CONTENTS

第1部 C言語プログラミングの基本構造

More Information

1.1 コンピュータにとってプログラムってどんなもの？

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ぶ方が多いのではないでしょうか。では、画面を見つめて、キーボードをカタカタさせて、何をしているのでしょう？ 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い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。「文書を書く」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てはじめて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えません。**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いるだけ、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。

*「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ので、個人用コンピュータが登場したとき、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。

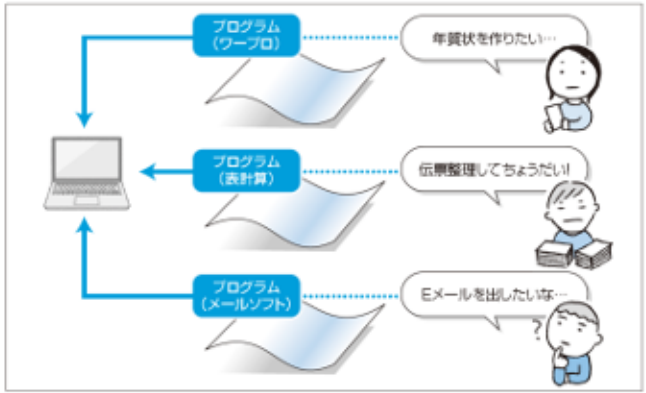
*計算を得意とするソフト。Microsoft Excelという商品が有名。

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ね。では何が変わったのでしょう？ 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。

Fig. 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、もう一歩進んで、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*。

*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クスプログラムについては扱いません。

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。

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い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。

*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。

```
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;
```

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がわかるように、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("%p: %F¥n", &ddata[0], ddata[0]);
    printf("%p: %F¥n", &ddata[1], ddata[1]);
    printf("%p: %F¥n", &ddata[2], ddata[2]);

    printf("----------¥n");
    printf("%p: [%c]¥n", &cdata[0], cdata[0]);
    printf("%p: [%c]¥n", &cdata[1], cdata[1]);
    printf("%p: [%c]¥n", &cdata[2], cdata[2]);

    printf("Input: ");
    scanf("%d", &idata[0]);
    printf("%p: %d¥n", &idata[0], idata[0]);

    return(0);
}
```

❷ ❸ ❹

実行結果
32ビット版での実行結果

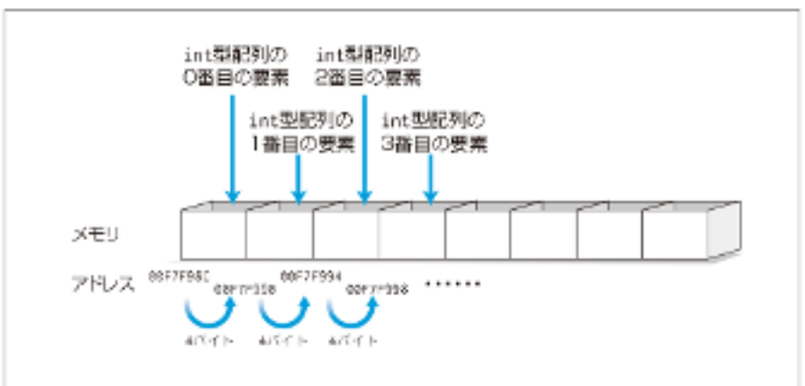
```
00F7F98C: 1
00F7F990: 2
00F7F994: 3
----------
00F7F96C: 1.100000
00F7F974: 2.200000
00F7F97C: 3.300000
----------
00F7F960: [A]
00F7F961: [B]
00F7F962: [C]
Input: 3
00F7F98C: 3
```

❶ ❷ ❸ ❹

int型配列要素のアドレスと値の表示（❶の部分）

サンプルプログラムの❶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。

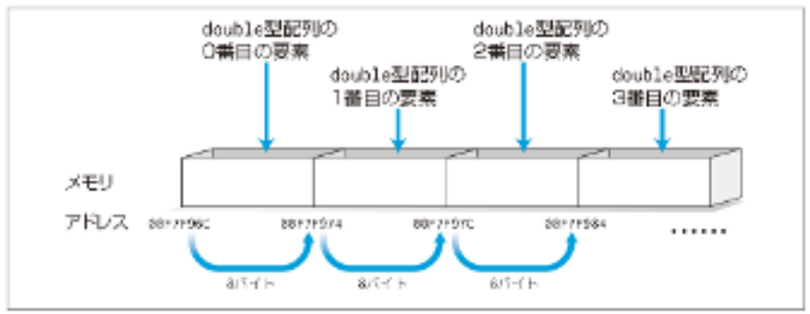
Fig. 13-3
int型配列のアドレス管理



double型配列要素のアドレスと値の表示（❷の部分）

❷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。

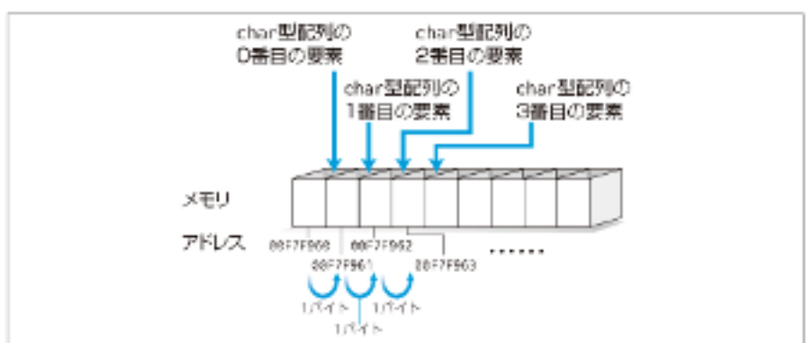
Fig. 13-4
double型配列のアドレス管理



char型配列要素のアドレスと値の表示（❸の部分）

❸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



❹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;          ← オ
}

                                    カ
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);
}
```

エ ／ キ

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);
}
```

パターン①の関数定義

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

パターン①の関数定義

```
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);
```

パターン①の結果は？ ク

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

パターン②の結果は？ ケ

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

パターン③の結果は？ コ

```
    return(0);
}
```

解答： Q1 関数

Q2 ア：double function(double dnum, int num);
イ：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！　キ：関数 main の定義

Q3 ク：①2-4　ケ：②3-5　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 5-1　繰り返し処理で値を入力させ、平均値を求める

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 5-2　10個の値の最大値を求める

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 5-3　任意個のデータの最大値と最小値を求める

買い物の金額を計算し、整理する会計プログラムを次の仕様で作りなさい。

「税抜き単価」と「個数」を入力すると、次の出力が得られる。

1. 合計金額（税抜きと税込み）
2. 一番単価の安かったものの金額（税抜き）
3. 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 5-4　総合応用問題1

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 5-5　総合応用問題2

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。

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」についてお話しました。もうひとつ、「変数の利用の無駄」とはどういうことでしょうか？

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。「なんだぁ、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文の繰り返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++)
{
    …
}
```
❶

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

❶❷の繰り返し処理は、それぞれ独立した繰り返し処理です。それぞれ独立して繰り返しをするのであれば、❶が終わったときにloop1という変数はもう不必要になっていることがよくあります。それなのに別にloop2という変数を❷の繰り返し処理のためだけ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、「**必要がなくなった変数は、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るはずです。

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