetwc:fgetc</code> に対応 | ● <code>wcsncmp:strncmp</code> に対応 |
| ● <code>fgetws:fgets</code> に対応 | ● <code>wcsxfrm:strxfrm</code> に対応 |
| ● <code>fputwc:fputc</code> に対応 | ● <code>wmemcmp:memcmp</code> に対応 |
| ● <code>fputws:fputs</code> に対応 | ● <code>wcschr:strchr</code> に対応 |
| ● <code>getwc:getc</code> に対応 | ● <code>wcscspn:strcspn</code> に対応 |
| ● <code>getwchar:getchar</code> に対応 | ● <code>wcspbrk:strpbrk</code> に対応 |
| ● <code>putwc:putc</code> に対応 | ● <code>wcsrchr:strrchr</code> に対応 |
| ● <code>putwchar:putchar</code> に対応 | ● <code>wcsspn:strspn</code> に対応 |
| ● <code>ungetwc:ungetc</code> に対応 | ● <code>wcsstr:strstr</code> に対応 |
| ● <code>wctod:strtod</code> に対応 | ● <code>wctok:strtoken</code> に対応 |
| ● <code>wctol:strtol</code> に対応 | ● <code>wmemchr:memchr</code> に対応 |

- `wcslen:strlen` に対応
- `wcsftime:strftime` に対応
- `wmemset:memset` に対応

以下は、C99 において追加された関数である。

- `wcstoull:strtoull` に対応
- `vwscanf:vscanf` に対応
- `wcstoll:strtoll` に対応
- `vswscanf:vsscanf` に対応
- `wcstof:strtouf` に対応
- `vfwscanf:vfscanf` に対応
- `wcstold:strtold` に対応

A.18 `wctype.h`(ワイド文字変換・C95)

本ヘッダは C95 において追加されたヘッダである。まず、型とマクロを示す。

- `wctype_t`:ワイド文字の種別を表す型
- `wctrans_t`:あるワイド文字を他のワイド文字に変換できるマッピングを表現する型
- `wint_t:wchar_t` に加えて、拡張文字で表示されない値をひとつ以上示す広義整数型
- `WEOF`:ファイル終端を示す `wint_t` 型の値

他のヘッダの関数と対応のない関数は以下の通り。

- `int iswctype(wint_t wc, wctype_t desc)`:`wc` が `desc` に属するワイド文字か否か判定し、属する場合は非 0 を、属さない場合は 0 を返す。
- `wint_t towctrans(wint_t wc, wctrans_t desc)`:`desc` の変換に従って `wc` を変換して返却する。
- `wctrans_t wctrans(const char *p)`:`p` によって識別される変換の値を返却する。
- `wctype_t wctype(const char *p)`:`p` によって識別されるワイド文字の種別を返却する。

上記以外に対応する関数が `ctype.h` にある。対応する関数の引数の型を `wint_t` に変更したものが本ヘッダ内の関数である。

- `iswalnum:isalnum` に対応
- `iswlower:islower` に対応
- `iswalpha:isalpha` に対応
- `iswprint:isprint` に対応
- `iswcntrl:iscntrl` に対応
- `iswpunct:ispunct` に対応
- `iswdigit:isdigit` に対応
- `iswspace:isspace` に対応
- `iswgraph:isgraph` に対応
- `iswupper:isupper` に対応

- `iswxdigit:isxdigit` に対応
- `iswxdigit:isxdigit` に対応
- `towupper:toupper` に対応
- `towlower:tolower` に対応

以下は、C99 において追加された関数である。

- `iswblank:isblank` に対応

A.19 `complex.h`(複素数演算・C99)

本ヘッダは C99 において追加されたヘッダである。まず、マクロを掲載しておく。

- `complex`:型名 `_Complex` を表す
- `_Complex_I:const float _Complex` 型の虚数単位
- `imaginary`:型名 `_Imaginary` を表す
- `_Imaginary_I:const float _Imaginary` 型の虚数単位
- `I`:虚数単位 (型は環境依存)

本ヘッダ内の関数は `math.h` に対応した関数のあるものが多い。以下にそれを挙げておく。
なお、関数の型・引数の型とも、特記しないものは `double complex` 型とする。

- `cacos(z):acos` に対応
- `cacosh(z):acosh` に対応
- `casin(z):asin` に対応
- `casinh(z):asinh` に対応
- `catan(z):atan` に対応
- `catanh(z):atanh` に対応
- `ccos(z):cos` に対応
- `cexp(z):exp` に対応
- `csin(z):sin` に対応
- `clog(z):log` に対応
- `ctan(z):tan` に対応
- `cpow(x,y):pow` に対応
- `ccosh(z):cosh` に対応
- `csqrt(z):sqrt` に対応
- `csinh(z):sinh` に対応
- `double cabs(z):fabs` に対応
- `ctanh(z):tanh` に対応

以下は複素数特有の関数である。

- `conj(z)`:共役複素数を返す。 \bar{z}
- `double cart(z)`:仰角を返す。
- `cproj(z)`:リーマン球面上への射影値を返す。
- `double creal(z)`:実部を返す。
- `double cimag(z)`:虚部を返す。

本ヘッダも `math.h` 同様、上記に挙げた関数の関数名終端に `f` を付すと関数・引数とも `float complex` 型に、`l` を付すと `long double complex` 型になる (`double` 以外で特記しているものを除く)。また、`math.h` も `complex.h` も含め、`tgmath.h` をインクルードすることで `math.h` の `double` 型関数名 (複素数に特有の関数は `complex.h` の `double complex` 型関数名) で関数が使えるようになる。

A.20 `fenv.h`(浮動小数点数環境・C99)

本ヘッダは C99 において追加されたヘッダである。

- `fenv_t`:現在の浮動小数点数環境情報を格納する型
- `fexcept_t`:例外に関する情報を格納する型
- `FE_DIVBYZERO`:ゼロ除算例外
- `FE_DOWNWARD`: $-\infty$ の方向へ丸める
- `FE_INEXACT`:不正確例外
- `FE_TONEAREST`:最も近い値へ丸める
- `FE_INVALID`:不正操作例外
- `FE_TOWARDZERO`:0 方向へ丸める
- `FE_OVERFLOW`:オーバーフロー例外
- `FE_UPWARD`: $+\infty$ の方向へ丸める
- `FE_UNDERFLOW`:アンダーフロー例外
- `FE_DFL_ENV`:デフォルトの浮動小数点数環境
- `FE_ALL_EXCEPT`:処理系定義の全例外
- `int fegetexceptflag(fexcept_t *f, int e)`:`e` で指定された例外フラグを実装定義依存の値として `f` に格納する。もし格納に成功した場合には 0 を、失敗した場合には、非 0 を返す。
- `int fesetexceptflag(const fexcept_t *f, int e)`:`f` で指定されたオブジェクトでの表現に従い、`e` で指定された例外フラグに対する完全な状態をセットする。`e` が 0 か状態の設定に成功した場合には 0 を返す。それ以外は非 0 を返す。この関数ではフラグの状態をセットするだけで例外は発生しない。
- `int feclearexcept(int e)`:`e` で指定された例外フラグのクリアを試みる。`e` が 0 かクリアが成功した場合に 0 を返す。それ以外は非 0 を返す。
- `int fetestexcept(int e)`:`e` で指定された例外フラグが現在セットされているか検査し、セットされている例外フラグ値を返す。
- `int feraiseexcept(int e)`:`e` で指定された例外を発生させる。`e` が 0 か例外の発生に成功した場合は 0 を返す。それ以外は非 0 を返す。
- `int fegetround(void)`:現在の丸めモードを丸めフラグの値で返す。
- `int fesetround(int r)`:`r` で指定された丸めモードにセットする。引数が丸めフラグの値でないときは状態は変更されない。引数で指定された丸め方向に設定できたときに限り 0 を返す。

- `int fegetenv(fenv_t *e)`:現在の浮動小数点数環境を `e` に格納する。成功すると 0 を返す。
- `int fesetenv(const fenv_t *e)`:現在の浮動小数点数環境に指定された浮動小数点数環境 `e` を設定する。設定に成功すると 0 を返す。
- `int feholdexcept(fenv_t *e)`:現在の浮動小数点数環境を `e` に格納し、例外フラグをクリアし、可能であれば全例外に対し (例外でも継続する) 非停止モードに設定する。もし非停止モードに設定できたときは 0 を返す。
- `int feupdateenv(const fenv_t *e)`:現在発生した例外を一時領域に格納し、`e` で指定された浮動小数点数環境を設定した後、一時領域に格納した例外を発生させる。例外発生に成功した場合には 0 を返す。

A.21 `inttypes.h`(整数型書式・C99)

本ヘッダは C99 において追加されたヘッダである。書式設定用マクロについては本文参照。ここでは対応関数を示すに留める。

- | | |
|--|---|
| ● <code>imaxabs</code> : <code>abs</code> の <code>intmax_t</code> 版 | ● <code>strtoumax</code> : <code>strtol</code> の <code>uintmax_t</code> 版 |
| ● <code>imaxdiv</code> : <code>div</code> の <code>intmax_t</code> 版 | ● <code>wcstoimax</code> : <code>wcstol</code> の <code>intmax_t</code> 版 |
| ● <code>strtoimax</code> : <code>strtol</code> の <code>intmax_t</code> 版 | ● <code>wcstoumax</code> : <code>wcstol</code> の <code>uintmax_t</code> 版 |

A.22 `stdbool.h`(論理・C99)

本ヘッダは C99 において追加されたヘッダである。マクロのみのヘッダである。

- `bool`:`_Bool` 型を示す
- `true`:整数定数 1 に展開する
- `false`:整数定数 0 に展開する
- `__bool_true_false_are_defined`:整数定数 1 に展開する

A.23 `stdint.h`(整数型管理・C99)

本ヘッダは C99 において追加されたヘッダである。マクロのみのヘッダである。なお、マクロ中の `N` には通常、8,16,32,64 の何れかが入り、これが幅指定となる。

- `INTN_MIN`:幅指定符号付き整数型の最小値
- `INTN_MAX`:幅指定符号付き整数型の最大値
- `UINTN_MAX`:幅指定符号無し整数型の最大値

- INT_LEASTN_MIN:最小幅指定符号付き整数型の最小値
- INT_LEASTN_MAX:最小幅指定符号付き整数型の最大値
- UINT_LEASTN_MAX:最小幅指定符号無し整数型の最大値
- INT_FASTN_MIN:最速最小幅指定符号付き整数型の最小値
- INT_FASTN_MAX:最速最小幅指定符号付き整数型の最大値
- UINT_FASTN_MAX:最速最小幅指定符号無し整数型の最大値
- INTPTR_MIN:ポインタ保持可能な符号付き整数型の最小値
- INTPTR_MAX:ポインタ保持可能な符号付き整数型の最大値
- UINTPTR_MAX:ポインタ保持可能な符号付き整数型の最大値
- INTMAX_MIN:最大幅符号付き整数型の最小値
- INTMAX_MAX:最大幅符号付き整数型の最大値
- UINTMAX_MAX:最大幅符号無し整数型の最大値
- PTRDIFF_MIN:ptrdiff_t の限界値下限
- PTRDIFF_MAX:ptrdiff_t の限界値上限
- SIG_ATOMIC_MIN:sig_atomic_t の限界値下限
- SIG_ATOMIC_MAX:sig_atomic_t の限界値上限
- SIZE_MAX:size_t の限界値
- WCHAR_MIN:wchar_t の限界値下限
- WCHAR_MAX:wchar_t の限界値上限
- WINT_MIN:wint_t の限界値下限
- WINT_MAX:wint_t の限界値上限
- INTN_C(値):値を int_leastN_t に対応する整数定数式に展開
- UINTN_C(値):値を uint_leastN_t に対応する整数定数式に展開
- INTMAX_C(値):値を intmax_t である整数定数式に展開
- UINTMAX_C(値):値を uintmax_t である整数定数式に展開

A.24 `tgmath.h`(型総称数学関数・C99)

本ヘッダはC99において追加されたヘッダである。このヘッダの中の関数マクロは`math.h`あるいは`complex.h`の`double`の関数と同名である。引数の型によって関数を使い分けるのがこのヘッダの目的である。詳細は本文を参照されたい。

付録B 一部解説の付記

この付録では、本文で触れなかった事項について、簡潔に補足解説を加える。

B.1 ASCII文字コード一覧

C言語でよく用いられる文字コードであるASCIIコードの一覧を示す。なお、特殊文字についてはその意味を記した。

なお、C言語でこの一覧を出力したい場合は、次のコードを実行すれば良い。

```
for(i=0;i<128;i++) printf("%d\t%x\t%c\n",i,i,(char)i);
```

10進	16進	文字	10進	16進	文字
0	0x00	NUL(null文字)	24	0x18	CAN(とりけし)
1	0x01	SOH(ヘッダ開始)	25	0x19	EM(メディア終了)
2	0x02	STX(テキスト開始)	26	0x1a	SUB(置換)
3	0x03	ETX(テキスト終了)	27	0x1b	ESC(エスケープ)
4	0x04	EOT(転送終了)	28	0x1c	FS(フォーム区切り)
5	0x05	ENQ(照会)	29	0x1d	GS(グループ区切り)
6	0x06	ACK(受信OK)	30	0x1e	RS(レコード区切り)
7	0x07	BEL(警告)	31	0x1f	US(ユニット区切り)
8	0x08	BS(後退)	32	0x20	(スペース)
9	0x09	HT(水平タブ)	33	0x21	!
10	0x0a	LF(改行)	34	0x22	"
11	0x0b	VT(垂直タブ)	35	0x23	#
12	0x0c	FF(改頁)	36	0x24	\$
13	0x0d	CR(復帰)	37	0x25	%
14	0x0e	SO(シフトアウト)	38	0x26	&
15	0x0f	SI(シフトイン)	39	0x27	'
16	0x10	DLE(データリンクエスケープ)	40	0x28	(
17	0x11	DC1(装置制御1)	41	0x29)
18	0x12	DC2(装置制御2)	42	0x2a	*
19	0x13	DC3(装置制御3)	43	0x2b	+
20	0x14	DC4(装置制御4)	44	0x2c	,
21	0x15	NAK(受信失敗)	45	0x2d	-
22	0x16	SYN(同期)	46	0x2e	.
23	0x17	ETB(転送ブロック終了)	47	0x2f	/

10 進	16 進	文字	10 進	16 進	文字	10 進	16 進	文字	10 進	16 進	文字
48	0x30	0	68	0x44	D	88	0x58	X	108	0x6c	l
49	0x31	1	69	0x45	E	89	0x59	Y	109	0x6d	m
50	0x32	2	70	0x46	F	90	0x5a	Z	110	0x6e	n
51	0x33	3	71	0x47	G	91	0x5b	[111	0x6f	o
52	0x34	4	72	0x48	H	92	0x5c	\	112	0x70	p
53	0x35	5	73	0x49	I	93	0x5d]	113	0x71	q
54	0x36	6	74	0x4a	J	94	0x5e	^	114	0x72	r
55	0x37	7	75	0x4b	K	95	0x5f	_	115	0x73	s
56	0x38	8	76	0x4c	L	96	0x60	'	116	0x74	t
57	0x39	9	77	0x4d	M	97	0x61	a	117	0x75	u
58	0x3a	:	78	0x4e	N	98	0x62	b	118	0x76	v
59	0x3b	;	79	0x4f	O	99	0x63	c	119	0x77	w
60	0x3c	<	80	0x50	P	100	0x64	d	120	0x78	x
61	0x3d	=	81	0x51	Q	101	0x65	e	121	0x79	y
62	0x3e	>	82	0x52	R	102	0x66	f	122	0x7a	z
63	0x3f	?	83	0x53	S	103	0x67	g	123	0x7b	{
64	0x40	@	84	0x54	T	104	0x68	h	124	0x7c	
65	0x41	A	85	0x55	U	105	0x69	i	125	0x7d	}
66	0x42	B	86	0x56	V	106	0x6a	j	126	0x7e	~
67	0x43	C	87	0x57	W	107	0x6b	k	127	0x7f	DEL(削除)

B.2 C 言語の予約語一覧

C 言語の予約語及びその意味を記す。

- void:型の無いことを宣言
- char:1 バイト・文字型
- short:2 バイト・単精度整数型
- int:4 バイト・整数型
- long:4 バイト・整数型
- float:4 バイト・単精度浮動小数点型
- double:8 バイト・倍精度浮動小数点型
- auto:自動変数、関数を抜けるとデータは消去、省略時のデフォルト
- static:静的変数、関数を抜けてもデータは残る
- const:書き換え不可、宣言時に格納

- signed:符号付変数を指定、省略時のデフォルト
- unsigned:符号なし変数を指定
- extern:異なるファイルから使用する際に宣言
- volatile:コンパイラに最適化をさせない
- register:レジスタに割り当て高速化、C++では使用出来るが意味は無い
- return:関数から抜ける、戻り値を指定できる
- goto:指定ラベルへジャンプする
- if:条件分岐 (else, else if, を利用)
- else:if 文の条件分岐、(else if) ととも使用
- switch:条件分岐、(case, break, default などを使用)
- case:switch 文での条件分岐
- default:switch 文での case に当てはまらない条件
- break:ループ文から抜ける、case 文の終了
- for:ループ文、(初期化; 終了条件; 変数更新)
- while:ループ文、条件が真の場合に繰り返す
- do:do-while 文で使用する、処理の開始
- continue:ループ文の先頭に戻る
- typedef:型に別名を付ける、意味は変わらず
- struct:構造体、変数をまとめて宣言するユーザー定義型
- enum:列挙型、整数の割り当て
- union:共用体、変数をまとめて宣言できるが、アドレスは共通
- sizeof:変数のサイズを取得

以上が C89 の予約語で、C99 は上記に加えて以下も予約語である。

- inline:コンパイラにインライン展開させ、高速化
- restrict:ポインタでアクセスする事を明示
- _Bool:真偽値型、0=false, 1=true (C++では bool 型)
- _Complex:複素数型 (float, double, long double 型の後に使用)
- _Imaginary:虚数型 (float, double, long double 型の後に使用)

B.3 演算子の評価順序

C言語における演算子の評価順序を記す。なお、演算順序を変更する `()` は、これらより先に評価される。また、前置イン/デクリメントはこれらの評価よりも前に加減算のみを行い、後置イン/デクリメントはこれらの評価よりも後に加減算を行う。

1. 参照演算子 `.` と `->`、関数引数の `()`、配列添字の `[]` が評価される。
2. キャスト以外の単項演算子が評価される。
3. キャストが評価される。
4. 2項の乗除算・剰余が評価される。
5. 2項の加減算が評価される。
6. ビットシフトが評価される。
7. 大小関係を表す比較演算子が評価される。
8. `==`および`!=`演算子が評価される。
9. 2項のビット演算子が評価される。ビット演算子は`&`,`^`,`|`の順の評価となる。
10. 論理演算子が評価される。ただし、`&&`,`||`の順の評価となる。
11. 条件演算子が評価される。
12. 代入演算子が評価される。
13. コンマ演算子が評価される。

B.4 マクロの詳細な文法

マクロに関する演算子と可変引数マクロについて説明する。

B.4.1 文字列化演算子 `#`

関数マクロ定義中において、演算子`#`を仮引数の前につけると、これはその仮引数を文字列化する役目を持つ (文字列化演算子 (Stringizing Operator))。たとえば、

```
#define PUTNAME(var) puts(#var)
```

というマクロを定義したとしよう。このマクロは、”実引数を標準出力するマクロ”である。以下のように呼び出しを行ったとしよう。

```
PUTNAME(test);
```

この時、このマクロは次のように展開される。

```
puts("test");
```

先には文字列化と記したが、決して `char *` 型の結果を返すのではなく、文字列リテラル化する演算子であるため、実際のコード中では文字列リテラルとして扱えばよい。たとえば、次の 2 つの `printf` 文は同じ出力になる。

```
#define NAME(var) #var

printf(NAME(arg) "%d\n", arg);
printf("%s=%d\n", NAME(arg), arg);
```

これはコンパイル時に隣接する文字列リテラルが結合されるという、文字列リテラルの性質を利用したコードである。

B.4.2 字句連結演算子 `##`

字句連結演算子 (Token-Pasting Operator) ないしトークン連結演算子 `##` は、やはり関数マクロ中で用いられ、コンパイル時に 2 つの字句を連結する。たとえば、

```
#define MYFUNC(x) myfunc_##x
```

という定義を行い、

```
MYFUNC(var)+MYFUNC(foo);
```

という記述を行ったとしよう。この時、マクロを展開すると

```
myfunc_var+myfunc_foo;
```

となる。このように、定まった接頭辞/接尾辞をつける際にこの演算子は役立つ。条件付きコンパイルと組み合わせることにより、ソースをかなり自在に書き換えることができる。

B.4.3 可変引数関数マクロ

C99 以降では関数マクロの引数が可変引数であることが許される。通常関数マクロとは違い、唯一の引数が可変引数であってもよい (もちろん、可変引数の前に識別子の並びをとっても良い。)

可変引数関数マクロを定義する際には、可変引数関数同様に

```
#define MACRO(...)
```

のように記し、可変引数を展開したい部分にはマクロ `__VA_ARGS__` を記す。

エラー出力用の関数マクロ `eprintf` を定義することを考えよう。これは次のように定義することができる。

```
#define eprintf(format,...) fprintf(stderr,format,__VA_ARGS__)
```

この関数マクロは `printf` と同様の引数をとることで、`stderr` への出力を行う。可変引数関数を用いて同様の機能を作ることができる (この場合は `vfprintf` を用いると便利) が、やはり型チェックなどの点での差異があるため、必要に応じて使い分けたい。

B.5 volatile 修飾子

型修飾子として `const` と `restrict` を紹介したが、C 言語の型修飾子にはもう一つ **volatile** 修飾子がある。これによって修飾された識別子については、処理系で行われる最適化を抑止する効果がある。組み込み処理やマルチスレッドプログラミングなどでよく用いられる。

最適化を抑止するというのは、コンパイル時に条件文をうまくまとめて減らしたり、処理手順を減らしたりする場合の問題を防ぐためにある。実際の活用については本書の範囲を大きく超えるので、ここではひとまず紹介のみとしておく。

なお、volatile 修飾子に関する文法は `const` のそれと同様であり、volatile 型修飾子がついた変数へのポインタなども存在する。

B.6 ロケール

ロケール (locale) とは、ソフトウェア毎に定められた、言語/国/地域への対応/指定機能である。ロケールを変更すると、日付、通貨記号などを始め、小数点の表記などに至るまで、様々な点で国に応じた変更が加えられる (但し、コンパイラがそのロケールに対応していればだが)。C 言語では、ロケール設定の機能は `locale.h` に入っている。ここでは、その設定方法を簡潔に記すに留める。

一般に、ロケールを設定する際には、

```
setlocale(LC_ALL, "ja");
```

などのように、`setlocale` 関数を用いる。第 1 引数に設定したいロケール属性を記し、第 2 引数にコンパイラのサポートしている国別コードを記す。第 1 引数の属性は、`locale.h` で定数として定められており、ここで用いた「全て変更する」`LC_ALL` 以外にも日付を指定する `LC_TIME` や数値の書き方を示す `LC_NUMERIC` などがある。第 2 引数の "ja" は日本のロケールであるが、これについてはそれぞれのコンパイラで確認されたい。なお、`setlocale` において対応していないなどの理由でロケールの設定が失敗した場合は `NULL` ポインタが返却される。そのため、`NULL` ポインタになっていないかどうかをチェックすることで例外処理を行うことができる。

一般にロケールは `locale.h` 内の関数のみを用いて変更し、自作のソースなどで変更すべきではない。同じ `locale.h` には `locale` を扱うための構造体 `struct lconv` とそれを取得するための関数 `localeconv` 関数が用意されているが、これはあくまでも移植性の確保などのためであって、`setlocale` 以外で変更するのは推奨されない。

なお、C 言語の標準ロケールは "C" であり、規格上はこのロケールにさえ対応していれば C 言語のコンパイラとしての要件は満たしている。"C" ロケールは元々の C 言語の定義に対して矛盾しない最低限のロケール設定を行うロケールである。

B.7 マルチバイト文字とワイド文字

日本語をはじめ、1 バイトで表せない文字は多い。これら 1 バイトでない文字は次の二つのいずれかの方法で表される。

1 バイトで表せない文字の表現方法

- 文字ごとに表すために用いるバイト数が異なるマルチバイト文字 (multibyte character)。
- すべての文字を同じバイト数で表現するワイド文字 (wide character)。

これらを扱うための関数群が標準ライブラリに用意されている。以下、これらのワイド文字/マルチバイト文字の扱いを簡単に説明する。

B.7.1 マルチバイト文字

さて、先に書いたとおり、マルチバイト文字は複数バイトで文字を表す方法である。日本語の場合、Shift_JIS や EUC-JP, UTF-8 など主要な文字コードは何れもマルチバイト文字である。これらの文字数などを適切に数えたい場合には、前もって日本語ロケールを設定した後、mb... などという名前の関数を用いれば良い。char 型などに突っ込んでも、大抵は適切に動作する。

C 言語で扱うマルチバイト文字の一部に 0x00 は存在しない。すなわち、マルチバイト文字を用いても途中で文字列が終わることはない。しかし、例えば\マークとなるような値が含まれることがある。これにより、文字化けを起こすこともあるのが、マルチバイト文字の難点である。一般に、日本語の処理を行える環境ではこれらを適切に処理してくれるが、必ずしも上手くいくとは限らない。上手くいかない場合には、ロケールの設定の対策などが必要になる。

B.7.2 ワイド文字

ワイド文字は1文字あたりのサイズが決まっているため、マルチバイト文字に比べれば、扱いが簡単のように見えるだろう。この為か、C 言語の規格で定められた関数はワイド文字に対するものが多いように思える。C95 で追加されたヘッダである、wctype.h や wchar.h はワイド文字に関するヘッダである。

この wchar.h において、ワイド文字を扱うための型 `wchar_t` が定義されている。ワイド文字が何バイトかは定まっていないので、`wchar_t` のサイズは処理系に依存する。

ワイド文字定数やワイド文字列リテラルを扱う場合、通常の文字定数/文字列リテラルの記法の前に `L` を付す。出力の際には `wprintf` 関数を用い、ワイド文字列の書式指定子には `%ls` を用いる。この規則さえ覚えておけば、残りの扱い方は文字列の扱いと大差ない。関数群が `str...` から `wcs...` に変わるだけである。

マルチバイト文字列を扱う場合も、日本語ロケールの設定を行なって、char 型配列にマルチバイト文字列を読み込んだ後、ワイド文字に変換して処理を行うほうが楽に書くことができる。マルチバイト文字/ワイド文字の変換には、`wctomb`, `mbtowc`, `mbstowcs`, `wcstombs` の4つの関数を用いると便利である。

付録C Cに関連したWebサービス

ここでは、C 言語に関連した Web 上のツールを紹介する。

- ideone.com
<http://ideone.com/>
Web 上において IDE を提供しているツール。C 言語をネットで扱うことが出来るため、環境を用意する必要がなく便利である。
- Share code real time
<http://codeshare.io/>
C 言語に限らず、ソースコードを共有することが出来るツール。コードレビューの際などに便利。
- Github
<https://github.com/>
ソースコードの等の管理に用いられるツール。

この他、ダウンロード形式で手に入れられるツールは枚挙に暇がない。最近では、スマートフォンにも GCC が入れられる他、スマートフォンのアプリとして、無料で C 言語のマニュアルを提供しているソフトもある (英語)。

参考書籍等紹介

本書を読む上で、あるいは読まれた後で参考となる書籍・サイトを紹介しておく。なお、参考となる書籍/サイトはこの他にも多数あるが、ここでは目についたものを紹介するにとどめた。実際には、学習者自身の目で本・サイトを選ばれたい。

文法の解説

- C 言語プログラミング (H.M. ダイテル/P.J. ダイテル著 小嶋隆一訳・ピアソンエデュケーション): 現在出ている C の文法書籍の中で、最も練習問題が多いもののひとつ。説明もなかなかわかり易く、内容も浅くない。
- C 実践プログラミング (S. オウアルライン著 望月康司監訳・オライリージャパン): 上記ダイテルに比べてやや難しめであるが、内容に妥協のない C の文法書。
- 美しい C プログラミング見本帖 (柏原正三著・翔泳社): C 言語の難所「ポインタ」の観点から、「習うより慣れろ」の精神で文法を見ていく好著。ポインタのみを書いている書籍よりも広い視点で C を学ぶことができる。
- 本当は怖い C 言語 (種田元樹著・秀和システム): 本書同様、C 言語を理論的に説明した好著。エラーを出すプログラムを掲載し、それにより学ぶなど随所に工夫が見られる。C99 にも対応している。
- 苦しんで覚える C 言語 (<http://9cguide.appspot.com/>): 私の知る限り最も丁寧な解説サイト。Windows での環境構築から全て書いてあり、非常にわかりやすい説明が行われている。最近書籍版が出た。
- 目指せプログラマー (<http://www5c.biglobe.ne.jp/~ecb/index.html>): C 言語を始め幾つかの言語と、アルゴリズム論の基礎を解説しているサイト。
- Programming Place Plus(http://www.geocities.jp/ky_webid/index.html): 他に見られない項目も含めて細かく C を解説しているサイト。

C リファレンス

- C リファレンスマニュアル 第 5 版 (S.P. ハービソン/G.L. スティール Jr. 著 玉井浩訳・SIB アクセス): 現在出ている C の辞書では最も充実しているもののひとつ。
- C 言語によるプログラミングスーパーリファレンス編 (内田智史他著・オーム社): 初心者向けの C の辞書だが、1999 年の C の改定に対応していないのが難点。

その他、参考になる本

- プログラミング言語 C 第 2 版 (カーニハン, リッチー著 石田晴久訳・共立出版):C 言語の原典である書籍。
- エキスパート C プログラミング (P. リンデン著 梅原系訳・アスキー):C 言語の内容についてかなり深く突っ込んだ書籍。
- C/C++の「迷信」と「誤解」(高木信尚著・技術評論社):非常に細かい、勘違いしやすい場所について仕様を基に突っ込んでいく書籍。
- 補講 C 言語 (平田豊著・工学社):入門書では余り触れられない実践テクニックを紹介する書籍。
- C 言語によるはじめてのアルゴリズム入門 (河西朝雄著・技術評論社):アルゴリズムについて、C 言語で、簡単なものを中心に紹介した好著。
- C 言語による最新アルゴリズム事典 (奥村晴彦著・技術評論社):「最新」ではなくなってきたが、C 言語を用いて多くのアルゴリズムが書かれている。
- アルゴリズム演習 300 題 (橋本英美著・日刊工業新聞社):簡単な問題を中心としたプログラミングの演習書。初歩的な問題も掲載されており、練習に調度良い。
- 演習でマスターする C 言語とデータ構造 (内藤広志, 斉藤隆著・共立出版):データ構造と派生型について 1 冊通して演習する書籍。
- プログラマのための論理パズル (D.E. シャシャ著 吉平健治訳・オーム社):プログラミングに必要な様々な発想をパズルを元に学ぶ書籍。アルゴリズムの能力を上げるのに役立つ。
- Short Coding(Ozy 著・毎日コミュニケーションズ):C 言語で「できる限り短いソースを書く」ことに視点を当てて書かれた書籍。この書籍を楽しむことで C 言語やアルゴリズムへの深い理解を得ることができる。お遊びの本といえばそうなのであるが、驚くほど有用である。
- プログラミングコンテストチャレンジブック (秋葉拓哉他著・毎日コミュニケーションズ):競技プログラミングに焦点を当てたプログラミングの書籍。
- ニューメリカル・レシピ・イン・シー (W.H. プレス他著 奥村晴彦他訳・技術評論社):数値計算に関する、標準的な参考書。
- ロベールの C++入門講座 (ロベール著・毎日コミュニケーションズ):C++の入門書であるが、前半 5 章は C にも共通する部分であり、非常に良い解説が行われている。引き続き C++を学ぶ気があるならば推奨する。ページ数の割に安い。
- ストラウストラップのプログラミング入門 (B. ストラウストラップ著 遠藤美代子訳・翔泳社):C++を作った人が書いた、プログラミング全般を C++を用いて入門するという内容の書。

おわりに

「お疲れ様でした！」本書を読み終えられた人に、まずはそう申し上げたい。同時に、構想から1年以上をかけ、ようやくテキストを書き終えた自分にも同じ言葉を贈りたい。

近年、「C言語は古びた言語であるから、学ぶ価値がない」というような論調を見聞する。たしかに、C++やJavaなどのより強力な言語があるし、LLも数多く出ている現在、Cを実践で使う機会は減ってきているのかも知れない。だが、私は「C言語を一度学ぶことはどの言語を主で使うにしても有用である」と思う。これは例えば、英文学を考えてみればわかるだろう。シェイクスピアの作品は古典と言われる部類に属するだろうが、非常に豊かな語彙をはじめとして、現代から見ても決して読む価値のないものではない。それどころか、シェイクスピアを知らないことは無教養とさえされているところがある。我が国で言えば、明治から大正にかけての文学は、たしかに古いものであるが、そこから学ぶところは決して少なくないだろう。

この様な重要な古典としてC言語を捉えた時、多くの言語に影響を与えていることがわかる。C言語を学ぶことは、現代よく用いられる言語の背景にある考えを理解することである。科学を見ても歴史を見ても、背景の考えを理解することが有用なのは言うまでもないことであろう。

これらの「背景の考え」を初心者のうちから理解していれば、他の言語の学習も捗る。この考えから、本書は「C言語の文法を中心に、背景の考えをよく理解できる説明」を行うようにしたつもりである。それがどの程度達成されているかは読者諸賢の判断を待つしかないが…。

本書は、理論の面からプログラミングの楽しさを伝えようとしてみたものであるが、果たして上手く伝えることができただろうか。その答えは、読者の皆様がこれからプログラミングに携わっていく中で自ずと見えてくることだろう。もしも本書を読むことによってプログラミング好きが一人でも増えたのであれば、本書の目的は達成されたといえるだろう。

本テキストを利用してくれた人たち、本テキストの執筆に携わってくれた全ての人たちに心よりの感謝を捧げ、筆を置くことにする。

中島みゆき「ローリング」を聞きながら
達哉ん

索引

A

AND 演算	48
Bellman-Ford 法	222
BFS	→ 幅優先探索
Class-NP	229
Class-P	229
const 修飾子	36
CPU	5
CUI	9
D&C	→ 分割統治法
DFS	→ 深さ優先探索
Dijkstra 法	223
DP	→ 動的計画法
FIFO	209
FILO	208
Gauss-Jordan 法	260
Gauss 法	257
GUI	9
include 文	24
Kahan の加算アルゴリズム	46
LIFO	208
LILO	209
LU 分解	259
main 関数	24
Newton 法	250
NOT 演算	48
NULL ポインタ	170
NULL 文字	132
OR 演算	48
OS	8
RAM	7
restrict 修飾子	177
ROM	7
sizeof	40
UI	9
void	83

volatile 修飾子	293
Warshall-Floyd 法	224
XOR 演算	48

あ

浅いコピー	179
値渡し	157
アドレス	8, 154
アドレス演算子	157
アドレス渡し	157
アラインメント	142
アルゴリズム	21
アロー演算子	167
アンダーフロー	34
安定なソート	230
暗黙の型変換	42
行きがけ順	214
1 バイト文字	4
入次数	225
入れ子	→ ネスト
インクリメント	40, 41
インクルードガード	95
インクルード文	95
インターポジショニング	51
インデックス	121
インデント	22
イントロソート	241
インライン関数	116
インラインコメント	24
エスケープシーケンス	27
枝	213
エッジ	219
オイラーグラフ	225
オイラー閉路	225
オイラー路	226
応用ソフトウェア	8
オーバーフロー	32

オーバーロード	54, 84
オクテット	4
オブジェクト形式マクロ	90
オフセット	121
オペランド	53
オペレーティングシステム	→ OS
(辺の) 重み	219

か

ガーベッジコレクション	187
外部リンケージ	97
カウンタ	74
帰りがけ順	215
カクテルソート	232
仮想メモリ	5
型変換	42
片方向リスト	211
可変長配列	121
可変引数関数	113
下方移動	217
仮引数	81, 84
カレントディレクトリ	11
関係演算子	→ 比較演算子
関数	25, 80
関数形式マクロ	91
関数原型	→ プロトタイプ宣言
関数プロトタイプ	→ プロトタイプ宣言
間接演算子	163
間接参照演算子	167
完全グラフ	220
完全二分木	214
木	213
記憶クラス指定子	88
記憶装置	5
擬似乱数	71
記数法	1
基本型	119
基本ソフトウェア	→ OS
キャスト	42
キャッシュメモリ	6
キュー	209
共用体	147
局所変数	→ ローカル変数

クイックソート	237
空間計算量	229
位取り記数法	1
グラフ	219
グローバル変数	87
計算誤差	45
桁落ち	45
高級言語	18
高水準言語	→ 高級言語
構造化定理	19
構造化プログラミング	19
構造体	138
構造体のネスト	143
後置反復	71, 72
コーディング	21
5大装置	5
固定小数点数	33
コマンドライン引数	204
コムソート	239
コメント	23
コンパイル	19

さ

サーチ	243
再帰	110
再帰関数	110
最短路問題	221
3文字表記	→ トライグラフ
参照	29
参照渡し	85, 131
シーケンシャルアクセス	7
シェーカーソート	232
シェルソート	233
時間計算量	229
識別子	30, 84
字句連結演算子	292
自己参照構造体	211
指示付き初期化子	124, 141
(ノードの) 次数	225
指数オーダー	229
実引数	81, 84
主記憶装置	5
出次数	225

10 進数	1
10 進法	1
出力装置	5
準オイラーグラフ	226
循環リスト	211
準ハミルトングラフ	226
準モンテカルロ積分	257
条件演算子	66
条件付コンパイル	92
上方移動	216
情報落ち	46
初期化	29
書式指定子	37
書式文字列	36
処理装置	5
シンプソン則	255
数値計算法	246
数値微分	252
スコープ	86
スタック	208
スタックオーバーフロー	8, 112
スタック領域	7, 154
ストランドソート	240
ストリーム	195
スパゲッティソースコード	64, 78
静的配列	120
静的変数	89
静的領域	7, 154
セグメンテーション違反	8
接続行列	221
節点	→ ノード
線形探索	243
線形リスト	211
宣言	29, 87
前進差分近似	252
選択演算子	142
選択ソート	231
前置反復	71
全点对最短路問題	221
全二分木	214
走査	214
相対パス	11
挿入ソート	230

双方向リスト	212
ソース	22
ソート	228

た

ターミネータ	70
大域変数	→ グローバル変数
台形積分	182, 254
台形則	→ 台形積分
対数オーダー	229
代入演算子	41
多項式オーダー	229
多次元配列	125
多重定義	→ オーバーロード
単一始点最短路問題	221
単純グラフ	220
単純挿入法	230
逐次探索法	247
中央演算処理装置	→ CPU
中央差分近似	252
中点則	255
頂点	→ ノード
定義	87
低級言語	19
低水準言語	→ 低級言語
定数オーダー	229
ティムソート	241
データ型	30
データ構造	208
テキストファイル	194
デクリメント	40, 41
デック	211
デバッグ	21, 23
動的計画法	122, 224
動的配列	121, 187
トークン連結演算子	292
通りがけ順	215
ドット演算子	142
トライグラフ	109

な

内部リンケージ	97
名前空間	158
2038 年問題	101

2 項演算子	41	比較演算子	57
2 進法	1	比較関数	234, 238
2 頂点对最短経路問題	221	引数	25
二分木	214	ビット	4
二分探索	243	ビットフィールド	145
二分探索木	217	ピボット (クイックソート)	237
二分ヒープ	215	(連立方程式の) ピボット選択	258
二分法	247	標準エラー出力	195
入力	29	標準関数	80
入力装置	5	標準出力	195
根	213	標準入出力	195
ネスト	57	標準入力	29, 195
ノード	213, 219	標準ライブラリ	24
ノームソート	232	ビンソート	→ バケットソート
は		ファイル	194
葉	213	ファイルポインタ	197
排他制御	199	不安定なソート	230
バイナリサーチ	243	深いコピー	179
バイナリファイル	194	深さ優先探索	214
配列	119, 120	不完全配列	166
バグ	23	複合条件文	61
バケットソート	242	複合リテラル	141
はさみうち法	250	副作用	81
パス	11	浮動小数点数	33
派生	119	部分木	213
派生型	119	プライオリティーキュー	211
バッファオーバーラン	133	プリプロセッサ命令	24
幅優先探索	215	フレキシブル配列メンバ	190
バブルソート	228	プログラミング	17
ハミルトングラフ	226	プログラミング言語	18
ハミルトン閉路	226	プログラム領域	7, 154
ハミルトン路	226	ブロックコメント	24
半角文字	35	プロトタイプ宣言	85
反復構造	69	分割統治法	175
反復制御文	75	分岐構造	56
反復法	248	分布数えソート	→ バケットソート
番兵	151	ヘッダファイル	24
汎用ポインタ	161	辺	→ エッジ
ヒープ	215	返却値	25
ヒープソート	239	変数	29
ヒープ領域	7, 154	変数の寿命	88
被演算子	→ オペランド	ポインタ	160
		ポインタのポインタ	184

ポインタ渡し	85
補助記憶装置	5
補数表現	32
<hr/> ま	
マージ	234
マージソート	235
前処理命令	24, 89
マクロ	90
マルウェア	159
マルチバイト文字	294
マルチブート	10
丸め誤差	45
無向グラフ	219
メモ化再帰	112, 122
メモリ	6
メモリリーク	188
メンバ	138
文字コード	34
文字列	119, 131
文字列化演算子	291
文字列リテラル	132
森	220
モンテカルロ積分	256
モンテカルロ法	256
<hr/> や	
有向グラフ	219
有効範囲	86
予約語	31
<hr/> ら	
ライブラリ	24
ラスベガス法	256
乱数の種	71
ランダウの漸近記法	229
ランダムアクセス	7
リスト	211
リダイレクト	14, 196
リテラル	38
リニアサーチ	243
両端キュー	211
リンクリスト	→ リスト
リンケージ	97

隣接行列	221
例外	105
例外演算	34
例外処理	105
レギュラ・ファルシ法 . → はさみうち法	
レジスタ	6
レジスタ変数	88
列挙型	149
連結グラフ	220
ローカル変数	87
ロケール	293
論理演算子	60
論理演算装置	5
<hr/> わ	
ワイド文字	294
ワイルドカード表現	13
割線法	248