P.S.P.T. 計算物理研究会
C 言語および数値計算の基礎 の基礎

計算物理研究会

2011 年 4 月 10 日

はじめに

P.S.P.T. 計算物理研究会にようこそ．このテキストは新入部員を対象として作成しています．難しいと思ったところは無理せずマスターや他の部員に聞いてください．また，例文や課題のプログラムをどんどん打ち込んだり書き換えたりすることが上達の近道です．

メールや P.S.P.T. の wiki や掲示板でも質問を受付けています．

p.s.p.t.since1982@gmail.com
http://www23.atwiki.jp/pspt
文責 26 期 澤田耕介

目次

例題一覧

第 1 章

C 言語の基礎

1.1 C 言語の基礎

1.1.1 プログラムとは

コンピュータは人間が手順を指示してあげなければ何もしてくれません．コンピュータに手順を伝え指示を与えるものをプログラムと呼びます．パソコンやテレビゲーム，携帯電話，家電，自動車，自動販売機など，世の中のあらゆるところでコンピュータが使われており，コンピュータはプログラムを実行しています．

この講義では **C 言語**と呼ばれるプログラミング言語の基礎知識を身に付け，物理や数学の問題を解くプログラムを作成することを目的としています．

1.1.2 C 言語について

C 言語 (Language C) は 1972 年，AT&T ベル研究所のデニス・リッチーとケン・トンプソンによって作られたプログラミング言語です．プログラミング言語は無数にありますが，その中で世界中でもっとも普及していると言われています．数値計算だけではなくオペレーティングシステムやアプリケーション，ゲーム，組込み用途など幅広く使われています．そのため解説書やインターネット上の情報量も多いので学習する環境に恵まれています．また大学や企業，研究所からの需要も大きいため身につけておくと将来役に立つでしょう．なお今後，C 言語のことを略して C と呼ぶこともあります．

ではさっそく C 言語の例題を見てみましょう．

例題 1.1　C 言語のプログラムの例

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      printf("hello, world¥n");
6
7      return 0;
8  }
```

実行結果

初めてのプログラムです

これは実行結果のように「hello, world」と画面に表示させるプログラムのソースコードです．ソースコードとはコンピュータへの指示を記述した文章のことです．`#include<stdio.h>` から `}` までのたった 6 行のプログラムですが立派なプログラムです．

今回はまず画面に文字を表示させる方法を学びます．5 行目の `printf("hello, world¥n");` と書かれた行に注目します．その他の行の意味は今後徐々に理解して行きましょう．

なお，コンピュータ上では半角の \(バックスラッシュ) と ¥ (円マーク) は同じ記号として扱われるため，今後区別しません．[*1]

1.1.3 出力 printf()

C 言語のプログラムは，いろいろな機能を持った関数の組み合わせで記述されます．この **printf()** も関数のひとつで，**ダブルクォーテーション**(`" "`) の間の文字列を画面に表示する機能を持っています．この文字列のことを printf() の引数(argument) と呼びます．[*2]printf() に限らず，一般に関数に渡す情報を引数と呼びます．**ダブルクォーテーション**(`" "`) 内にはほぼ自由に文字と数字を書くことができます．しかし `"` `'` `¥` といった記号はプログラム内で意味を持つためそのままでは printf() 関数で表示できません．これらの記号を表示したり，特別な意味を持った制御文字を埋め込んだりするために用意されている文字をエスケープシーケンスと呼びます．

文字列を表示

```
printf("ここに文字列を記入します");
```

エスケープシーケンス

エスケープシーケンス	意味
¥n	改行
¥t	タブ
¥"	ダブルクォーテーションを表示
¥'	シングルクォーテーションを表示
¥¥	円マーク (環境によってはバックスラッシュ) を表示
¥0	ヌル (文字列の終端)

[*1] バックスラッシュと円マークは歴史的に文字コードが同じなのでコンピュータ内部では区別されません．/(スラッシュ) は全く違う記号です．

[*2] 「いんすう」と読んではいけません．

例題 1.2　printf() の実行例

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      printf("改行 ¥n します¥n");
6      printf("タブ ¥t です¥n");
7
8      return 0;
9  }
```

実行結果

```
改行
します
タブ　です
```

printf や return の前のまとまった空白は TAB キーで入力します．これはソースコードを読みやすくするための暗黙の約束事です．読みやすいソースコードを書くための様々な約束事については付録 A.1 で詳しく解説します．

1.1.4　C 言語の文法

1. 文字列を扱うとき以外，半角英数字でプログラムを記述します．
2. プログラムは原則的に上の行から下の行へと処理されます．
3. 好きなところに改行や半角スペース，タブを入れても構いません．このルールを活用して**他の人にも読みやすいソースコードを書きましょう**．ただし，単語の間にこういった区切り文字を入れてはいけません．また全角スペースは使用してはいけません．
4. 文の最後に必ず ；（**セミコロン**）が必要になります．上記のように自由な改行が許されるため，処理の区切りを明示する必要があるためです．
5. ソースコードの途中に，/* **文字列** */ や // **文字列** [*3]と挿入すると **文字列** の部分が無視されます．この部分にコメントを書くことで他の人にも読みやすいソースコードになります．

1.1.5　コンパイルと実行

コンピュータはソースコードを直接解釈して実行することはできません．そこでコンピュータが解釈できる形式に翻訳してあげる必要があります．その翻訳作業のことを**コンパイル**と呼びます．[*4]C 言語では以下のステップでプログラムを組み実行します．

1. テキストエディタでソースコードを記述

[*3] 実は // **文字列** でのコメントは C++ の命令であって C では規定されていません．C++ に対応したコンパイラ以外では使用できないことがあります．

[*4] コンパイルを行なうプログラムをコンパイラと呼びます．

2. ソースコードをコンパイル (およびリンク) し実行ファイルを作成
3. プログラムを実行

コンパイル後に作成されるファイルはオブジェクトファイルと呼びます．ソースコードは人間が理解できる文字で記述された**テキストファイル**であるのに対し，オブジェクトファイルはコンピュータが直接理解できる機械語で記述された**バイナリファイル**となります．binary とは二進数という意味の英単語です．オブジェクトファイルとライブラリと呼ばれるものを組合せることで実行ファイルが作成されます．この作業をリンク[*5]と呼びます．コンパイルとリンクを合わせてメイクと呼びます．また慣習的にメイクのことをコンパイルと呼ぶことが多いので混乱しないようにしましょう．

1.1.6 課題

課題 1　プログラムの実行結果がつぎのように表示されるプログラムを組んでみよう．

こんにちは

課題 2　プログラムの実行結果がつぎのように表示されるプログラムを組んでみよう．

自分の学生番号，名前
物理の好きな分野

[*5] リンクを行うプログラムのことをリンカと呼びます．

1.2　変数と代入，算術式の計算

今回は変数の使いかた，代数計算のしかたと計算結果の表示のしかたを学びます．

1.2.1　変数型と変数の宣言

領域に名前を付けて, 様々な値を代入できるようにしたものを**変数**といいます. 数を代入する際, **型**と呼ばれるものを宣言し, 変数の種類を明確にする必要があります．変数の宣言は処理の最初に行なわなければなりません．

変数の宣言

```
型　変数名;
```

int 型の変数 var を用意する場合は次のように宣言します．

```
int foo;
```

同じ型の変数であればカンマ演算子「,」で区切ることで複数個を同時に宣言することができます．

```
int foo, baa, boo;
```

型の種類

型	用途	サイズ
char	**文字**	**1 byte: -128〜127**
int	**符号付き整数**	**4 byte: -2,147,483,648〜2,147,483,647**
float	浮動小数点	4 byte: 1.17549e-38〜3.40282e+38
double	**倍精度浮動小数点**	**8 byte: 2.22507e-308〜1.79769e+308**

数値計算では桁数の多い実数を扱うため，double 型がよく使われます．回数を数え上げる用途など，整数以外用いない場合は int 型を使います．また変数型のサイズは環境によって変わることがあります．

例題 1.3　変数の宣言

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      int a;
6      double b;
7      char c;
8
9      return 0;
10 }
```

5,6,7 行で変数を定義しました．**このプログラムを実行しても何も表示されません．**

1.2.2 値の代入

変数に値を代入するときは，= **(代入演算子)** を用います．右辺の値や計算結果を左辺の変数に代入します．**数学のイコール記号 = とはまったく意味が違います．左辺←右辺**と考えると理解しやすいでしょう．代入する値に対応する型を宣言します．

値の代入

変数名 = 値;

ここで注意すべきは，**=は"等しい"という意味ではなく，"代入"を意味する**ものです．

変数の宣言時に初期値を代入することもできます．例えば double 型の変数 pi の宣言と同時に 3.141592 を代入したい場合は次のようにします．

```
double pi = 3.141592;
```

文字を代入する場合は次のように char 型[*6]で変数を宣言し，代入式の右辺で代入する文字を ' **(シングルクォーテーション)** で挟みます．

```
char c;
c = 'A';
```

例題 1.3 のソースコードに少し付け加えて値の代入をしてみましょう．

例題 1.4 変数の宣言と代入

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      int a;
6      double b;
7      char c;
8
9      a = 10;
10     b = 3.14;
11     c = 'A';
12
13     return 0;
14 }
```

9-11 行で a, b, c にそれぞれ整数，浮動小数，文字を代入しました．**実行してもまだ何も表示されません．**

変数名は英数字と_**(アンダーバー)** を自由に組み合わせて付けることができます．ただし，数字や_ではじめてはいけません．(例：123abc , _xyz はダメ) また int や for などの C 言語で用意されているコマンド (予約語) を変数名に使うことはできません．

[*6] 「キャラクター型」と読みます．たまに「チャー型」と読む人もいます．

1.2.3 出力 printf()

第 1.1 節 でディスプレイ (**標準出力**) への文字の表示に **printf()** を学びましたが，今回は変数の値も表示できるようにしましょう．

変数の値を表示
printf(”文字列 フォーマット指定子 文字列”, 変数名);

このように記述することによってフォーマット指定子のところに変数の中身が代入されます．

フォーマット指定子の種類

型	用途
%c	**文字**
%d	**符号付き 10 進整数**
%f	**10 進の浮動小数点数**
%e	%f の指数部付き表記
%g	%f か %e かどちらか短い方
%s	文字列

例題 1.4 のソースコードにさらに 13-16 行を付け加えてみましょう．

例題 1.5　printf() の実行例

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      int a;
6      double b;
7      char c;
8
9      a = 10;
10     b = 3.14;
11     c = 'A';
12
13     printf("変数␣a␣に代入した値は␣%d␣です ¥n", a);
14     printf("変数␣b␣に代入した値は␣%f␣です ¥n", b);
15     printf("変数␣c␣に代入した値は␣%c␣です ¥n", c);
16     printf("%c␣のアスキーコードは␣%d␣です ¥n", c, c);
17
18     return 0;
19 }
```

実行結果

```
変数 a に代入した値は 10 です
変数 b に代入した値は 3.140000 です
変数 c に代入した値は A です
A のアスキーコードは 65 です
```

char 型の変数には文字 (ASCII 文字) が一文字入っていますが，実際には ASCII コード表と対応した数字 (コード) が格納されています．そのため %d で整数として表示させることができます．

1.2.4 算術式の計算

C 言語の式は普通の代数式に則っています．コンピュータに加減乗除させるのに必要な演算子を**算術演算子**と呼びます。

算術演算子の種類

演算子	用途
+	加算
-	減算
*	乗算
/	除算
%	剰余

剰余は，除算の余りを求める算術演算子です．整数型 (int 型) 以外には使えません．浮動小数点型での除算では小数点以下まで計算して余りが出ないためです．*，/，%，は +，-より優先されますが，() を使うことにより計算順を変えることができます．このあたりも数学の代数式と同じです．

また，先程から代入演算子 = が数学のイコール記号と意味がまったく違うとくり返していますが，次の例を見ればその理由がわかると思います．

例題 1.6　代入演算子としてのイコール記号

```
1  #include <stdio.h>
2  int main(void)
3  {
4      int a;
5
6      a = 10;
7      printf("a␣の中身は %d¥n", a);
8
9      a = 9 * 6;
10     printf("a␣の中身は %d¥n", a);
11
12     a = a + 100;
13     printf("a␣の中身は %d¥n", a);
14
15     a = a + a * a;
16     printf("a␣の中身は %d¥n", a);
17
18     return 0;
19 }
```

実行結果

```
a の中身は 10
a の中身は 54
a の中身は 154
a の中身は 23870
```

高校の力学の問題を解いてみよう．

初速度 100[m/s] で運動する物体が 20[s] 後に速度 8800[m/s] になった．$x = v_0 t + 1/2at^2$ の式より，この間に進んだ距離と加速度を求める．

例題 1.7 計算問題

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      double a, v1, v2, t, x;
6      v1 = 100.0;
7      v2 = 8800.0;
8      t = 20.0;
9
10     a = (v2 - v1) / t;    /* a 加速度 */
11     x = v1 * t + (1.0 / 2.0) * a * t * t; /* x 移動距離 */
12
13     printf("加速度 ¥t%f␣[m/s^2]¥n", a);
14     printf("距離 ¥t%f␣[m]¥n", x);
15
16     return 0;
17 }
```

実行結果

```
加速度 435.000000 [m/s^2]
距離   89000.000000 [m]
```

1.2.5 入力 scanf()

scanf() はキーボード (**標準入力**) から入力された値を変数に代入する関数です．printf() と違い，変数名には&(アンパサンド) をつける必要があります．[*7]

キーボードから値を入力

scanf(”フォーマット指定子”, &変数名);

フォーマット指定子の種類

型	**用途**
%c	**文字**
%d	**符号付き 10 進整数**
%f	浮動小数点
%lf	**倍精度浮動小数点**
%s	文字列

数値計算でよく使われる **double 型**変数は，**printf()** では **%f**，**scanf()** では **%lf** で扱うため注意が必要です．

[*7] 現段階では少し高度な説明になりますが，**scanf()** には変数の値ではなく変数のアドレスを渡す必要があるため，アドレスを取り出す演算子&を変数名に付けます．ポインターについて理解するまではあまり気にしなくて大丈夫です．

例題 1.8　scanf() の実行例

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      int a, b, c, d;
6
7      printf("整数を␣2␣つ入力してください ¥n");
8      scanf("%d␣%d", &a, &b);
9
10     c = a + b;
11     printf("合計は␣%d␣です ¥n", c);
12     printf("積は␣%d␣です ¥n¥n", a * b);
13
14     c = a / b;
15     d = a % b;
16     printf("商は␣%d␣余りは %d␣です ¥n", c, d);
17
18     return 0;
19 }
```

実行結果

```
整数を 2 つ入力してください
10 3 ←キーボードから入力
合計は 13 です
積は 30 です

商は 3 余りは 1 です
```

1.2.6　課題

課題 3　例題 1.7 のプログラムを **scanf()** を使ってキーボード (標準入力) から v1，v2，t を与えられるように改造してみよう．

課題 4　弧度法 (ラジアン) から度数法に変換するプログラムを書いてみよう．次に度数法から弧度法 (ラジアン) に変換するプログラムを書いてみよう．(近似値でよい)

ヒント $180° = \pi[rad], \quad 1° = \pi[rad]/180$

1.3 C 言語と関数

C のプログラムは必ず 1 つ以上の関数で構成されます．関数とはプログラムのどこからでも呼びだすことができる独立した小さなプログラムです．プログラム中で何回も行なわれる一連の処理 (ルーチン) を独立した部品にして再利用できるようにしたものです．関数に引数(ひきすう)を与えるとなにかしらの仕事をしてくれます．例えば printf() 関数に引数を与えると画面に文字を表示してくれます．scanf() 関数ならキーボードから入力された文字を変数に代入してくれます．main() 関数も同様に Windows や UNIX などの OS から引数をもらい，処理結果を OS に返します．

1.3.1 数学関数と math.h

プログラムの冒頭に `#include <stdio.h>` [*8]と入力していたのは printf() や scanf() などの関数を使用するためです．今挙げた 2 つの関数は stdio.h というファイル内で定義されていて，それをプログラムに組込むことによって使用可能となります． `#include <math.h>` とし，`math.h` を読み込むことで，三角関数や指数関数，対数関数などの数学的な動作をする関数が使用できるようになります．

`stdio.h` や `math.h` のようなファイルを**ヘッダファイル**と呼びます．ヘッダファイルをプログラムに組込むことを**インクルード**と呼びます．用途に応じたヘッダファイルをインクルードすることで様々な関数が利用可能になります．

math.h と数学関数

関数	返り値	定義域
`double sin (double arg);`	**arg[rad] の正弦**	
`double cos (double arg);`	**arg[rad] の余弦**	
`double tan (double arg);`	**arg[rad] の正接**	
`double asin (double arg);`	arg の逆正弦	$-1 \leq arg \leq 1$
`double acos (double arg);`	arg の逆余弦	$-1 \leq arg \leq 1$
`double atan (double arg);`	arg の逆正接	
`double atan2 (double y, double x);`	y/x の逆正接	
`double sinh (double arg);`	arg の双曲線正弦	
`double cosh (double arg);`	arg の双曲線余弦	
`double tanh (double arg);`	arg の双曲線正接	
`double exp (double arg);`	**自然対数 e の arg 乗**	
`double log (double num);`	**num の自然対数**	$0 < num$
`double log10 (double num);`	num の常用対数	$0 < num$
`double pow (double base, double exp);`	**base の exp 乗**	i.$base > 0$, ii.$base = 0$ かつ $exp > 0$ iii.$base < 0$ かつ $exp =$ 整数
`double sqrt (double num);`	**num の平方根**	
`double fabs (double num);`	**num の絶対値**	
`double ceil (double num);`	num を切り上げ	
`double floor (double num);`	num を切り捨て	

[*8] stdio.h は Standard I/O header の略です．I/O とは Input/Output のことです．「標準入出力ヘッダ」の意味となります．読み方は「スタンダード・アイ・オー・エイチ (またはヘッダ) です．「スタジオエイチ」とは読みません．

今まで触れていませんでしたが，**関数の引数と返り値の型は決められています．**sin() 関数の場合，引数として double 型の角度を関数に渡し，関数の呼び出し元に double 型で正弦が返されます．お馴染の printf() 関数の場合は引数に char 型の制御文字列，返り値は int 型となっています．

例題 1.9　sin() 関数

```
1  #include <stdio.h>
2  #include <math.h>
3  int main(void)
4  {
5      double a;
6
7      a = 3.14159;
8      a = sin(a);
9      printf("sin␣(pi)␣は␣␣␣%f␣です ¥n", a);
10
11     a = 3.14159;
12     a = a / 2;
13     a = sin(a);
14     printf("sin␣(pi/2)␣は␣␣␣%f␣です ¥n", a);
15
16     return 0;
17 }
```

実行結果

```
sin (pi) は    0.000003 です
sin (pi/2) は    1.000000 です
```

例題 1.10 math.h の主な関数

```
1  #include <stdio.h>
2  #include <math.h>
3  int main(void)
4  {
5      double a = 2.0;
6
7      printf("sin␣(pi/2)␣は␣␣␣%f␣です ¥n", sin(3.14159 / 2));
8      printf("√ %f␣は␣%f␣です ¥n", a, sqrt(a));
9      printf("|%g|␣は␣%g␣です ¥n", -a, fabs(-a));
10     printf("2^10␣は␣␣%g␣です ¥n", pow(2, 10));
11     printf("e^1␣は␣␣%g␣です ¥n", exp(1.0));
12     printf("ln␣e␣は␣%f␣です ¥n", log(exp(1.0)));
13
14     return 0;
15 }
```

実行結果

```
sin (pi/2) は   1.000000 です
√ 2.000000 は 1.414214 です
|-2| は 2 です
2^10 は  1024 です
e^1 は  2.71828 です
ln e は 1.000000 です
```

1.4　条件文，関係演算子と論理演算子

数学でいう場合分けのように，C 言語でも**条件文**あるいは**選択文**と呼ばれる構文により．ある条件の元なら処理 A に，別の条件なら処理 B にといった記述をすることができます．例えば 2 次方程式の解の判別を行うプログラムは以下のようなステップで作ることができます．

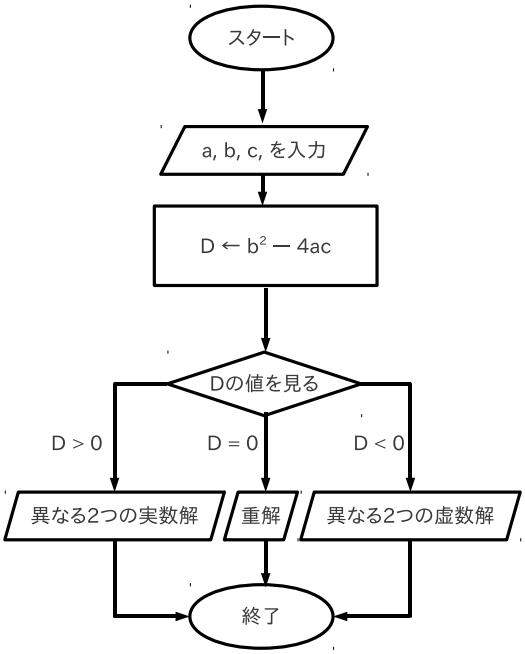

図 1.1　2 次方程式の判別フローチャート

この節では最終的に 2 次方程式の解の判別プログラムを C 言語で作成することを目標とします．

1.4.1　関係演算子と論理演算子

関係演算子は，2 つの数値を比較して**真なら 1，偽なら 0** を返す演算子です．論理演算子は，論理演算 (論理積 AND，論理和 OR，否定 NOT) を行う演算子です．これらを組み合せて**条件式**に使います．

関係演算子

演算子	意味
>	より大きい
>=	以上
<	より小さい
<=	以下
==	等しい
!=	等しくない

論理演算子

演算子	意味
&&	論理積 AND
||	論理和 OR
!	否定 NOT

演算子の評価

式	評価
`100 < 200`	真 1
`'A' == 'B'`	偽 0
`'A' != 'B'`	真 1
`100 < 200 && 'A' == 'B'`	偽 0
`100 < 200 || 'A' == 'B'`	真 1
`!0`	真 1
`!1`	偽 0
`!5`	偽 0

論理演算子の真理表

p	q	p&&q	p||q	!p
0	0	0	0	1
0	1	0	1	1
1	1	1	1	0
1	0	0	1	0

1.4.2 条件文 if()

条件式が真のときと偽のときとで違う処理を選択する構文を **if 文**と呼びます.

```
if（条件式）{
    条件式が真のときの処理;
}
```

処理が 1 行で済む場合，以下のようにも書けます.

```
if（条件式）条件式が真のときの処理;
```

例題 1.11　if 文 条件真

```
#include <stdio.h>

int main(void)
{
    int a = 30;

    if (a >= 20) {
        printf("a は 20 以上です¥n");
    }

    printf("終了します ¥n");
    return 0;
}
```

実行結果

```
a は 20 以上です
終了します
```

例題 1.12　if 文 条件偽

```
#include <stdio.h>

int main(void)
{
    int a = 10;

    if (a >= 20) {
        printf("a は 20 以上です¥n");
    }

    printf("終了します ¥n");
    return 0;
}
```

実行結果

```
終了します
```

else を用いると偽の場合の処理をさせることができます.

```
if（条件式）{
    条件式が真のときの処理;
}
else {
    条件式が偽のときの処理;
}
```

例題 1.13 if と else

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      int a = 10;
6
7      if (a >= 20) {
8          printf("a は 20 以上です¥n");
9      }
10     else {
11         printf("a は 19 以下です¥n");
12     }
13
14     printf("終了します ¥n");
15     return 0;
16 }
```

実行結果

```
a は 19 以下です
終了します
```

さらに **else if** を用いて条件分岐をいくらでも増やすこともできます.

```
if(条件式 1) {
    条件式 1 が真のときの処理;
}
else if (条件式 2) {
    条件式 1 が偽 かつ 条件式 2 が真のときの処理;
}
else if (条件式 3) {
    条件式 1 が偽 かつ 条件式 2 が偽 かつ 条件式 3 が真のときの処理;
}
else {
    条件式 1〜条件式 3 が偽のときの処理;
}
```

例題 1.14　else if

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      int a = 10;
6
7      if (a >= 20) {
8          printf("a は 20 以上です¥n");
9      }
10     else if (a >= 5) {
11         printf("a は 10 以上 20 未満です¥n");
12     }
13     else {
14         printf("a は 4 以下です¥n");
15     }
16
17     printf("終了します ¥n");
18     return 0;
19 }
```

実行結果

```
a は 10 以上 20 未満です
終了します
```

また，{} 内の固まりをブロックといい，このブロック内にまた条件などのブロックを書くことを入れ子，またはネストといいます．

```
if（条件式 1）{
    if（条件式 2）{
        if（条件式 3）{
            条件式 1 が真 かつ 条件式 2 が真 かつ 条件式 3 が真のときの処理;
        }
    }
}
```

例題 1.15 if 文のネスト

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      int a = 40;
6
7      if (a >= 20) {
8          printf("a は 20 以上です¥n");
9
10         if (a >= 30) {
11             printf("a は 30 以上です¥n");
12         }
13         else {
14             printf("a は 20 以上 30 未満です¥n");
15         }
16     }
17
18     else {
19         printf("a は 19 以下です¥n");
20     }
21
22     printf("終了します ¥n");
23     return 0;
24 }
```

実行結果

a は 20 以上です
a は 30 以上です
終了します

これらの例題の数値や条件式をいろいろ変えて試してみてください．条件文は次節の繰り返し文とならび，C 言語の中でもっとも重要な文法です．

さて，準備が整ったので，2 次方程式の解の判別プログラムを C で作成してみましょう．

例題 1.16　2 次方程式の解の判別

```c
#include <stdio.h>

int main(void)
{
    double a, b, c, d;

    printf("a␣b␣c␣を入力してください ¥n");
    scanf("%lf␣%lf␣%lf", &a, &b, &c);

    d = b * b - 4 * a * c;
    printf("(%gx^2)␣+␣(%gx)␣+␣(%g)␣=␣0␣は␣", a, b, c);

    if (d > 0) {
        printf("異なる 2つの実数解を持ちます ¥n");
    }
    else if (d == 0) {
        printf("重解を持ちます ¥n");
    }
    else {
        printf("異なる 2つの虚数解を持ちます ¥n");
    }

    return 0;
}
```

実行結果

```
a b c を入力してください
1 3 2 ←キーボードから入力
(1x^2) + (3x) + (2) = 0 は 異なる 2 つの実数解を持ちます
```

1.4.3　課題

課題 5　数字を入力し，入力された数字が偶数か奇数か判別して画面に表示するプログラムを書いてみよう．

課題 6　1 から 100 までの数を表示する．ただし 3 の倍数のときは数の代わりに「Fizz」とプリントするプログラムを書いてみよう．

課題 7　さらに 5 の倍数のときは「Buzz」，3 と 5 の倍数のときは「FizzBuzz」とプリントするプログラムを書いてみよう．

1.5 繰り返し文

今回は**繰り返し文**を学びます．繰り返し文では，条件式が真である限り同じ処理を繰り返し (ループ) ます．if などの条件文と合わせて**制御文**と呼ばれるプログラミングにおいて大変重要な構文です．

1.5.1 インクリメント · デクリメント

繰り返し文の前に**インクリメント**と**デクリメント**について説明します．C 言語ではある変数に 1 を足したり，ある変数から 1 を引いたりすることがよくあります．これはもちろん

```
i = i + 1;
i = i - 1;
```

のように書くことができますが,

```
i++;
i--;
```

と書くこともできます．++ をインクリメント演算子，--をデクリメント演算子と呼びます．C 言語に慣れてくると後者の書き方の方が読みやすくなります．また，コンピュータ内部の処理の違いにより後者の方が高速で動作します．

例題 1.17 インクリメントとデクリメント

```c
#include <stdio.h>
int main(void)
{
    int i = 10;
    int j = 100;

    i++; /* インクリメント */
    printf("i␣=␣%d¥n", i);

    j--; /* デクリメント */
    printf("j␣=␣%d¥n", j);

    return 0;
}
```

実行結果

```
i = 11
j = 99
```

ほぼ同じ意味で

```
++i;
```

```
--i;
```

と書くこともできますが，少し動作が違ってきます．現時点ではあまり重要なことではないので読み飛ばして問題ありません．興味のある人だけ読んでください．

インクリメント・デクリメント演算子を変数の後に置くと (ex i++)，式の中でその変数が使用された後にインクリメント・デクリメントされます．変数の前に置くと (ex ++i) インクリメント・デクリメントされてから変数が使用されます．ちょっとややこしいですが，通常は i++ と i--を使用すればいいでしょう．

例題 1.18　i++ と ++i の違い

```
#include <stdio.h>
int main(void)
{
    int i, j;

    i = 5;
    j = i++;
    printf("i=%d␣j=%d¥n", i, j);

    i = 5;
    j = ++i;
    printf("i=%d␣j=%d¥n", i, j);
    return 0;
}
```

実行結果

```
i=6 j=5
i=6 j=6
```

1.5.2　繰り返し for

ある処理を一定の回数繰り返したい場合に使用するのが for 文です．

```
for（初期化式；条件式；増分）{
    条件式が真のときの処理；
}
```

処理が 1 行で済む場合，ブロックを使わずに以下のようにも書けます．

```
for（初期化式；条件式；増分）条件式が真のときの処理；
```

1 から 10 までの和を求めるプログラムです．変数 i と a がどのように変化するのかよく追ってください．

例題 1.19 for 文で 1 から 10 までの和を求める

```
#include <stdio.h>

int main(void)
{
    int i;
    int a = 0;

    for (i = 0; i <= 10; i++) {
        a = a + i;
        printf("%d␣%d¥n", i, a);
    }

    return 0;
}
```

実行結果

```
0 0
1 1
2 3
3 6
4 10
5 15
6 21
7 28
8 36
9 45
10 55
```

次の例では初期化式で `i = 0` とし，条件式で `i <= 16` である限り for 以下のブロック内の処理を繰り返します．処理が終わるごとに `i` が 1 ずつ増えます．簡単に言うと，処理を 16 回繰り返してからループを抜けると言うことです．

例題 1.20　for 文で 2 のべき乗を求める

```
1  #include <stdio.h>
2  #include <math.h>
3
4  int main(void)
5  {
6      int n, An, Sn = 0; /* n 項数, An 第 n 項, Sn 第 n 項までの和 */
7
8      printf("n¥t2␣の␣n␣乗␣¥t␣数列の和␣¥n");
9      for (n = 1; n <= 16; n++) {
10         An = pow(2 ,n); /* 2 の n 乗を計算 */
11         Sn = Sn + An;
12         printf("%d¥t%d¥t%d¥n", n, An, Sn);
13     }
14
15     return 0;
16 }
```

実行結果

```
n   2 の n 乗      数列の和
1   2    2
2   4    6
3   8    14
4   16   30
5   32   62
6   64   126
7   128  254
8   256  510
9   512  1022
10  1024     2046
11  2048     4094
12  4096     8190
13  8192     16382
14  16384    32766
15  32768    65534
16  65536    131070
```

この例題を繰り返し文を使わずに書くと，かなり大変だということがわかると思います.

1.5.3　代入演算子

例題 1.20 に

```
Sn = Sn + An;
```

という行がありましたが，次のように書くこともできます．

```
Sn += An;
```

このような演算子を**代入演算子**と呼びます．なお，今まで使っていた=も代入演算子です．

代入演算子の種類

演算子と代入式	意味
n_1 += n_2	$n_1 + n_2$ を n_1 に代入
n_1 -= n_2	$n_1 - n_2$ を n_1 に代入
n_1 *= n_2	$n_1 * n_2$ を n_1 に代入
n_1 /= n_2	n_1/n_2 を n_1 に代入
n_1 %= n_2	n_1/n_2 の剰余を n_1 に代入

1 から 10 までの和を求める例文を代入演算子 +=を使って書き直してみましょう．当然実行結果は同じになります．

例題 1.21　for 文で 1 から 10 までの和を求める (代入演算子を用いて)

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      int i;
6      int a = 0;
7
8      for (i = 0; i <= 10; i++) {
9          a += i;
10         printf("%d␣%d¥n", i, a);
11     }
12
13     return 0;
14 }
```

実行結果

```
0 0
1 1
2 3
3 6
4 10
5 15
6 21
7 28
8 36
9 45
10 55
```

1.5.4 繰り返し while

for 文は繰り返し回数が決っている場合に使いますが，while 文は繰り返し回数は決めずに条件式だけ与えられる場合に使います．

```
while（条件式）{
    条件式が真のときの処理;
}
```

処理が 1 行で済む場合，ブロックを使わずに以下のようにも書けます．

```
while（条件式）条件式が真のときの処理;
```

while 文で 1 から 10 までの和を求めてみます．

例題 1.22 while 文で 1 から 10 までの和を求める

```
#include <stdio.h>

int main(void)
{
    int i;
    int a = 0;

    i = 0;
    while (i <= 10) {
        a = a + i;
        printf("%d␣%d¥n", i, a);
        i++;
    }

    return 0;
}
```

実行結果

```
0 0
1 1
2 3
3 6
4 10
5 15
6 21
7 28
8 36
9 45
10 55
```

for 文と while 文とはほぼ同じことができますが，用途によって使い分けることでプログラムが分かりやすくなります．例えば，例題 1.20 では for 文を使い 2 の 16 乗まで計算しましたが，2 の何乗で 10,000 を超えるのかを知りたい場合など，ループ回数がわからない場合には while 文の方が適しています．

例題 1.23　while 文で 2 のべき乗を調べる

```
1  #include <stdio.h>
2  #include <math.h>
3
4  int main(void)
5  {
6      int n=0, An=0, Sn=0; /* n 項数, An 第n 項, Sn 第n 項までの和 */
7
8      printf("n¥t2 の n 乗¥t 数列の和¥n");
9      while (An < 10000) {
10         n++;
11         An = pow(2, n); /* 2のn 乗を計算 */
12         Sn += An;
13         printf("%d¥t%d¥t%d¥n", n, An, Sn);
14     }
15     printf("n␣=␣%d␣で␣10,000を超えます．¥n", n);
16
17     return 0;
18 }
```

実行結果

```
n   2 の n 乗      数列の和
1   2    2
2   4    6
3   8    14
4   16   30
5   32   62
6   64   126
7   128  254
8   256  510
9   512  1022
10  1024     2046
11  2048     4094
12  4096     8190
13  8192     16382
14  16384    32766
n = 14 で 10,000 を超えます．
```

1.5.5　繰り返し do while

while 文は条件式が偽の場合，処理を 1 度も行ないません．**do while** 文を使うとまず処理を 1 回行なってから条件式を評価します．

```
do {
    条件式が真のときの処理;
} while (条件式)
```

処理が 1 行で済む場合，ブロックを使わずに以下のようにも書けます．

```
do 条件式が真のときの処理; while (条件式);
```

例題 1.24 do while 文の使用例

```
1  #include <stdio.h>
2  int main(void)
3  {
4      int i = 1;
5
6      do { /* 最低 1度は実行される */
7          printf("処理 1␣i=%d¥n", i);
8          i++;
9
10     } while (i < 0);
11
12     do { /* 最低 1度は実行される */
13         printf("処理 2␣i=%d¥n", i);
14         i++;
15     } while (i < 5);
16
17     return 0;
18 }
```

実行結果

```
処理 1 i=1
処理 2 i=2
処理 2 i=3
処理 2 i=4
```

1.5.6 for と while の使い分け

for 文と while 文は書き方次第で同じ意味を持たせることができますが，用途によって使い分けるべきです．一般に繰り返し回数があらかじめわかっている場合は for 文，わかっていない場合は while 文で書けばいいでしょう．こうすると他の人が読んでも理解しやすいプログラムになります．

例題 1.25　for 文に向いた処理

```
#include <stdio.h>
int main(void)
{
    int i, n = 0;

    for (i = 1; i < 11; i++) { /* 1から 10までの和を計算 */
        n += i;
    }
    printf("%d¥n", n);
    return 0;
}
```

実行結果

```
55
```

例題 1.26　while 文に向いた処理

```
#include <stdio.h>
int main(void)
{
    char a;

    printf("A と入力するまで繰り返す¥n");
    while ( a != 'A') { /* キーボードから A と入力されるまで繰り返す */
        scanf("%c", &a);
    }

    return 0;
}
```

実行結果

```
A と入力するまで繰り返す
a ← 以降 キーボードから入力
b
1
B
A
```

1.5.7　課題

課題 8　$S = \sum_{n=1}^{100} n$ を求めてみよう.

課題 9　例題 1.20 のプログラムを pow() 関数を使わずに書いてみよう．

課題 10　$e = \sum_{n=0}^{\infty} \frac{1}{n!}$ を求めてみよう．コンピュータでは計算を無限回繰り返すことはできないので適当に打ち切る．

課題 11　$f(x) = x^2$ の $f(x)$ を $-2 \leq x \leq 2$ の範囲で求めてみよう．x の刻み幅を 0.1 とする．出力形式は次のようにする．

1.6 配列

1.6.1 1 次元配列

配列(Array) とは変数をまとめてリストにしたものです．関連性のある複数のデータをまとめて扱いたい場合などに使用します．宣言のしかたは変数の場合とほぼ同じですが，配列のサイズ (配列要素の数) を指定する必要があります．配列の各要素が 1 つの変数と同じようなものになります．

配列の宣言
型　配列名 [サイズ];

要素数 100 の int 型の配列 array を用意する場合は次のように宣言します．

```
int array[100];
```

同じことを変数でやろうとすると大変です．

```
double array001;
double array002;
...
double array100;
```

配列の要素にアクセスする場合には配列に**添え字 (インデックス)** を付け配列要素を指定します．**配列の要素は 0 からサイズ-1** となります．上の例では要素を 100 個作りましたが，要素の番号は 0 から 99 となります．array の 3 番目の要素に 100 を代入する場合は次のようにします．

```
array[2] = 100;
```

例題 1.27　1 次元配列

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      int a[4];
6
7      a[0] = 0;
8      a[1] = 10;
9      a[2] = 20;
10     a[3] = 30;
11
12     printf("%d␣%d␣%d␣%d¥n", a[0], a[1], a[2], a[3]);
13
14     return 0;
15 }
```

実行結果

```
0 10 20 30
```

配列の宣言時に初期値を与える場合は次のようにします.

```
int array[5] = {134, 12, 34, 4332, 243};
```

例題 1.28 1 次元配列 宣言と同時に初期値代入

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      int a[4] = {0, 10, 20, 30};
6
7      printf("%d␣%d␣%d␣%d¥n", a[0], a[1], a[2], a[3]);
8
9      return 0;
10 }
```

実行結果

```
0 10 20 30
```

配列は変数とほとんど同じ使い方ができますが，配列を別の配列にまるごと代入することはできません．for 文を使うなどして要素をひとつずつ代入する必要があります.

また，コンパイル時に配列の添え字の範囲はチェックされません．存在しない要素にアクセスした場合，プログラムや OS がクラッシュしたり不安定な状態になったりする可能性があります.

配列へは要素番号を使ってアクセスできるので繰り返し文などと組み合わせると記述が簡単になります．配列を使わない場合と使う場合の比較をベクトルの足し算の例で見てみましょう.

例題 1.29　ベクトルの足し算 配列を使わない例

```
#include <stdio.h>

int main(void)
{
    int a1 = 5;
    int a2 = 3;
    int a3 = 8;

    int b1 = 6;
    int b2 = 9;
    int b3 = 2;

    int c1, c2, c3;

    c1 = a1 + b1;
    c2 = a2 + b2;
    c3 = a3 + b3;

    printf("%d␣+␣%d␣=␣%d¥n", a1, b1, c1);
    printf("%d␣+␣%d␣=␣%d¥n", a2, b2, c2);
    printf("%d␣+␣%d␣=␣%d¥n", a3, b3, c3);

    return 0;
}
```

実行結果

```
5 + 6 = 11
3 + 9 = 12
8 + 2 = 10
```

例題 1.30 ベクトルの足し算 配列を使った例

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      int a[3] = {5, 3, 8};
6      int b[3] = {6, 9, 2};
7      int c[3];
8      int i;
9
10     for (i = 0; i < 3; i++) {
11         c[i] = a[i] + b[i];
12         printf("%d␣+␣%d␣=␣%d¥n", a[i], b[i], c[i]);
13     }
14
15     return 0;
16 }
```

実行結果

```
5 + 6 = 11
3 + 9 = 12
8 + 2 = 10
```

配列と繰返し文を使うと記述が楽になることがわかると思います．数値計算では数千〜数万個の変数を扱うことが珍しくありません．配列を使えば簡単に扱えますが，普通の変数では事実上不可能でしょう．

1.6.2 多次元配列

次のように宣言することで 2 次元の配列を作ることができます．

2 次元配列の宣言

型　配列名 [サイズ][サイズ];

int 型の 10×5 の配列を宣言する場合は次のようにします．

```
int array[10][5];
```

3 次元の配列なら

```
int array[10][5][17];
```

のように宣言します．このように各次元ごとに要素のサイズを追加すればいくらでも次元を増やすことができます．使い勝手やメモリーサイズの都合上から 4 次元配列以上はあまり使われないようです．

2 次元配列 array の要素番号 [2][3] に 100 を代入する場合は次のようにします．

```
array[2][3] = 100;
```

2 次元配列の宣言時に初期値を与える場合は次のようにします.

```
int array[3][2] = {{1, 2}, {3, 4}, {5, 6}};
```

2 次元配列は行列のイメージそのままに使用することができます. 次の例題では以下の計算を 2 次元配列を使って行います.

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{pmatrix} + \begin{pmatrix} 7 & 8 \\ 9 & 10 \\ 11 & 12 \end{pmatrix} = \begin{pmatrix} 8 & 10 \\ 12 & 14 \\ 16 & 18 \end{pmatrix}$$

例題 1.31　2 次元配列で 3 行 2 列の行列の和を計算

```
#include <stdio.h>

int main(void)
{
    int i, j;
    int matrixA[3][2] = {    /* 3行 2列の行列A を宣言 */
        {1, 2},
        {3, 4},
        {5, 6}
    };

    int matrixB[3][2] = {  /* 3行 2列の行列B を宣言 */
        {7, 8},
        {9, 10},
        {11, 12}
    };

    int matrixC[3][2];  /* 3行 2列の行列C を宣言 */

    for (i = 0; i < 3; i++) { /* 行列A + 行列B を 行列C に代入 */
        for (j = 0; j < 2; j++) {
            matrixC[i][j] = matrixA[i][j] + matrixB[i][j];
        }
    }

    for (i = 0; i < 3; i++) { /* 行列C の内容を表示 */
        for (j = 0; j < 2; j++) {
            printf("%d¥t", matrixC[i][j]);
        }
        printf("¥n");
    }

    return 0;
}
```

実行結果

```
8       10
12      14
16      18
```

1.6.3 課題

課題 12　3 次元ベクトルの内積を求めるプログラムを作ってみよう.

課題 13　3 行 3 列の行列 A と B の積 AB を求めるプログラムを作ってみよう.

ヒント: 行列 C を A と B の積とすると C の要素 C_{ij} は次式で与えられる.

$$C_{ij} = \sum_{n=1}^{k} A_{in}B_{nj} = A_{i1}B_{1j} + \cdots + A_{ik}B_{kj}$$

for ループを 3 重にネストして C_{ij} を求める.

1.7　自作関数

今回**自作関数**について学びます．今まで main() 関数，printf() 関数や scanf() 関数などの標準入出力関数，sin() 関数や sqrt() 関数などの数学関数など，C 言語処理系にもともと用意されている関数を利用してきました．関数というものはとても便利なもので，() 内に**引数**を与えて関数を呼び出すとブラックボックス的に処理が行われ，自動的に処理結果が**返り値**として戻ってきます．プログラムを組む際に関数内の処理や仕組みを気にする必要がありません．

また，これまでに数列の和を求めるプログラムや 2 次方程式の解の判別プログラムなど, 様々なプログラムを組んできました．これらは main() 関数内に直接処理内容を記述していました．しかし，例えば初期値の違う数列の和をいくつも求めたい場合など，プログラム内の違う場所で同じような処理を行いたい場合には main() 関数内に似たような記述を何度も書く必要がでてきます．また一連の処理が非常に長くなった場合などには main() 関数内の見通しが悪くなります．このような場合に一連の処理を関数化し，main() 関数の外に追い出すことができます．このように自分で作成する関数を**自作関数**と呼びます．自作関数は今まで利用してきた関数と同様に，引数を受けとり自動的に処理を行い**返り値**を戻します．

自作関数を活用すると重複した処理を再利用可能な形にするため (サブルーチン化，モジュール化)，効率がよく理解しやすいプログラムを作ることができます．main() 関数内には最低限の記述だけをすればよいのでプログラムが大きくなっても処理の流れを把握しやすくなります．また自作関数は引数と返り値だけをしっかり決めておけばいいので別のプログラムに流用することも容易です．

自作関数は一般的に次のように宣言します．

```
/* ヘッダファイルのインクルードなど */
#include <stdio.h>
...

/* 自作関数のプロトタイプ宣言 */
返り値の型 関数名1(仮引数の型 仮引数名1, 仮引数の型 仮引数名2, ... , 仮引数の型 仮引数名N);
...
返り値の型 関数名N(仮引数の型 仮引数名1, 仮引数の型 仮引数名2, ... , 仮引数の型 仮引数名N);

int main(void)
{
    ...
    return 0;
}

/* 自作関数 */
返り値の型 関数名1(仮引数の型 仮引数名1, 仮引数の型 仮引数名2, ... , 仮引数の型 仮引数名N)
{
    ...
    return 返り値;
}
```

```
...

返り値の型 関数名 N(仮引数の型 仮引数名 1, 仮引数の型 仮引数名 2, ... , 仮引数の型 仮引数名 N)
{
    ...
    return 返り値;
}
```

1.7.1 自作関数とプロトタイプ宣言

自作関数は main() 関数の外に記述します．自作関数は

1. 呼び出し側の引数の値を受け取って
2. 仮引数にコピーし
3. 処理を行い
4. 呼び出し側に返り値を返し

ます．自作関数も通常の関数と同様に，引数と返り値の型がわからなければ呼び出したり返り値を変数に格納したりすることができません．そのため自作関数が実際に呼び出される前に，あらかじめ処理の先頭で関数名と返り値の型，仮引数の型と名前を宣言しなくてはなりません．このことを関数の**プロトタイプ宣言**と呼びます．呼び出し側の引数の型と受け取る側の自作関数の仮引数の型は一致している必要があります．自作関数の実際の処理は main() 関数の後に記述します．引数を取らない場合や返り値を返さない自作関数の場合は void 型を用います．

まず引数も取らず返り値も返さない自作関数の例を見てみましょう．main() 関数内では printf() 関数を使っていないにも関わらず，画面には「こんにちは」と表示されています．

例題 1.32　自作関数で文字列を表示

```
1  #include <stdio.h>
2
3  /* プロトタイプ宣言 */
4
5  void hello(void);
6
7  int main(void)
8  {
9      hello(); /* 自作関数の呼び出し */
10
11     return 0;
12 }
13
14 /* 自作関数 */
15 void hello(void)
16 {
17     printf("こんにちは ¥n");
18 }
```

実行結果

```
こんにちは
```

次に引数を受け取り，返り値を返す自作関数の例を見てみましょう．関数 sum() は 1 から受け取った数までの和を計算し，計算結果を返します．

例題 1.33 1 から n までの和を求める

```
#include <stdio.h>

/* プロトタイプ宣言 */

int sum(int);

int main(void)
{
    int n, m;
    n = 100;
    m = sum(n); /* 自作関数の呼び出し */

    printf("%d¥n", m);

    return 0;
}

/* 1からn まで足す自作関数 */
int sum(int n)
{
    int i;
    int l = 0;

    for (i = 0; i <= n; i++) {
        l += i;
    }

    return l;
}
```

実行結果

```
5050
```

関数の型と引数の型が同じであるとは限りません．次の例は int 型の引数を関数に渡し，double 型の返り値を取る例です．整数型の引数を関数に渡して `for()` ループの回数を指定し，関数内で計算を行い実数を返します．

例題 1.34　関数の型と引数の型が異なる場合の例

```c
#include <stdio.h>

double func(int n)
{
    int i;
    double x = 1.0;

    for (i = 0; i <= n; i++) {
        x = x / 2.0;
    }

    return x;
}

int main(void)
{
    int n = 2;
    double x;

    x = func(n);
    printf("1␣/␣2^%d␣=␣%f¥n", n, x);

    return 0;
}
```

実行結果

```
1 / 2^2 = 0.125000
```

次は階乗を求めるプログラムです．自作関数を使用しない場合と使用する場合を比較してみましょう．まず自作関数を使用しないプログラムの例です．

例題 1.35 階乗を求める 自作関数を使わない

```c
#include <stdio.h>

int main(void)
{
    int x, y, i;

    x = 3;
    y = 1;
    for (i = 1; i < x; i++) {
        y = y + y * i;
    }
    printf("%d␣の階乗は␣%d␣です ¥n", x, y);

    x = 7;
    y = 1;
    for (i = 1; i < x; i++) {
        y = y + y * i;
    }
    printf("%d␣の階乗は␣%d␣です ¥n", x, y);

    return 0;
}
```

実行結果

```
3 の階乗は 6 です
7 の階乗は 5040 です
```

次に，階乗を求めている for ループの部分を自作関数化し，factorial() 関数として使用できるようにします．例えば y = factorial(4) のように呼び出すと，変数 y に 4! の計算結果 24 が代入されるようにします．これは factorial() 関数に引数として 4 を渡すと，返り値として 24 が戻されることを意味します．先の例は次のように書き換えられます．

例題 1.36　階乗を求める 自作関数を使う 1

```
1  #include <stdio.h>
2
3  /* プロトタイプ宣言 */
4  int factorial(int n);
5
6  int main(void)
7  {
8      int x, y;
9
10     x = 3;
11     y = factorial(x); /* 自作関数の呼び出し */
12     printf("%d␣の階乗は␣%d␣です ¥n", x, y);
13
14     x = 7;
15     y = factorial(x); /* 自作関数の呼び出し */
16     printf("%d␣の階乗は␣%d␣です ¥n", x, y);
17
18     return 0;
19 }
20
21 /* 階乗を求める自作関数 */
22 int factorial(int n)
23 {
24     int ans, i;
25     ans = 1;
26
27     for (i = 1; i < n; i++) {
28         ans = ans + ans * i;
29     }
30
31     return ans; /* ans を返り値として返す */
32 }
```

実行結果

```
3 の階乗は 6 です
7 の階乗は 5040 です
```

まず自作関数 `factorial()` のプロトタイプ宣言をしています．引数 n の階乗を求める関数なので返り値と仮引数も整数となります．そのためともに int 型で宣言します．[*9]`main()` 関数内では int 型変数 `x` を引数にとって自作関数 `factorial()` を呼び出し，返り値を int 型変数 `y` に格納しています．

自作関数 `factorial()` の中身を見てみましょう．まず int 型で宣言された仮引数 `n` に引数 `x` の値 3 がコピーされます．仮引数はその関数内で変数として使用できます．`n` の階乗を計算し，int 型変数 `ans` に計算結果を格納します．

[*9] 返り値の型を関数自体の型とみなして factorial() を int 型の関数と呼ぶこともあります．

return ans; とすることで変数 ans の値を返り値として呼び出し側に返します.

ここで main() 関数についても少し説明しましょう. いつも int main(void) と宣言していますが, これは OS(MS Windows や Linux など) から引数を受け取らず, プログラム自体の返り値を OS に int 型で返すということを意味しています. return 0; とすると OS に 0 が返されます. 今はなんのことか分からないと思いますが, C 言語の勉強を進めるとなんの役にたつのか分かってきます. ここでは main() 関数も自作関数も同じように返り値と仮引数があるということが分かれば十分です.

なお, 自作関数を main() 関数の前に書くとプロトタイプ宣言を省略することができます. しかし, 自作関数が長くなってくると main() 関数がプログラムの後ろに追いやられて見通しが悪くなるので, この記法はあまりお勧めしません.

例題 1.37 階乗を求める 自作関数を使う 2 プロトタイプ宣言を省略

```
1  #include <stdio.h>
2
3  /* 階乗を求める自作関数 */
4  int factorial(int n)
5  {
6      int ans, i;
7      ans = 1;
8
9      for (i = 1; i < n; i++) {
10         ans = ans + ans * i;
11     }
12
13     return ans; /* ans を返り値として返す */
14 }
15
16 int main(void)
17 {
18     int x, y;
19
20     x = 3;
21     y = factorial(x); /* 自作関数の呼び出し */
22     printf("%d␣の階乗は␣%d␣です ¥n", x, y);
23
24     x = 7;
25     y = factorial(x); /* 自作関数の呼び出し */
26     printf("%d␣の階乗は␣%d␣です ¥n", x, y);
27
28     return 0;
29 }
```

実行結果

```
3 の階乗は 6 です
7 の階乗は 5040 です
```

1.7.2 関数の独立性と変数の値渡し

自作関数は main() 関数や自作関数から呼び出されます．呼び出された際に引数を受け取り，処理を行い，**return 文で返り値を返します**．また，**関数内で宣言された変数は，関数ごとに独立しています**．[*10]main() 関数と自作関数でそれぞれ x という変数を宣言して内容を変更してもお互いに影響を受けません．**関数間での引数のやりとりでは変数そのものではなく，変数の中身の具体的な値がコピーされて渡されます**．これは C 言語において非常に重要な考え方で，このような変数の渡し方を**値渡し**(call by value) と呼びます．[*11]そのため関数を呼び出すときに関数内での変数名を気にする必要はありません．各関数で同じ名前の変数を宣言してもまったく別の変数として扱われます．

次の例では main() 関数と func 関数でそれぞれ同じ名前の変数 x，y を定義し，関数内で値を変更しています．実行結果を見ると変数の値の変更が関数の外に影響しないことがわかります．

[*10] 関数ごとに独立した変数を**ローカル変数**と呼びます．関数外で宣言した変数を**グローバル変数**と呼びプログラムの全体で使用することができますが，関数の独立性が弱くなるので多用するべきではありません．

[*11] 後で学ぶポインタというものを使うと関数間で変数自体を渡すことができます．このことを**参照渡し**と呼び変数の中身ではなく変数のアドレスをやりとりします．配列は実は参照渡しになります．

例題 1.38 変数の値渡し

```
#include <stdio.h>

double func(double x);

int main(void)
{
    double x, y;
    x = 1.0;
    y = 2.0;

    printf("in␣main¥tx␣=␣%f␣y␣=␣%f¥n", x, y);
    y = func(x);
    printf("in␣main¥tx␣=␣%f␣y␣=␣%f¥n", x, y);

    return 0;
}

double func(double x)
{
    double y;
    x = 3.0;
    y = 4.0;
    printf("in␣func¥tx␣=␣%f␣y␣=␣%f¥n", x, y);
    return x;
}
```

実行結果

```
in main x = 1.000000 y = 2.000000
in func x = 3.000000 y = 4.000000
in main x = 1.000000 y = 3.000000
```

変数 x は自作関数内で x = 3.0 とされていますが，main() 関数内の変数 x は 1.0 のままです．

1.7.3 課題

課題 14 2^n を求めるプログラムを作り，うまくいったら指数を求める部分を自作関数 pow2() にしてみよう．例えば pow2(8) とすると 256 が返されるようなプログラムにする．引数，返り値は整数とする．

課題 15 さらに a^n を求めるプログラムにしてみよう．例えば powan(3, 4) とすると 81 が返されるようなプログラムにする．

課題 16 ヘロンの公式を用いて三角形の面積を求める自作関数を作ってみよう．引数，返り値は実数とする．

ヘロンの公式は三角形の 3 辺の長さから面積を求める公式である．3 辺の長さがそれぞれ a, b, c の三角形の場合，$s = \dfrac{1}{2}(a+b+c)$ とすると面積 S は $S = \sqrt{s(s-a)(s-b)(s-c)}$ で求められる．

第 2 章

数値計算の基礎

数値計算とアルゴリズム

x を t の関数とする微分方程式があるとします．線形微分方程式などの性質の良い微分方程式であれば一般解が存在し，$x(t)$ の関数を求めることができます．このことを解析的に解くといいます．

しかし世の中の多くの微分方程式には一般解が存在しないため，その場合は t の関数を求めることができません．しかし，$t = 0$ のときの値 (初期条件) を決めると t が微小時間 Δt だけ進んだときの $x(t)$ の値そのものを求めることができます．これを漸化式を用いて繰り返すことにより，$t = a$ のときの $x(a)$ の値を近似的に求めることができます．

解析的に解けない数学の問題をコンピュータを使って近似的に解く手法のことを数値計算といいます．代数方程式でも 2 次方程式の解は一般的に知られていて解析的に解くことができますが，多くの高次方程式の解を求めるには数値計算が強力な手法となります．

主にコンピュータを使用してなんらかの問題を解決する一連の手順をアルゴリズムと呼びます．数値計算で使われる有名なアルゴリズムとして，非線形方程式を解くニュートン法，常微分方程式を解くルンゲ・クッタ法，連立方程式を解く LU 分解法などがあります．数値計算以外のアルゴリズムも無数にあります．例えばランダムに入力された英単語をアルファベット順に並べかえて表示するのにはソート (並び換え) と呼ばれるアルゴリズムを考えることになります．この講義ではこれらの数値計算用のアルゴリズムを使って数学や物理の問題を解く方法を学びます．

2.1 非線形方程式の解法

関数 $y = f(x)$ が x 軸と $x = \alpha$ で交わっているとします．このとき $f(\alpha) = 0$ となり，$x = \alpha$ は方程式 $f(x) = 0$ の解となります．この解 α を求めることが今回の目的となります．

この方程式が一次方程式 (線形方程式) であれば手計算で簡単に解くことができます．また二次の代数方程式である場合は非線形方程式となりますが，解の公式を用いて簡単に解くことができます．しかし一般的な非線形方程式の場合は数値計算を用いないと簡単には解けません．ここでは**二分法**(Bisection method) と**ニュートン法**(Newton's method) と呼ばれるアルゴリズムを使って近似的に解を求める手法を学びます．

2.1.1 二分法 (Bisection method)

二分法は解の存在する区間を調べて区間を $\frac{1}{2}$ ずつ狭めていくアルゴリズムです．区間が十分小くなり，$f(x)$ の絶対値が十分小さい値 ε(イプシロン) 以下となればその値を近似解とみなします．

$f(x)$ を連続関数とし，2 点 $a, b(a < b)$ 内で $f(a) < 0, f(b) > 0$ を満たしている場合は以下のように解を求めます．($f(a) > 0, f(b) < 0$ の場合は $f(x)$ を $-f(x)$ などとする．)

1. 解を含むおおまかな区間を設定し，下限 a と上限 b を定める．
2. a と b の中間点 c を求める．
3. $f(c) > 0$ ならば解は c, b 間に存在する．c を a に置き換える．
4. $f(c) < 0$ ならば解は a, c 間に存在する．c を b に置き換える．
5. $f(c) > \varepsilon$ なら 2 に戻る．$f(c) \leq \varepsilon$ ならば c が近似解．

二分法はニュートン法と比較して解の収束は遅いものの関数が連続で解がひとつだけであれば確実に収束するという特徴を持っています．

例として方程式 $x^2 - 2 = 0$ の近似解をひとつ求めてみます．

例題 2.1　二分法

```
#include <stdio.h>
#include <math.h>

int main(void)
{
    double a = 0.0, b = 10.0, c, fc, eps;
    int i = 0;
    eps = 0.0001; /* 微小な値 ε (イプシロン) */

    c = (a + b) / 2;
    fc = c * c - 2; /* 関数を定義 */

    printf("i¥ta¥t¥tc¥t¥tb¥t¥t¥tfc¥n");

    while (fabs(fc) > eps) {
        printf("%d¥t%f¥t%f¥t%f¥t%f¥n", i, a, c, b, fc);

        if (fc > 0) b = c;
        else a = c;

        c = (a + b) / 2;
        fc = c * c - 2;

        i++;
    }

    printf("%d¥t%f¥t%f¥t%f¥t%f¥n", i, a, c, b, fc);
    return 0;
}
```

実行結果

```
i   a         c         b         fc
0   0.000000    5.000000    10.000000   23.000000
1   0.000000    2.500000    5.000000    4.250000
2   0.000000    1.250000    2.500000    -0.437500
3   1.250000    1.875000    2.500000    1.515625
4   1.250000    1.562500    1.875000    0.441406
5   1.250000    1.406250    1.562500    -0.022461
6   1.406250    1.484375    1.562500    0.203369
7   1.406250    1.445312    1.484375    0.088928
8   1.406250    1.425781    1.445312    0.032852
9   1.406250    1.416016    1.425781    0.005100
10  1.406250    1.411133    1.416016    -0.008704
11  1.411133    1.413574    1.416016    -0.001808
12  1.413574    1.414795    1.416016    0.001645
13  1.413574    1.414185    1.414795    -0.000082
```

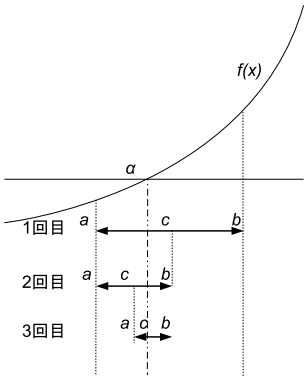

図 2.1 二分法

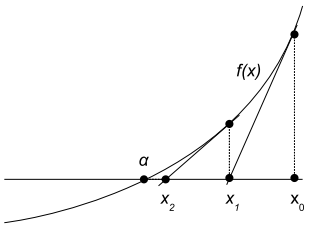

図 2.2 ニュートン法

2.1.2 **ニュートン法** (Newton's method)

ニュートン法は次の漸化式を計算することで方程式の近似解を得るアルゴリズムです.

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)} \tag{2.1}$$

$x = \lim_{k\to\infty} x_k = \alpha$ で解となりますが実際のプログラムでは $f(x_k) \leq \varepsilon$ となった時点で計算を打ち切り，そのときの x_k を近似解とします．差分法の例と同じく方程式 $x^2 - 2 = 0$ の近似解を求めてみます.

例題 2.2　ニュートン法

```
#include <stdio.h>
#include <math.h>

int main(void)
{
    double x, eps;
    int i = 0;
    x = 10.0; /* 初期値 x0 を与える */
    eps = 0.0001; /* 微小な値 ε (イプシロン) */


    printf("i¥tx¥t¥tfx¥n");

    /* 関数f(x)が 0に近づくまでニュートン法を繰り返す */
    while (fabs(x * x - 2) > eps) {
        printf("%d¥t%f¥t%f¥n", i, x, x * x - 2);
        x = x - (x * x - 2) / (2 * x); /* 漸化式 */
        i++;

    }

    printf("%d¥t%f¥t%f¥n", i, x, x * x - 2);
    return 0;
}
```

実行結果

```
i   x       fx
0   10.000000   98.000000
1   5.100000    24.010000
2   2.746078    5.540947
3   1.737195    1.017846
4   1.444238    0.085824
5   1.414526    0.000883
6   1.414214    0.000000
```

ニュートン法は二分法より収束が速く実用的なので非常によく使われています．しかし，ニュートン法では初期値 x_0 の選び方によっては求めようとする解とは別の解に収束したり，まったく収束しないこともあります．

ニュートン法の原理と導出

ニュートン法の原理を図で考えます．図 (2.2) のように，$y = f(x)$ が x 軸と $x = \alpha$ で交わっている場合，α が求める解となります．まず $x = x_0$ の点に垂線を立て，$f(x)$ との交点で $f(x)$ に接線を引きます．接線と x 軸の交点を x_1 とし，また垂線を立てます．この操作を繰り返すと $x = \lim_{k\to\infty} x_k = \alpha$ となります．式 (2.1) の漸化式ではこの操作を意

味していると解釈できます.

また，式 (2.1) の漸化式は関数 $f(x)$ をテイラー展開 (Taylor expansion) したものの一次近似より得られます．関数 $f(x)$ を $x = \alpha$ の周りでテイラー展開をすると一般的に次のようなべき級数で表されます.

$$f(x) = \sum_{n=0}^{\infty} \frac{f^{(n)}(\alpha)}{n!}(x-\alpha)^n = f(\alpha) + f'(\alpha)(x-\alpha) + \frac{f''(\alpha)}{2!}(x-\alpha)^2 + \cdots + \frac{f^{(n)}(\alpha)}{n!}(x-\alpha)^n$$

関数 $f(x) = 0$ となる x を求めたいとき，$x = x_0$ を適当に決め，$x = x_0$ の周りのテイラー展開の一次近似より

$$f(x) = f(x_0) + f'(x_0)(x - x_0) \tag{2.2}$$

$f(x_1) = 0$ となる x_1 を式 (2.2) に代入すると

$$f(x_1) = f(x_0) + f'(x_0)(x_1 - x_0) = 0 \tag{2.3}$$

整理して

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)} \tag{2.4}$$

これにより x_1 が求まります．次に x_2 を求めます．式 (2.4) において，$x_0 \to x_1$，$x_1 \to x_2$ と 置き換えると

$$f(x_2) = f(x_1) + f'(x_1)(x_2 - x_1) = 0 \tag{2.5}$$

$$x_2 = x_1 - \frac{f(x_1)}{f'(x_1)} \tag{2.6}$$

これを繰り返すと式 (2.1) の漸化式を得ます．式 (2.4) の右辺を値を変えると $f(x) = 0$ となる解に限らず，さまざまな $f(x)$ の値を解とする漸化式を導出することもできます.

2.1.3 課題

課題 17　$x + \log x = 0$ の解を二分法，ニュートン法それぞれ用いて求めてみよう．区間や初期値をよく考える.

課題 18　$x^3 - 2x + 2 = 0$ は初期値 $x_0 = 1.0$ とするとニュートン法では解に収束しない．その理由を考えてみよう.

2.2 常微分方程式の解法

2.2.1 常微分方程式とは (解析解)

物理の基本的な法則は微分方程式 (導関数を含む関係式) で表現されることが多く，微分方程式を解くことにより様々な情報を得ることができます．例えば運動方程式は 2 階線形微分方程式と言われるものです．運動方程式を解くと関数 $x(t)$ が得られ，時間 t における物体の位置 x を求めることができます．力学では時間 t における物体の位置 x を求めることが大きなテーマとなります．F を力,m を質量とすると運動方程式は

$$m\frac{d^2x(t)}{dt^2} = F \tag{2.7}$$

と表現されます．加速度を a とすると $F = ma$ と置け，等加速度運動の場合式 (2.7) は

$$\frac{d^2x(t)}{dt^2} = a \tag{2.8}$$

となります．この 2 階線形微分方程式の両辺を積分すると

$$\frac{dx(t)}{dt} = at + C_1 \tag{2.9}$$

もう一度両辺を積分すると

$$x(t) = \frac{1}{2}at^2 + C_1t + C_2 \tag{2.10}$$

となり関数 (一般解) を得ます．C_1, C_2 は積分定数です．

このように微分方程式から微分を含まない元の関数を求めることを**解析的に微分方程式を解く**と言います．導関数の中の変数が 1 つの場合を常微分方程式，複数の場合を偏微分方程式と呼びます．

式 (2.10) の積分定数 C_1 は初速度 v_0，C_2 は初期位置 x_0 を意味し，これらを初期条件と呼びます．初期条件を与えると $x(t)$ を完全に決定することができ，初期条件が与えられた場合の微分方程式を解く問題のことを**初期値問題**と言います．この例では初期条件と時間 t を与えればその時点での物体の位置を完全に求めることができます．

また，式 (2.7) は 2 階線形微分方程式でしたが

$$\frac{dx(t)}{dt} = v(t) \tag{2.11}$$

とすると式 (2.7) は

$$\frac{dv(t)}{dt} = a \tag{2.12}$$

と置け，1 階の連立微分方程式に分解することもできます．

2.2.2 数値解法

解析的に解く方法とは別の方法でも時刻 t おける位置 x を知ることができます．

$t_0 \to t$ を n 分割し，Δt ごとの位置の変化分 Δx を求めます．その変化分 Δx を n 個すべて足し合わせると時刻 t における位置 x がわかります．

$\dfrac{\Delta x}{\Delta t} \to \dfrac{dx}{dt}$ の極限では近似ではない正確な値がわかりますが，計算機では無限小を扱うことができないため有限な数字 ($\Delta t = 0.01$) などで近似します．

このように近似的に解を得る方法を数値解法と呼びます．Δx を求める方法は後ほど説明しますが非常に簡単です．

さて微分方程式が与えられたときに，式 (2.10) のように関数の形までもっていくことができれば一番よいのですが，物理の多くの問題では解析的に解くことが困難であったり不可能であったりします．例えば単振り子の運動方程式は

$$\frac{d^2\theta(t)}{dt^2} = -\frac{g}{l}\sin\theta(t) \tag{2.13}$$

と書けますが，この方程式は $\sin\theta$ を含み非線形なので解析解を求めることは困難です．[*1]一般的な力学の教科書では振り子の振れ幅が小さいときに $\sin\theta \simeq \theta$ と近似して

$$\theta(t) = \theta_0 \sin\left(\sqrt{\frac{g}{l}}t + \delta\right) \tag{2.14}$$

として解析解を出し，時刻 t における角度を求める関数 $\theta(t)$ を得ています．[*2]この方法だと振り子の振れ幅が大きい場合には正しい値を得ることができません．

しかし数値計算を使うと運動方程式から簡単に時刻 t におけるほぼ正確な角度を求めることができます．

このように解析的に解けない微分方程式でも数値計算により簡単に近似的な解を得ることができます．今回は常微分方程式の数値計算法の中でもアルゴリズムを理解しやすい**オイラー法**(Euler method) について学びます．

2.2.3 オイラー法 (Euler method)

オイラー法は次の漸化式を計算することで微分方程式の近似解を得るアルゴリズムです．

$$x_{k+1} = x_k + hf(x_k, t_k) \tag{2.15}$$

1 階常微分方程式

$$\frac{dx}{dt} = f(x(t), t) \tag{2.16}$$

を数値的に解くことを考えます．

$t-x$ グラフを考え，区間を $t = t_0, t_1, t_2, \cdots, t_k, \cdots, t_n$ のように区切り，対応する x の数値 $x = x_0, x_1, x_2, \cdots, x_k, \cdots, x_n$ を求めてみましょう．

まず微分を差分に置き換えます．t のきざみ幅 Δt を h とすると ($\Delta t = h = t_{k+1} - t_k$)，式 (2.16) の左辺は次のように近似できます．

$$\left.\frac{dx}{dt}\right|_{t=t_k} \simeq \left.\frac{\Delta x}{\Delta t}\right|_{t=t_k} = \frac{x_{k+1} - x_k}{h} \tag{2.17}$$

この式を前進差分と呼びます．

式 (2.16)，(2.17) から

$$x_{k+1} = x_k + hf(x_k, t_k) \tag{2.18}$$

の漸化式を得ます．

この式の物理的な意味を座標 x と時刻 t で考えてみましょう．(右辺) 速度 $f(x_k, t_k)$ と微小時間 h の積は微小時間での位置の変化分となります．変化分を時刻 t_k 時の質点の座標 x_k に足すと，(左辺) 微小時間 h 後の座標 $x(k+1)$ が得られます．

この操作を繰り返し，微小時間ごとの位置の増分を足し合わせることで任意の時刻の質点の位置を知ることができます．

[*1] 楕円関数というものを使うと解析解を得ることができますが，高度な数学的知識が必要になります．

[*2] θ_0 は振動の最大角，δ は初期位相

図 (2.3) のように，オイラー法をグラフを使って考えることもできます．x_k, t_k における傾きにきざみ幅 h を掛けると Δx_{k+1} を得ます．それまで得ている $x_k (k = 0$ の場合は初期値 $x_0)$ に Δx_{k+1} を足すことで x_{k+1} を得ます．この操作を繰り返すことで $x(t)$ を得ることができます．

また，図から明らかなように，k を増やすたびに真の値との誤差が蓄積されていきます．一般的には $\lim_{h \to 0} h$ とすることで真の値となりますが，コンピュータでは有限の値しか扱えないことと，オイラー法では変化分が 1 次近似のため精度が悪いのでどうしても誤差が大きくなります．

このように誤差の大きいオイラー法はあまり実用的ではありませんが，アルゴルズムを理解しやすいため常微分方程式の数値計算の概念を学ぶには最適な手法です．実用的には 4 次近似で非常に精度の高いルンゲ・クッタ法がよく使われます．オイラー法をよく理解したらルンゲ・クッタ法を学んでみましょう．

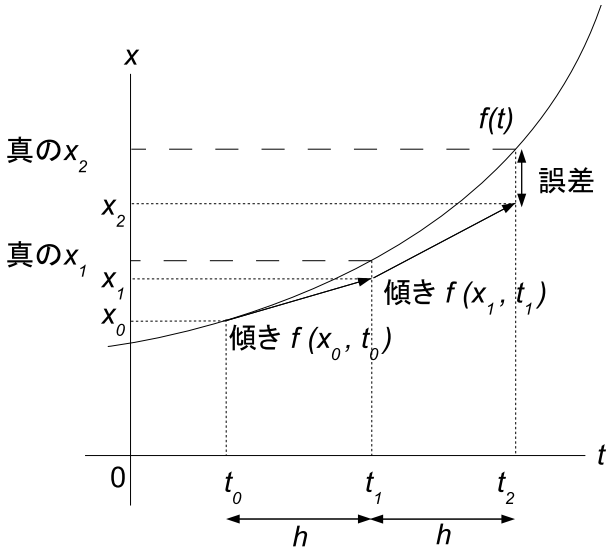

図 2.3　オイラー法

2.2.4　1 階のオイラー法

では，微分方程式 $\dfrac{dx}{dt} = t$ をオイラー法を使って解いてみましょう．初期条件は $x(0) = 0$ で，$t = 5$ の時点での解を求めます．

例題 2.3 オイラー法 1

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      double x, t, h, t_last;
6      x = 0;
7      t = 0;
8      h = 0.01;
9      t_last = 5.0;
10
11     printf("#t¥tx¥n");
12     while (t <= t_last) {
13         x = x + h * (t); /* 漸化式 */
14         printf("%f¥t%f¥n", t, x);
15         t = t + h;
16     }
17         printf("%f¥t%f¥n", t, x);
18     return 0;
19 }
```

実行結果

```
#t  x
0.000000    0.000000
0.010000    0.000100
...
5.000000    12.525000
5.010000    12.525000
```

次に微分方程式 $\dfrac{dx}{dt} = x(t)$ をオイラー法を使って解いてみましょう．初期条件は $x(0) = 1$ で，$t = 5$ の時点での解を求めます．

例題 2.4　オイラー法 2

```c
#include <stdio.h>

int main(void)
{
    double x, t, h, t_last;
    x = 1;
    t = 0;
    h = 0.01;
    t_last = 5.0;

    printf("#t¥tx¥n");
    while (t <= t_last) {
        x = x + h * (x); /* 漸化式 */
        printf("%f¥t%f¥n", t, x);
        t = t + h;
    }
        printf("%f¥t%f¥n", t, x);
    return 0;
}
```

実行結果

```
#t  x
0.000000    1.010000
0.010000    1.020100
...
5.000000    146.220500
5.010000    146.220500
```

2.2.5 課題

課題 19　微分方程式 $\dfrac{dx}{dt} = \cos(t)$ をオイラー法で解いてみよう．初期条件は $x(0) = 1$ で，$t = 7$ とする．

課題 20　微分方程式 $\dfrac{dx}{dt} = x\cos(t)$ をオイラー法で解いてみよう．初期条件は $x(0) = 1$ で，$t = 7$ とする．

課題 21　各例題課題の解析解とオイラー法で求めた数値解との誤差を比較してみよう．

2.2.6 2 階のオイラー法

運動方程式は 2 階の微分方程式ですが，オイラー法では 1 階の微分方程式しか解けません．そこで 1 階の連立微分方程式に帰着させて解くことにします．

$$\frac{d^2x(t)}{dt^2} = f(x(t), t) \tag{2.19}$$

上の 2 階微分方程式は次のように一階連立微分方程式と同等と考えることができます．

$$\frac{dx(t)}{dt} = g(x(t), t) \tag{2.20}$$

$$\frac{dg(x(t), t)}{dt} = f(x(t), t) \tag{2.21}$$

ここでは自由落下，斜方投射，単振動，単振り子といった基本的な運動方程式を例にとって解説して行きます．

自由落下

重力加速度を g とすると，$x-y$ 座標系での自由落下の運動方程式は次の式のようになります．

$$\frac{d^2y(t)}{dt^2} = -g \tag{2.22}$$

これを

$$\frac{dy(t)}{dt} = v_y(t) \tag{2.23}$$

$$\frac{dv_y(t)}{dt} = -g \tag{2.24}$$

とすると 1 階の連立微分方程式となります．

例題 2.5　2 階のオイラー法 自由落下

```
#include <stdio.h>

int main(void)
{
    double t, y, vy;
    double h, t_last;

    h = 0.1;
    t_last = 10.0;

    /* 初期値 */
    t = 0.0;
    y = 0.0;
    vy = 0.0;

    while (t < t_last) {
        printf("%f¥t%f¥t%f¥n", t, y, vy);
        y = y + h * (vy); /* 位置 ← ∫ 速度 */
        vy = vy + h * (-9.8); /* 速度 ← ∫ 加速度 */
        t = t + h;
    }
    return 0;
}
```

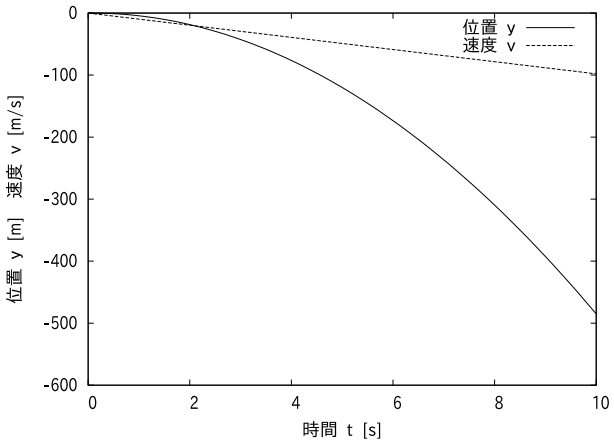

図 2.4　自由落下の $t-y$, $t-v$ グラフ

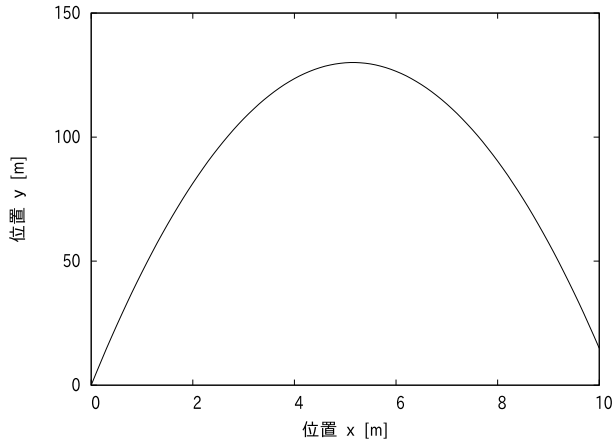

図 2.5　斜方投射の $x-y$ グラフ

#define **プリプロセッサディレクティブ**

オイラー法と直接は関係ありませんが，プログラムの先頭付近に

#define マクロ名 文字列

と記述するとコンパイル時にプログラム中に出てくる全てのマクロ名が文字列に置換されます．例えば#define g 9.8 のようにすると，プログラム中で使用されている g は 9.8 と置換されます．#define はセミコロンで終了しないことに注意してください．物理定数のような不変な値を#define で置換させることにより，自作関数を含めたプログラム全体で定数を共有することができます．なお，乱用すると関数の独立性を弱めてしまうので注意しましょう．また，変数と区別するためマクロ名は大文字で宣言すると良いでしょう．

次の例は自由落下のプログラムの重力加速度を#define g 9.8 とし，さらにオイラー法の関数部分を自作関数に追い出し少しだけ汎用的にしたものです．

例題 2.6　2 階のオイラー法 自由落下その 2

```
#include <stdio.h>
#define G 9.8

double f(double a);
double g(double v);

int main(void)
{
    double t, y, vy;
    double h, t_last;

    h = 0.1;
    t_last = 10.0;

    /* 初期値 */
    t = 0.0;
    y = 0.0;
    vy = 0.0;

    while (t < t_last) {
        printf("%f¥t%f¥t%f¥n", t, y, vy);
        y = y + h * g(vy); /* 位置 ← ∫速度 */
        vy = vy + h * f(-G); /* 速度 ← ∫加速度 */
        t = t + h;
    }
    return 0;
}

double f(double a)
{
    return a;
}

double g(double v)
{
    return v;
}
```

この例では関数部分を自作関数化するメリットはあまりありませんが，こういう表現に慣れておくと今後もっと複雑なアルゴリズムを使用する場合に応用が効きます．

斜方投射

自由落下では 1 次元の運動でしたが，斜方投射では 2 次元の運動になります．x 成分の運動方程式に加え x 成分の運動方程式を考えます．x 方向には加速度が働かないので

$$\frac{d^2x(t)}{dt^2} = 0 \tag{2.25}$$

となります．これを 1 階連立微分方程式にし，成分ごとにそれぞれ解きます．

$$\frac{dx(t)}{dt} = v_x(t) \tag{2.26}$$

$$\frac{dv_x}{dt} = 0 \tag{2.27}$$

例題 2.7　2 階のオイラー法 斜方投射

```
#include <stdio.h>
#define G 9.8

double f(double a);
double g(double v);

int main(void)
{
    double t, x, y, vx, vy;
    double h, t_last;

    h = 0.1;
    t_last = 10.0;

    /* 初期値 */
    t = 0.0;
    x = 0.0;
    y = 0.0;
    vx = 1.0;
    vy = 50.0;

    while (t < t_last) {
        printf("%f¥t%f¥t%f¥n", t, x, y);

        /* x 成分についてオイラー法 */
        x = x + h * g(vx);
        vx = vx + h * f(0);

        /* y 成分についてオイラー法 */
        y = y + h * g(vy);
        vy = vy + h * f(-G);

        t = t + h;
```

```
34      }
35      return 0;
36  }
37
38  double f(double a)
39  {
40      return a;
41  }
42
43  double g(double v)
44  {
45      return v;
46  }
```

単振動

k をばね定数とおくと単振動の運動方程式は

$$\frac{d^2x(t)}{dt^2} = -kx(t) \tag{2.28}$$

となります．今までの例と同様に簡単に 1 階の連立微分方程式に置き換えて解くことができます．試しに各自で解いてみてください．

単振り子

一般化座標を θ にとった単振り子の運動方程式は，糸の長さを l，重力加速度を g とおくと

$$\frac{d^2\theta(t)}{dt^2} = -\frac{g}{l}\sin\theta(t) \tag{2.29}$$

となります．この章の冒頭でも述べましたが，単振り子の運動方程式は 2 階非線形微分方程式であるため解析解は容易に求めることができませんが，オイラー法などの手法を使うと簡単に数値解を求めることができます．

一階の連立微分方程式にすると次のようになります．

$$\frac{d\theta(t)}{dt} = v_\theta(t) \tag{2.30}$$

$$\frac{dv_\theta(t)}{dt} = -\frac{g}{l}\sin\theta(t) \tag{2.31}$$

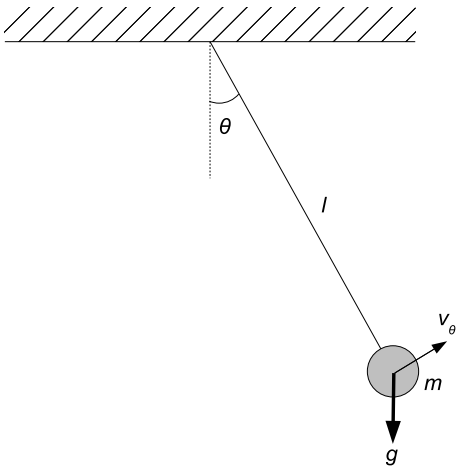

図 2.6　単振り子

例題 2.8　2 階のオイラー法 単振り子

```
1  #include <stdio.h>
2  #include <math.h>
3  #define G 9.8
4  #define L 1.0
5
```

```
6   /* θを求める運動方程式内の関数 */
7   double f(double theta);
8   double g(double vtheta);
9
10  /* θからx y 座標を求める*/
11  double x(double theta);
12  double y(double vtheta);
13
14  int main(void)
15  {
16      double t, theta, vtheta;
17      double h, t_last;
18
19      h = 0.1;
20      t_last = 10.0;
21
22      /* 初期値 */
23      t = 0.0;
24      theta = 1.0;
25      vtheta = 0.0;
26
27      while (t < t_last) {
28          printf("%f¥t%f¥t%f¥t%f¥t%f¥n", t, theta, vtheta, x(theta), y(theta));
29
30          theta = theta + h * g(vtheta); /* θ←∫ g(v θ )dt */
31          vtheta = vtheta + h * f(theta); /* v θ←∫ f(θ )dt */
32
33          t = t + h;
34      }
35      return 0;
36  }
37
38  /* θを求める運動方程式内の関数 */
39  double f(double theta)
40  {
41      return -G / L * sin(theta);
42  }
43
44  double g(double vtheta)
45  {
46      return vtheta;
47  }
48
49  /* θからx y 座標を求める*/
50  double x(double theta)
51  {
52      return L * sin(theta);
53  }
```

```
54
55  double y(double theta)
56  {
57      return L - L * cos(theta);
58  }
```

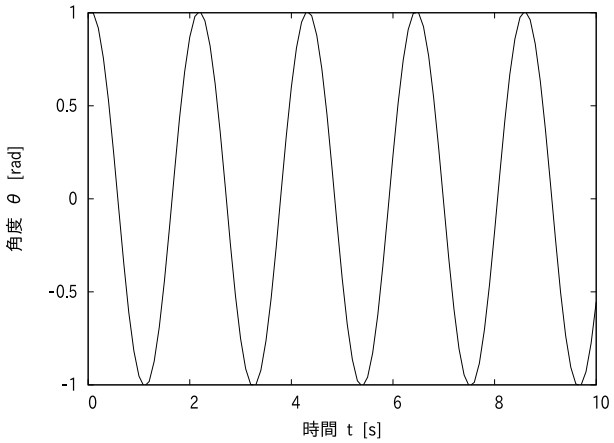

図 2.7　単振り子の $t-\theta$ グラフ

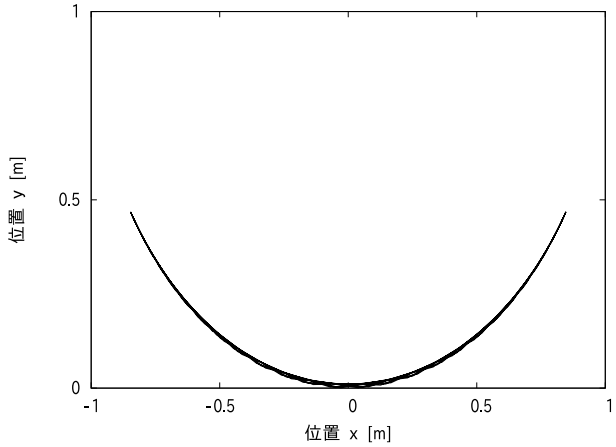

図 2.8　単振り子の $x-y$ グラフ

2.2.7　課題

課題 22　単振動の運動方程式をオイラー法で解いてみよう．

課題 23　空気抵抗がある場合の自由落下の運動方程式を立て，オイラー法で解いてみよう．十分時間が経つと速度が一定になることを確かめる．

付録 A

ソースコードの書き方

A.1 読みやすいソースコードを書くために

以下のようなことに気をつけてソースコードを記述することによりプログラムが格段に読みやすくなります.

A.1.1 名前のつけ方

- 変数名
  - 小文字ではじめる
  - 用途が連想できるようわかりやすい名前をつける
  - 長くなる場合はアンダーバー_で区切る
  - for などのループカウンタには `i j k` を使用する
  - `i j k l m n` は整数型以外に用いない
- 定数名
  - 変数名と明確に区別するため大文字で始める
- 関数名
  - 変数名のつけ方に準ずる
- 構造体
  - 大文字で始める

A.1.2 コーディングスタイル

- インデント
  - ブロック (論理的な意味のかたまり) 単位でインデントする
  - インデントは半角スペース 4 文字または 8 文字分のタブとする
- 中カッコ
  - ブロック開始行の行末に`{`を置く (改行しない！)
  - ブロック最終行の行頭に`}`を置く
  - 例外として関数定義の`{`は改行してから置く
- スペース
  - キーワードと小カッコの間にはスペースを置く

    `for (...), while (...), if (...), switch (...), case (...)`

- 関数と小カッコの間にはスペースを置かない
  `printf(...), scanf(...), sin(...), etc...`
- 二項演算子や三項演算子の前後にはスペースを置く
  `x + y, x * y, x == y, etc...`
- 単行演算子の後にはスペースを置かない
  `&x, *x, ++i, i++, etc...`
- 引数の区切りのカンマ, の後にはスペースを置く

A.1.3 その他

- goto 文は使わない
- グローバル変数は使用しない
- 一行に複数の処理を書かない
- 同じような処理を複数回書かない
  繰り返し処理や関数にする
- ネストが 3 重以上になったら処理を見直すか関数にする
- コメントがなくても理解しやすいコードを書く

以上は Linux カーネル コーディング規約[*1] に大きく準じています．各項目の理由や使用例が気になる人は規約を読んでみてください．非常に参考になります．

A.2 デバッグ

文法のミスはコンパイラが検出してくれるので簡単に取り除くことができます．しかしアルゴリズムのミス (バグ) はコンパイラには検出できず，プログラムは期待通りに動作しません．その場合ソースコードの問題点を的確に見つけ出すことが重要になります．

まずは以下のような点に注意しましょう．

- 変数は初期化されているか？
  変数を宣言した時点ではその値は不定です．必ず初期化してから使いましょう．
- 配列の範囲外にアクセスしていないか？
- 型は正しいか
  `3. / 2.` は `1.5` ですが `3 / 2` は `0` です．

A.2.1 printf() デバッグ

ミスの多くは変数の値をチェックすることによって見つけることができます．ソースコードの気になる部分に `printf()` を挿入し変数の値が期待通りになっているか確認します．

[*1] 日本語訳 `http://archive.linux.or.jp/JF/JFdocs/kernel-docs-2.6/CodingStyle.html`

A.2.2 条件付きコンパイル

デバッグ用の printf() を有効/無効にするためにいちいちコメント行にしたり外したりするのは面倒です．プリプロセッサの条件付きコンパイル機能を使いましょう．簡単な使用方法を説明します．詳細は参考書を参考にしてください．

```
#define DEBUG

    中略

#ifdef DEBUG
    printf(...);
#endif
```

この例では DEBUG というマクロ名が定義されていた場合#ifdef DEBUG と endif に挟まれた部分がコンパイル時に有効になります．DEBUG が定義されていない場合はコンパイル時に無効になります．#ifdef #endif の対はソースコード中に何個でも埋め込むことができます．つまり#define DEBUG 行をコメントアウトするかどうかで複数の処理をまとめて有効/無効にできます．

A.2.3 デバッガ

デバッガを使うとプログラムを動作させながら任意の変数の値をチェックすることができます．また，任意のところでプログラムを一時停止させたり (ブレークポイント)，1 ステップごとにプログラムを実行する (ステップ実行) などの機能を持ちます．

Borland C++ に対応した Turbo Debuger，GCC に対応した GDB などが有名です．また Visual Studio をはじめとする多くの統合環境は内部にデバッガを持っています．

A.2.4 その他

最も重要なことはデバッグしやすいソースコードを書くことです．付録 A.1 のようにソースコードを記述することでバグを発見しやすくなります．特に適切にインデントされていないソースコードは論理構造がつかみにくくデバッグは非常に困難になります．

また複雑な処理は可読性が悪くなるため，簡単な処理に分解してからそれらを組み合わせて書くようにしましょう．[*2]可読性が上がるだけでなく記述もしやすくなり，処理の単位が小さいのでバグを見つけやすくなります．気になる部分だけをコピー&ペーストで抜き出して別にコンパイルし検証したり，逆に簡単なプログラムを作成して検証したあとメインのプログラムに統合することも有効です．

[*2] 処理が複雑になりこんがらがったソースコードを**スパゲティコード**と呼びます

付録 B

その他の文法

本文中に記述できなかった重要な文法を簡単に紹介します.

B.1 型キャスト

B.2 条件文

B.2.1 条件演算子

B.2.2 break continue

B.2.3 switch case

B.3 ポインタ

B.3.1 ポインタと配列

B.3.2 ポインタと文字列

B.3.3 関数ポインタ

B.4 malloc - 動的メモリ確保と配列のサイズ

B.5 argv argc - main() 関数の引数

B.6 構造体と共用体

付録 C

その他の数値計算

本文中に記述できなかった重要な数値計算アルゴリズムを簡単に紹介します.

C.1 常微分方程式

C.1.1 ルンゲ・クッタ法

C.2 連立方程式

C.2.1 直接解法

ガウスの消去法

C.2.2 反復解法

ヤコビ法

C.3 固有値問題

C.3.1 べき乗法

C.4 簡単な偏微分方程式

C.4.1 差分法

C.5 モンテカルロ法

C.6 数値積分

C.6.1 台形則

C.7 関数の近似と補間法

C.7.1 最小 2 乗法

C.7.2 ラグランジュ補完

付録 D

アルゴリズムとデータ構造

代表的なアルゴリズムを簡単に紹介します.

参考文献

[1] 亀山峻吾. 月曜講義テキスト 2008, 日本大学理工学部計算物理研究会
[2] B.W. カーニハン D.M. リッチー. プログラミング言語 C 第 2 版. 2004, 共立出版.
[3] Herbert Schildt. 独習 C 第 4 版. 2007, 翔泳社.
[4] 椋田實. ANSI C 対応 はじめての C 改訂第 3 版. 1994, 技術評論社.
[5] 小澤一文. C で学ぶ数値計算アルゴリズム. 2008, 共立出版.
[6] 川上一郎. 理工系の数学入門コース 8 数値計算. 1989, 岩波書店
[7] 杉江日出澄 岡崎明彦 岩堀祐之 小栗宏次.　FORTRAN77 と数値計算法. 1997, 培風館
[8] 藪下信. 現代の数理科学シリーズ 1 計算物理 (I). 1982, 地人書館.
[9] 山本由紀. C 言語による物理基礎演習. 1991, パワー社
[10] C 言語で数値計算プログラミング
http://www.math.meiji.ac.jp/ mk/labo/studying-C/Programing-in-C/Programing-in-C.html
[11] Wikipedia http://ja.wikipedia.org