

CONTENTS

第1部 C言語プログラミングの基本構造

第1章	プログラムってなんだろう？	プログラミング言語とは	13		
1.1	コンピュータにとってプログラムってどんなもの？		14		
1.1.1	コンピュータ、プログラム、人間の関係	15	1.1.4	より簡単にコンピュータに指示を伝えるための方法とは？	19
1.1.2	融通のきかないコンピュータ	16	1.1.5	やさしく巧みなプログラム	20
1.1.3	コンピュータがわかる言葉ってどんなもの？	18			
1.2	プログラミングに必要なこと		21		
1.2.1	プログラミングの考えに慣れること	21	1.2.2	プログラミングに必要な3つの知識	22
1.3	プログラムの考え方		23		
1.3.1	問題の曖昧なところをはっきりさせる	24	1.3.3	柔軟な発想で考える	29
1.3.2	問題を細かく分解して整理する	26			
1.4	プログラムの作り方		32		
1.4.1	いろいろなプログラミング言語	32	1.4.4	プログラムを実行し、正しく動いているか調べる(動作確認)	38
1.4.2	プログラムを書く(コーディング)	34	1.4.5	プログラムの誤りを修正する(デバッグ)	39
1.4.3	プログラムを翻訳する(コンパイラとリンク)	35			
第2章	はじめの一步	記述規則を実践理解	47		
2.1	最も単純な構造のプログラムを入力して実行する		48		
2.1.1	プログラムを入力して実行してみる	48	2.1.3	プログラムを読みやすい形にするとは	50
2.1.2	プログラムを書き換えてみる	49			
2.2	エラーメッセージと警告メッセージの初歩の初歩		52		
2.2.1	エラーなのか、警告なのかを読み取る	52	2.2.4	うっかり書き忘れが原因であることが多いエラー表示	59
2.2.2	いろいろな警告表示	53	2.2.5	ほんとうに怖い「表示されないエラー」	61
2.2.3	スペルミスが原因であることが多いエラー表示	56			
第3章	データを入力して、結果を表示してみよう	入出力処理	65		
3.1	結果を表示するということ		66		
3.2	プログラムの詳細		67		
3.2.1	C言語の構成と記述規則	67	3.2.3	結果を表示する関数 printf() の詳細	71
3.2.2	プログラムを見やすくするために	71			

3.3	値を記憶しておく箱を利用して結果を求めて、表示する	75
3.3.1	変数という箱を用意する	76
3.3.2	データ型の種類と詳細	80
3.4	プログラムを実行しながら、さまざまな結果を得るには？	83
3.4.1	プログラムを実行中に、値を入力する	84
3.4.2	入力を受ける関数 scanf() の詳細	65
3.5	ここまでの知識でどんなことができるのか？	86

第4章 プログラムの処理の流れを理解し、使いこなす① 分岐処理 93

4.1	プログラムの処理の流れの重要要素「分岐処理」とは？	94
4.2	条件判断を行って、分岐する処理を行う (if else 文と if 文)	95
4.2.1	処理の流れをあらわすと…	95
4.2.2	もし○○ならば処理1を行い、そうでなければ処理2を行う (if else 文)	96
4.2.3	もし○○ならば○○の処理を行う (if 文)	98
4.2.4	入れ子の分岐処理	101
4.3	複雑な分岐処理を見やすく記述する (switch case)	104
4.4	分岐処理の詳細	110
4.5	どんなときにどの分岐処理を使えばよいのか？	113

第5章 プログラムの処理の流れを理解し、使いこなす② 繰り返し処理 123

5.1	プログラムの処理の流れの重要要素「繰り返し処理」とは？	124
5.2	同じ処理を繰り返す① (for 文)	125
5.2.1	同じ処理を繰り返したい場合とは？	125
5.2.2	決まった回数繰り返す (for 文)	127
5.3	同じ処理を繰り返す② (while 文と do while 文)	131
5.3.1	○○となるまで何度も繰り返す (while 文)	131
5.3.2	次のことを繰り返す、ただし、○○となった ら終了する (do while 文)	134
5.4	繰り返し処理の詳細	131
5.4.1	どんなときにどの繰り返し処理を使えばよいのか？	139
5.5	ここまでの知識でどんなことができるのか？	140

第6章 たくさんの値を記憶する 配列の利用 151

6.1	たくさんの値を記憶する必要性	152
6.1.1	実用的なプログラムを作るために必要なこと	152
6.1.2	ここまでの記述方法での限界	153

6.2	配列とは値を入れる箱 (変数) をまとめて棚を作ること	155
6.3	いろいろな棚 (配列) の作り方	156
6.3.1	一列に並べて、何番目として管理する (1次元配列)	156
6.3.2	処理を簡単化するための発想の流れ	161
6.3.3	棚を作り、何番目の何番目として管理する (2次元配列)	164
6.4	配列の使用法の詳細	170
6.4.1	変数の復習	170
6.4.2	配列の宣言の記述方法	171
6.4.3	配列の扱い方	172
6.5	どんなときに配列を使えばよいのか？	172
6.6	配列を「繰り返し処理」と組み合わせると何倍も便利に！	174
6.7	ここまでの知識でどんなことができるのか？	175

第7章 データを保存する・保存したデータを読み込む ファイルの利用 183

7.1	データを保存すること、保存したデータを読み込むこと	184
7.2	ファイルを利用してデータを入力するにはどのようにすればよいのか？	186
7.2.1	大量のデータを入力して結果を表示させる	186
7.2.2	入力するデータをテキストファイルにしておくと……	188
7.3	ファイルを利用してデータを出力するにはどのようにすればよいのか？	191
7.3.1	結果を画面に表示するプログラム	192
7.3.2	結果をファイルに書き込むプログラム	194
7.4	結果を保存しておき、次回プログラムを実行したときに保存データを読み込む	196
7.5	ファイルを利用するときにはエラー処理も必須	201
7.5.1	実用的なファイルのオープン方法	202
7.6	ファイルの利用方法の詳細	204
7.6.1	ファイルを操作できる状態にする、操作を終える (fopen, fclose)	204
7.6.2	ファイルから読み込む、ファイルに書き込む (fscanf, printf)	205

第2部 アルゴリズムを組み立てる

第8章 プログラムで文字を扱うには？ 文字と文字列の取り扱い 211

8.1	プログラムで文字を扱うということ	212
8.2	C言語で文字列を扱うにはどうしたらよいのか？	213
8.2.1	文字列を扱うには配列を使う	213
8.2.2	文字列の代入方法	214
8.3	コンピュータでは文字をどのように扱っているの？	219
8.3.1	すべての文字は番号で管理されている！?	220
8.3.2	文字型の1次元配列の中身	223

8.3.3	文字を比較する？	226
8.4	文字の基本的取り扱い方の整理	229
8.5	基本的な文字の取り扱い方の詳細	232
8.5.1	文字列の読み込み・ファイル入力方法の詳細	232
8.5.2	文字列の表示・ファイル出力方法の詳細	235
第9章	文字列をもっと自在に扱うには？ 文字列処理の関数利用	243
9.1	文字列を操作する便利な関数	244
9.1.1	C言語の標準にはない関数を利用する方法	244
9.1.2	文字列をコピーする	246
9.1.3	文字列をつなぎ合わせる	248
9.1.4	文字列を比較する	250
9.1.5	何文字あるか調べる	252
9.2	便利な文字列操作関数の詳細	254
9.3	文字列を利用する応用場面	256
9.4	ファイルの中身をすべて読み出す	259
9.5	さらに自在に文字列をコントロールする応用テクニック	262
9.5.1	文字列からの読み込み・文字列への書き出し	232
第10章	新しい機能を設計する 独自に関数を作る	271
10.1	標準で用意されている関数と自分で作る関数	272
10.1.1	C言語は、関数で成り立っている	272
10.1.2	用意されている関数がなければ自分で作る！	273
10.1.3	便利な関数は誰かが作ってくれている！？	274
10.2	自分で関数を作って、利用してみよう	275
10.2.1	関数を作る場面1：何度も使う記述は1回だけにまとめる	275
10.2.2	関数を作る場面2：プログラムを機能別に見やすくまとめる	282
10.2.3	関数を作る場面3：他のプログラムでも利用できる資源を作る	288
10.3	さまざまな関数を作ってみよう	289
10.3.1	変数を選さない関数	290
10.3.2	変数を選ぶが、変数の値は変更しない関数	291
10.3.3	変数を選び、変数の値を変更する関数	292
10.3.4	配列を関数に渡して利用する	295
10.3.5	関数の名前を事前登録しておく～関数のプロトタイプ宣言	297
第11章	関数を呼び出して活用する 標準ライブラリの利用	305
11.1	関数を活用する意義	306

11.2	ちょっと便利な関数を使う～stdlib.hの活用	306
11.3	数学知識を活用する～math.hの活用	310
11.4	文字の取り扱いツールを活用する～ctype.hの活用	312
11.5	時間をコントロールする～time.hの活用	315
11.6	再帰関数に触れてみる	317

第12章	データをまとめて管理する 構造体	323
12.1	どんなふうにデータをまとめて扱うと便利か？	324
12.2	実際にデータをまとめてプログラミングをしてみよう	325
12.2.1	単純にデータをまとめる	325
12.2.2	構造体×配列で効果絶大！	330

第13章	アドレスとポインタを活用し中級プログラミングに挑戦	343
13.1	変数とコンピュータのメモリの関係	344
13.1.1	動作しているプログラムがメモリをどのように利用しているのか調べてみる	344
13.1.2	変数のアドレスを表示させる方法	347
13.1.3	変数のアドレスを確認する	348
13.2	配列とコンピュータのメモリの関係	349
13.3	scanf()や関数の利用を振り返る	352
13.3.1	アドレス演算子「&」と間接演算子「*」	353
13.3.2	配列の場合の関数受け渡し	353
13.4	基本的なポインタの使用例	354
13.5	構造体とポインタの共演	357
13.6	メモリの動的確保と利用	359

第14章	プログラミングの道はまだまだ続く その他の記述方法	367
14.1	ここまでで紹介しなかった「実際に使えるプログラミング技法」	368
14.1.1	何度も記述する定数をあらかじめ定義しておく	368
14.1.2	定義に名前をつける	369
14.1.3	ファイルの名前を与えて読み込む・書き出す	371
14.1.4	何行書いてあるかわからないファイルを全部読み込みたい	372
14.2	デバッグに役立つ小技！	373
14.2.1	コンパイラのエラーがどこを指すのかを特定する小技	374
14.2.2	コンパイルは正常終了し、実行結果がおかしくなるときのエラー箇所を見つける小技	375

14.2.3 エラーの場所がわかったあと、どうする? 376	14.2.4 便利な道具「デバッガ」を使う 378
--------------------------------	---------------------------

14.3 共同作業の第一歩 ～分割コンパイル 378

14.3.1 4つのプログラムのファイルを同じプロジェクトに追加する 379	14.3.2 4つのファイルについて 382
--	------------------------

14.4 これからどのようなことを学んでいけばよいのか? 382

14.4.1 次に学びたいことは 383	14.4.2 さらにプログラミング技術を高めるために 383
----------------------	--------------------------------

■ 章末練習問題の解説編 385

More Information

なぜさまざまな種類のコンパイラがあるのか?	36
どうしてC言語	42
歴史あるアルゴリズム - Tower of Hanoi	45
生成AIとプログラミング	46
日本語文字の文字コードの呪い	62
時代とともに変化するC言語	64
C言語で関数というのはどういうものなの?	70
論理演算子について	111
無駄をなくして効率のよいプログラムを作るには? ①	119
無駄をなくして効率のよいプログラムを作るには? ②	149
大きなプログラムを作るときの心得	181
コンピュータにおけるファイルの種類	185
エラー処理にはどんなことが求められるの?	201
日本語の文字列を扱うためのヒント	264
一歩進んだ関数の使い方	298
標準ライブラリを調べてみよう	320
巨大データの並び替え	356
必要なときに必要なだけ記憶する場所を作って管理する	362

1.1

コンピュータにとってプログラムって
どんなもの？

「パソコン」といえば、もっと身近に感じるでしょうか。パソコンはパーソナル・コンピュータの略、個人用コンピュータという意味です。昔は個人がコンピュータを所有するなんて考えられなかったもので、個人用コンピュータが盛場したとき、わざわざこの名称が付けられ今に至っています。というわけでパソコンももちろん、コンピュータです。

※計算を得意とするソフト、Microsoft Excelという商品がある。

コンピュータ*を使っていると聞いたとき、どんなイメージが浮かびますか？画面を見つめて、キーボードをカタカタさせて…マウスでカチカチ操作して…というイメージが浮かぶ方が多いのではないのでしょうか。では、画面を見つめて、キーボードをカタカタさせて、何をしていますのでしょうか？ワープロで文章を書いたり、表計算ソフト*で伝票を整理したり、インターネットにつないでメールを読んだり……、いろいろなイメージが浮かびます。

なかにはスマートフォンをイメージした方もいるのではないのでしょうか。多くの人にとって、現在最も身近なコンピュータはスマートフォンだと言っても過言ではありません。そうしたスマートフォンでも、インターネットにつないでSNSを利用したり、メールを書いたり、Webで情報収集したり、動画・映画を視聴したり……いろいろな操作イメージがあると思います。スマートフォンとノートパソコンやデスクトップパソコンとの間に、本質的な違いはありません。

デスクトップパソコン、ノートパソコン、タブレット端末、スマートフォン……それら全部をまとめてコンピュータだと理解してください。

コンピュータ操作イメージの中の「文章を書く」「計算をする」「メールを読む」「動画を見る」という行為は、コンピュータだけがあればできるものではありません。「文章を書く」ためのワープロソフト、「計算をする」ための表計算ソフト、「メールを読む」ためのメールソフトといった、**ソフトウェア** (software) がコンピュータに入っているように使えるようになります。スマートフォンでは、こうしたソフトウェアを「アプリ」という言葉で耳にすることが多いですが、アプリケーションソフトウェアを略した言葉ですので、本書ではすべて「ソフトウェア」と記載します。

これらのワープロソフトや表計算ソフトなどのソフトウェアは、すべてプログラムでできています。「そんなプログラムなんて作ったことはない」と思うかもしれませんが、それは誰かが作ったプログラムを使っているからです。プログラムを作ったことはなくても、コンピュータを使っていればプログラムを利用していないことはありません。ワープロソフトだけでなく、いつもお世話になっているWindowsなどのオペレーティングシステム(OS)やスマートフォンのAndroid OSやiOSも誰かが書いたプログラムなのです。

とすると、プログラムがないとコンピュータって何ができるの？と疑問に思うでしょう。そのとおり、プログラムがなければ、コンピュータは電気をむだに使って

るだけ、暖房器具ぐらい(電源を入れると熱を持つから)にしかなりません。

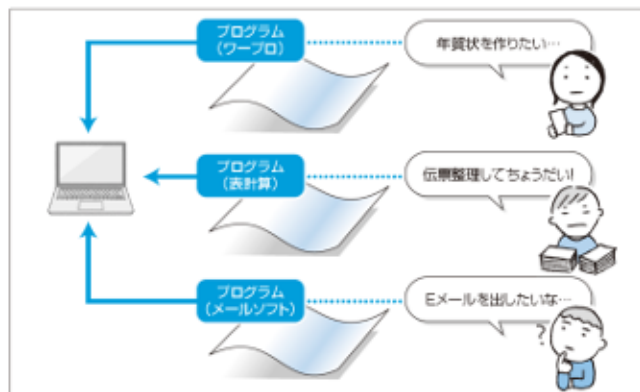
このように、プログラムがなければ、コンピュータはなにもできない「ただの箱」です。コンピュータとプログラムには、親密な関係が成り立っているのです。

1.1.1 コンピュータ、プログラム、人間の関係

コンピュータを使うときには、何か目的があるはずです。たとえば、ワープロで年賀状を作りたい、デジタルカメラで撮影した自分の写真をもっときれいに加工したい、など。そのようなときに、プログラムとコンピュータはそれぞれどんな役割を持っているのでしょうか？

ワープロで文章を書くときも写真の加工をするときも、コンピュータ自体には何もオプションをつけたり、変形させたり、合体させたりはしていません。ということは、わたしたちがコンピュータを使って行うことが変わっても、コンピュータ自身は何も変わっていないのです。では何が変わったのでしょうか？そう、使うソフトウェア(プログラム)と、それを使う人間にとっての操作方法が変わったわけです。コンピュータ、プログラム、人間の間には、次の図のような関係があります。

図 1-1
コンピュータ、プログラム、人間の関係



すなわち、人間が行いたいことはプログラムを通してコンピュータに伝えられるのです。このような、コンピュータに何をどのようにさせるかを記述したものを**プログラム** (program) と呼びます。

6.1 たくさんの値を記憶する必要性

6.1.1 実用的なプログラムを作るために必要なこと

前章までに紹介してきたプログラミング技法を使うことで、実にさまざまなプログラムを作ることができます。本章からは、もう一步進んで、より実用的なプログラムを作ることを考えていきましょう。

実用的なプログラムには、ワープロや表計算などいろいろなものがあります。これらのソフトウェアを使っているときを考えてみてください。たとえば、表計算ソフトでは、画面いっぱいに並んだ数字を足したり引いたり、集計したり、グラフにしたり……、とさまざまなことができるようになっていきます。このような機能を実現するには、どのようなプログラミング技術が必要になるのでしょうか？

- 足し算引き算などは、ここまでの章で学んだことでできます。
- 集計は、どのように集計するのかを式にまとめれば、ここまでの章で学んだ知識でできます。
- グラフにすることは、画面に絵を描く方法がわかれば、簡単にできます*。

あれ？ それなら、画面に表示すること以外は、もうできるのでしょうか？ 確かに、できるかもしれません。しかし、プログラムをどのように記述していくのかを考えてください。表計算ソフトでは、画面いっぱいに広がる、ときには画面よりもはるかに多い値をプログラムですべて記憶しておかなくてはなりません。もちろん、グラフを描くにも、すべての点の情報を記憶しておかなくてはなりません。

ワープロソフトを例に考えてみると、入力された文字はすべて記憶しておくようにプログラムを書かなければなりません。画像ソフトなどでは、画面上に描かれたすべての点が何色なのかや位置の情報などを記憶しておく必要があります。

*画面に絵を描く方法は、ハードウェアの仕様や、Windows、UNIXなどのOSの仕様、開発環境の違いによってさまざまに異なります。そのため、本書ではC言語によるグラフィックプログラムについては触れません。

Fig. 6-1
大量のデータや値を記憶する必要性



このように、実用的なプログラムを作るときには、大量のデータや値を記憶させて、うまく利用することが不可欠なのです。

忘れずに間違えずに「100個の適当な数を覚えておくこと」は、人間にとってはとてもできない作業です。しかし、コンピュータにとっては、1つの数を記憶することも10000の数を記憶することも、さほど違いはありません。記憶する場所さえあれば、忘れることや間違えることは絶対にありません。

このような、人間にとっては単純な作業だが、しかし人間では到底できないような大量の作業をしたときこそコンピュータに仕事をさせるのに最も適した場面といえるのです。

6.1.2 ここまでの記述方法での限界

データを記憶することは、コンピュータでいうならば、「何らかの値(数値)」を記憶することです。C言語では値を記憶させるときには、「変数」を使って記憶させます。では、大量の値を記憶させるときに、「値を記憶させていく変数」の宣言をここまで学んできた方法で記述するとどうなるでしょう？

たとえば、2000個の整数を記憶させる場所を用意する「変数宣言」を行うとすると次のようになります。

```
int a0, a1, a2, a3, a4, a5, a6, a7, a8, a9;
int a10, a11, a12, a13, a14, a15, a16, a17, a18, a19;
:
int a1990, a1991, a1992, a1993, a1994, a1995, a1996, a1997, a1998, a1999;
```

*ここで、変数名を「a1」からではなく、「a0」からにしています。これは、C言語で続き番号をあらわすときには、「0」から使用する」という慣例があるためです。

どうですか？ このように大量の変数宣言を行って、変数を用意しなくてはなりません。入力していくだけでも大変です。まして、どの変数がどの値をあらわしているのかわかるように、プログラムを見やすく書くのは至難の業です。

また、次のような原稿用紙に書かれている文字を、文字が書かれている原稿用紙の場所もわかるように記憶させておくにはどうすればよいでしょう。ここまでの記述方法を使ったのでは、次のように変数宣言をして文字を記憶させるしかありません。

```

printf("-----\n");
printf("%o: %F\n", &data[0], data[0]);
printf("%o: %F\n", &data[1], data[1]);
printf("%o: %F\n", &data[2], data[2]);

printf("-----\n");
printf("%o: [%c]\n", &data[0], data[0]);
printf("%o: [%c]\n", &data[1], data[1]);
printf("%o: [%c]\n", &data[2], data[2]);

printf("Input: ");
scanf("%d", &data[0]);
printf("%o: %d\n", &data[0], data[0]);

return(0);
}
    
```

実行結果
32ビット版での
実行結果

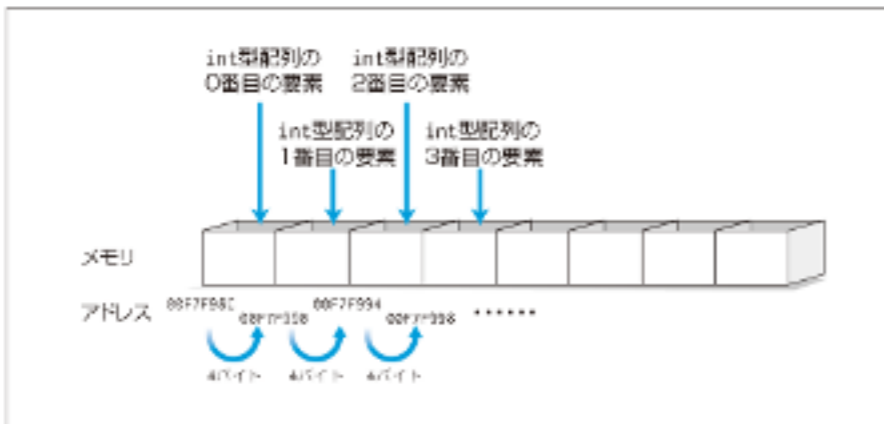
```

00F7F99C: 1
00F7F998: 2
00F7F994: 3
-----
00F7F99C: 1.100000
00F7F974: 2.200000
00F7F97C: 3.300000
-----
00F7F968: [A]
00F7F961: [3]
00F7F962: []
Input: 3
00F7F98C: 3
    
```

int型配列要素のアドレスと値の表示 (①の部分)

サンプルプログラムの①の部分は、int型配列要素についてアドレスと値を順番に表示しています。実行結果を見ると、16進数で4つずつ増えており、4バイトずつアドレスが増えていることが確認できます。Visual Studioの標準設定でintは4バイトの大きさで定義されているので、int型の配列としてメモリを利用した場合には、無駄なく連続したアドレスに値を入れて管理していることがわかります。

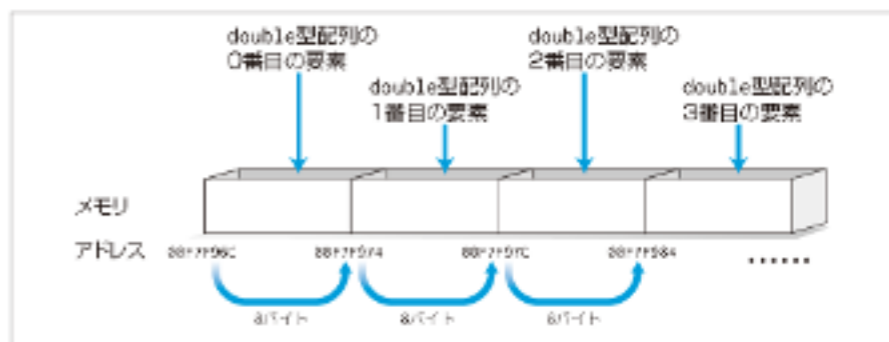
Fig. 13-3
int型配列の
アドレス管理



double型配列要素のアドレスと値の表示 (②の部分)

②の部分は、double型配列要素についてアドレスと値を順番に表示しています。結果を見ると、16進数で8つずつ増えており、8バイトずつアドレスが増えていることが確認できます。Visual Studioの標準設定でdoubleは8バイトの大きさで定義されているので、double型の配列としてメモリを利用した場合には、無駄なく連続したアドレスに値を入れて管理していることがわかります。

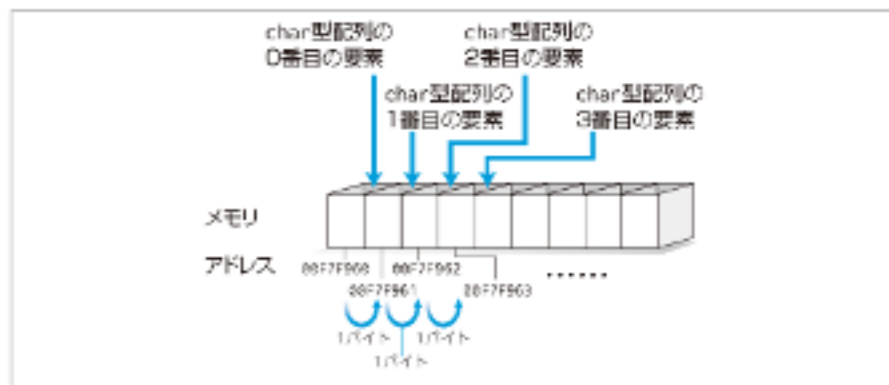
Fig. 13-4
double型配列の
アドレス管理



char型配列要素のアドレスと値の表示 (③の部分)

③の部分は、char型配列(文字列)要素についてアドレスと値を順番に表示しています。結果をみると、16進数で1つずつ増えており、1バイトずつアドレスが増えていることが確認できます。char型は1バイトの大きさで定義されているので、double型の配列としてメモリを利用した場合には、無駄なく連続したアドレスに値を入れて管理していることがわかります。

Fig. 13-5
char型配列の
アドレス管理



④の部分は、値を代入し直しても、配列の要素の場所(アドレス)は変化しないことを確認しています。あくまで値の変更であり、値を記憶している場所の変更はありません。



理解度チェック!

Q1 今まで利用してきたmainという記述は、の記述です。

Q2 以下は、関数functionを定義し、main関数で関数functionを2回利用しているプログラムです。各記述の意味を考えてみましょう。

```

#include <stdio.h>
// ア // 左に関数のプロトタイプ宣言を入れてください
// イ //
// ウ //
double function(double dnum, int num)
{
    int loop;
    double ans;

    ans = 1.0;
    for (loop = 0; loop < num; loop++)
    {
        ans = ans * dnum;
    }
    return ans;
}

int main()
{
    printf("3.0の2乗は: %f\n", function(3.0, 2));
    printf("3.5の3乗は: %f\n", function(3.5, 3));
    return(0);
}
    
```

ア: `// 左に関数のプロトタイプ宣言を入れてください`
 イ: `double function(double dnum, int num)`
 ウ: `{`
 エ: `for (loop = 0; loop < num; loop++)`
 オ: `return ans;`
 カ: `int main()`
 キ: `printf("3.0の2乗は: %f\n", function(3.0, 2));`
 `printf("3.5の3乗は: %f\n", function(3.5, 3));`
 `return(0);`

Q3 以下は、関数の受け渡しに関する確認プログラムです。①、②、③のパターンを実行したら、どのような結果が表示されるか考えてみましょう。

```

#include <stdio.h>

int funcA(int x, int y)
{
    x = 3; y = 5;
    return(0);
}
    
```

パターン①の関数定義

```

int funcB(int *x, int *y)
{
    *x = 3; *y = 5;
    return (0);
}

int funcC(int z[3])
{
    z[0] = 9; z[1] = 9; z[2] = 9;
    return (0);
}

int main()
{
    int a=2, b=4, c[3] = { 1,2,3 };
    printf("初期状態: %d-%d-%d %d %d\n", a, b, c[0], c[1], c[2]);

    funcA(a, b);
    printf("① %d-%d\n", a, b);

    funcB(&a, &b);
    printf("② %d-%d\n", a, b);

    funcC(c);
    printf("③ [%d %d %d]\n", c[0], c[1], c[2]);

    return(0);
}
    
```

パターン①の関数定義
 パターン②の関数定義
 パターン①の結果は?
 ク
 パターン②の結果は?
 ケ
 パターン③の結果は?
 コ

- 解答: Q1** 関数
Q2 ア: double型で関数functionを作る ウ: 引数を2つ受け取る関数として定義
 エ: 関数functionの定義 オ: double型で作った関数は、returnする値もdouble型
 カ: 関数呼び出し ※定義した引数の数と型を合わせることに! キ: 関数mainの定義
Q3 ク: ① 2-4 ケ: ② 3-5 コ: ③ [9-9-9]

補足解説 ①関数に変数を受け渡す場合、関数呼び出し側でも定義している関数でも単純に変数名を記載する。このときには、関数呼び出し側の変数は値がコピーされて、定義している関数の引数の変数に受け取られる。
 ②関数に変数を受け渡す場合、関数呼び出し側で引数である変数の前に「&」を付け、定義している関数の引数変数や関数の中の変数利用には「*」を変数名の前に付ける。このときには、関数を呼び出したときの変数と定義した関数の中の変数の値は同じものとして取り扱われる。
 ③関数に配列を渡す場合は、「&」や「*」を付けなくても②と同じように利用できる。



練習問題 5

Lesson 繰り直し処理で値を入力させ、平均値を求める

- 5-1 10人の試験結果(0~100の「整数」が入力されることのみを考えればよい)を入力させ、平均点(「実数」)を求めるプログラムを作成しなさい。

Lesson 10個の値の最大値を求める

- 5-2 10人の身長データを順にcm単位で入力したとき、一番大きな身長は何cmかを表示するプログラムを作成しなさい。

Lesson 任意値のデータの最大値と最小値を求める

- 5-3 買い物金額を計算し、整理する会計プログラムを次の仕様で作らなさい。

「税抜き単価」と「個数」を入力すると、次の出力が得られる。

- 合計金額(税抜きと税込み)
- 一番単価の安かったものの金額(税抜き)
- 一番単価の高かったものの金額(税抜き)

- ただし、すべての商品が課税対象で、税率10%とする(税込み単価: 単価 * 110/100として小数を利用しない計算方法を利用する)。
- 単価に0を入力したら処理を終了する。

Lesson 総合応用問題1

5-4 素数判定

正の整数を入力したとき、その数値が素数であれば「素数です」、素数でないなら「素数ではありません」と表示するプログラムを作成しなさい。

Lesson 総合応用問題2

- 5-5 あるインターネットプロバイダの料金は、以下のようになっている。

10時間以内	2000円
10~20時間まで1時間	210円
20時間を越えると1時間	205円

このとき、使用時間を入力すると、料金が表示されるプログラムを作成したい。加えて、一度計算しただけでプログラムが終了するのではなく、使用時間で0と入れたときにだけプログラムが終了し、それ以外は、何度も使用時間の入力、料金の表示ができるようにプログラムを作成しなさい。

C More Information

無駄をなくして効率のよいプログラムを作るには?②

変数の利用の無駄とは

4章末のコラムでは「処理の流れの無駄」についてお話ししました。もうひとつ、「変数の利用の無駄」とはどのようなことでしょうか?

それは、変数に名前をつけるときに、字数を短くすることではありません。コンパイルすればどんな長い変数の名前も機械的に別名が割り振られますので、**変数の名前は、むしろ一目見てわかるような、意味のある長めの名前のほうがよいです。**

変数の無駄とは、**必要ない変数宣言をしな**いということです。「なんだ、そんなことわかっている」と思うかもしれませんが、これが気をつけていないとうっかりと無駄な変数を用意してしまいがちなのです。

たとえば、2つのfor文の繰り直し処理を連続で書くときに、次のようにしていませんか?

```
for (loop1 = 0; loop1 < 10; loop1++)
{
    ...
}
```

①

```
for (loop2 = 0; loop2 < 20; loop2++)
{
    ...
}
```

②

①②の繰り直し処理は、それぞれ独立した繰り直し処理です。それぞれ独立して繰り直しをするのであれば、①が終わったときにloop1という変数はもう不要になることがよくあります。それなのに別にloop2という変数を②の繰り直し処理のために準備するのは無駄です。loop1を再利用すればよいのです。

このように、変数を利用するときには、「**必要なくなった変数は、他の処理で再利用することができる**」ということを意識すれば、無駄な変数の利用をもっと減らすことができます。

変数を用意するということは、コンピュータのメモリに値を入れる箱を用意することです。そのため、小さなプログラムを作っている間はさほど悪影響がありませんが、大きなプログラムを作るときには、小さな無駄の積み重ねによって、本当に必要な変数を