情報システム第 1

鈴木 宏正

東京大学工学部精密機械工学科
1998 年 10 月

目次

1 はじめに

1. 担当: 鈴木宏正
   〒 113 東京都文京区本郷 7–3–1
   東京大学大学院工学系研究科精密機械工学専攻
   電話: 03-5689-7319　　FAX: 03-5689-7243
   電子メール: suzuki@cim.pe.u-tokyo.ac.jp
   WWW: http://www.cim.pe.u-tokyo.ac.jp/~suzuki
   研究室: 工学部 14 号館 925 号室
2. C 言語プログラミングの基礎
   ANSI C (gcc コマンド) (cc コマンドは， K & R)
3. 教科書・参考書
   - 基礎 C 言語，土居，岩波，情報処理入門シリーズ 6
     このテキストの作成でもっとも参考にした本.
   - プログラミング言語 C， B.W. カーニハン /D.M. リッチー著，共立出版
     C の言語仕様について詳しい．専門家向き.
   - C 言語によるプログラミング基礎編 / 応用編，内田著，オーム社.
     大変丁寧な説明で分かりやすい.
   - ON TO C, P.H. Winston, Addison Wesley
     著者は人工知能で有名．最近出版されたものでは秀逸．日本語訳 (ウィンストンの C，アジソンウェスレー) も出ており，教材にしようかと迷ったがプログラムリストに誤植があり，断念した.
4. このテキストは LaTeX で書かれている．その dvi ファイル (cmain.dvi) と掲載されている C プログラムのファイルを，~hsuzuki/InfoSys/Text に置いておくので，参考にされたい.
   - テキストを見るには，% xdvi ~hsuzuki/InfoSys/Text/cmain とする.
   - プログラムを見るには，例えば図 1: hello.c とある場合には，
     ~hsuzuki/InfoSys/Text/hello.c が，そのプログラムファイルである．mule などで読み込んで見ることができる.

   なお，テキストに誤りなどを見つけた時には，電子メール等で連絡してください.

2 簡単なプログラム

2.1 こんにちは

最初のプログラムは，画面に Hello. と表示するプログラム.

```
1  /* hello.c */
2  #include <stdio.h>

3  main()
4  {
5    printf("Hello.\n");
6  }
```

図 1: hello.c

1. /* コメント */
   最初の行はコメント．最初の行に限らず，プログラムの中で/* */ で囲ってコメントを書く.

   漢字をつかってもよい. 逆に注釈と文字列 (後述) 以外では漢字は使えない.

2. #include<stdio.h>
   このプログラムでは画面に文字を表示するためにprintf という関数 (後述) を使っている．画面やキーボードを 標準入出力 (standard I/O) と言うが，これらの関数を使って標準入出力に表示や入力をする時には，このお呪いが必要であり，一般にプログラムの始めの方に書く． #include<stdio.h> のstdio は standard I/O の略.

   この #include<stdio.h> のように # で始まる行は マクロ命令 と呼ばれる (後述). #include<stdio.h> は，printf のような標準入出力を操作する関数に関する情報が入ったシステムのヘッダーファイル stdio.h[1]を読み込む命令.

   マクロ命令は，必ず一番左端 (1 カラム目) から書く.

3. 関数 main() 本体
   Pascal 言語では，プログラムは手続きと関数で構成されたが，C 言語では，関数 (function) だけで定義される．このプログラムは，もっとも簡単なもので関数main だけで構成されているが，一般には数多くの関数で構成される.

   逆に，すべてのプログラムには，このmain 関数が必ず一つだけなければならず，プログラムを実行した場合，main から計算が開始される.

4. 文とブロック
   このプログラムでは 文 は， printf("Hello.\n"); の一つだけだが，一般に関数の本体は，{文 文 …文} のように，main() の後に続く{ } の中に，計算などを行う文を並べる．文は;(セミコロン) で区切られる．このように複数の文を{ } で囲ったものを ブロック (複合文) という.

[1]試しに /usr/include/stdio.h を見てみよ.

5. 逐次実行
   文は，並びの順序に実行される．これを逐次実行という．

6. `printf("Hello.\n");`
   この文は，関数`printf` を使って，文字列`"Hello."`を画面 (標準出力) に表示する．`\n` は復帰改行 (画面上の次の行の頭に行くこと) を指定する制御文字.

7. フリーフォーマット
   C 言語は， Pascal と同様にフリーフォーマットなので，図 2のように詰め込んで書くこともできる．

```
1  /* hello2.c */
2  #include <stdio.h>

3  main(){ printf("Hello.\n"); }
```

図 2: hello2.c

作成・実行の仕方

1. mule ウィンドウの File メニューで Open File (C-x C-f) を選択し， hello.c と入力する．
2. 図 1のプログラムを入力し， File メニューの Save Buffer (C-x C-s) でセーブする．
3. shell ウィンドウで図 3を行なう．

実行例 1

```
> gcc hello.c
     (文法間違いがあると，ここでエラーメッセージがでる.
      その時は，mule ウィンドウでプログラムを修正し，再度セーブし，
      もう一度このコマンドを実行. )
> gcc hello.c
> a.out            gcc コンパイラによって作成された実行可能ファイル
Hello.
>
```

注意: a.out で Command not found. というエラーが出たときは、次のようにする。

```
> ./a.out
Hello.
>
```

実行例 2

```
> gcc -o hello hello.c      実行可能ファイルの名前を指定
> hello
Hello.
>
```

図 3: hello.c の実行例

2.2 華氏から摂氏へ

アメリカ合州 (衆) 国などでは華氏で気温を表わすことが多い．目安として華氏 0 度と 50 度と 100 度を摂氏で表わしてみよう．華氏 x から摂氏への変換は，$\frac{5}{9}(x-32)$ で行なわれる．

```
1  /* fahren.c -- Farenheit/Centigrade Conversion */
2  #include <stdio.h>

3  void main(void)
4  {
5    printf("0 F is %f C\n", 5.0/9.0*(-32));
6    printf("50 F is %f C\n", 5.0/9.0*(50-32));
7    printf("100 F is %f C\n", 5.0/9.0*(100-32));
8  }
```

図 4: fahren.c

```
> gcc fahren.c
> a.out
0 F is -17.777778 C
50 F is 10.000000 C
100 F is 37.777778 C
>

注意: a.out で Command not found. というエラーが出たときは、
> ./a.out
```

図 5: fahren.c の実行例

1. `void main(void)`
   main は関数である．一般に関数は，入力パラメータを受け取って，計算を行い，ある値を戻すものであり，C 言語では次のように定義される．
   戻す値の型 関数名 (入力パラメータ (引数) 列){ 本体 }
   ところが，このプログラムのmain 関数は，入力パラメータも無く，また値も戻さないという，ちょっと変わった関数である．図 1のプログラムも同様で，main() のように括弧() の中に何もパラメータが書かず (引き数が空リスト)，また戻す型も指定しなかった．

   これを ANSI C では，void main(void) のようにも書く．先頭のvoid は関数main が戻す値の型を規定するが，これは “値を返さない” という型である．また(void) は，関数が何も引数を持たないことを示す．このように，void は二つの意味で使われるので注意を要する．

関数の型については後述する．

2. `printf("0 F is %f C\n", 5.0/9.0*(-32));`
これは，式`5.0/9.0*(-32)` の計算結果 (実数値) を画面 (標準出力) に表示する．まず式の値が 評価 (計算) される (計算式については後述)．その後`printf` 関数が呼び出される．この関数は，次の二つのパラメータを持っていることに注意．

   (a) `"0 F is %f C\n"`

   (b) `5.0/9.0*(-32)`

   `"0 F is %f C\n"`は，(前例の`"Hello"`のように) 基本的には画面に表示する文字列であるが，その中の`%f` は，そこに二つ目のパラメータ`5.0/9.0*(-32)` の値を表示することを意味する．

   具体的には，(b) 式の計算結果の実数値を画面に表示できる数字 (`0,1,2,...,9`) などからなる文字列に変換し出力する．`%f` のようなものを 制御文字列 という．また，この (a) のような表示の位置と形式を指定するパラメータを 変換仕様 という．

3 整数の四則演算

プログラミング入門の定番．二つの整数を入力し，その四則演算結果を表示する．ポイントは，変数，代入，そしてキーボードからの数値の読み込み．

```
1  /* arith.c -- arithmetic */
2  #include <stdio.h>

3  void main(void)
4  {
5    int m, n;
6    int s, d, p, q, r;

7    scanf("%d %d", &m, &n);
8    s = m+n;
9    d = m-n;
10   p = m*n;
11   q = m/n;
12   r = m%n;
13   printf("%d %d\n", m, n);
14   printf("%d %d %d %d %d\n", s, d, p, q, r);
15 }
```

図 6: arith.c

```
> gcc arith.c
> a.out
1994 6
1994 6
     2000     1988     11964       332       2
>
```

図 7: 実行の様子

1. `int m, n;` 変数宣言文
   変数宣言は，ある型をもつ変数を宣言する．ここで，int は，整数型を指定する 型指定子 であり，m，n は宣言される変数の名前，すなわち 変数名 である．この宣言によって変数が 定義 される (計算機の中に数値データを格納するための記憶領域が作られる)．

2. 整数のデータ型には，int，short，long の三つがある．

型	取り得る整数値の範囲
int	$(-2^{15} \sim 2^{15}-1$ または $-2^{31} \sim 2^{31}-1)$
short	$(-2^{15} \sim 2^{15}-1)$
long	$(-2^{31} \sim 2^{31}-1)$

なお，short はshort int，そしてlong はlong int のように書いてもよい．int の整数値範囲が二通りあることについては次の**記憶領域の大きさ**の節で述べる．

また，次のように 0 以上の (符合のない) 整数を表わす型も用意されている．

unsigned int	$(0 \sim 2^{16}-1)$ または $(0 \sim 2^{32}-1)$
unsigned short	$(0 \sim 2^{16}-1)$
unsigned long	$(0 \sim 2^{32}-1)$

例として，unsigned int m; のように使う．

3. 名前の規則
変数名や関数名は，英大文字，小文字，数字，下線からなる文字列で表す．ただし先頭は数字以外．習慣で下線も先頭に使わない．

4. scanf("%d %d", &m, &n); scanf 関数
キーボード (標準入力) からデータを変数 m，n に読み込む．
"%d %d" は，"1994 6"のようなキーボードから入力された文字列を，二つの整数に変換することを示す 変換仕様．他に%f %c などがある．
入力するデータの区切りは空白かタブか改行であり，コンマはダメ．○ "1994 6" × "1994,6"
コンマで区切りたければ，変換仕様として"%d,%d" を用いる．

&m，&n は，読み込んだ値を代入する変数 (m，n) に& を付けたもの．なぜ，& を付けるかについては後述するが，これは Pascal の変数引数に相当するもので，変数の所在 (アドレス) を示すもの．

5. s = m+n;
一般には，変数 = 式; の形をしており，右辺の式を評価し，その結果の値を左辺の変数の値にする代入文．

6. 算術演算子
+，-，*，/，%(加，減，乗，除，余り)．
整数同士の割算結果の小数点以下は切り捨て．

7. printf("%d %d\n", m, n);
変数 (式)m，n の値を画面 (標準出力) に表示．
"%d %d\n"の%d は，値を 10 進数で表示することを指定する制御文字列．

3.1 記憶領域の大きさ

int 型の表現の範囲は処理系 (計算機) によって異なる．それは一つのint 型に割り当てられるメモリーの大きさ (記憶単位) によって決まる (詳しくは第 15節で述べる)．sizeof 演算子は，sizeof(int のようにして，その大きさをバイト単位で返す．つまり，この値が 2 ならば，int の記憶単位は 2 バイト (16 ビット) であり，4 ならば 4 バイト (32 ビット) である．int，long などの整数型の記憶単位を調べるプログラムを作成してみよ．

4 浮動小数点 (floating-point number) の計算

```
 1  /* float.c -- area of triangle */
 2  #include <stdio.h>
 3  #include <math.h>

 4  void main(void)
 5  {
 6    float a, b, c, s, area;

 7    printf("Enter three lengths of the edges of a triangle. \n");
 8    scanf("%f %f %f", &a, &b, &c);
 9    s = (a+b+c)/2;
10    area = sqrt(s*(s-a)*(s-b)*(s-c));
11    printf("The area of the triangle is %f.\n", area);
12  }
```

図 8: float.c

1. `#include <math.h>` 標準数学関数用のヘッダファイル
   このプログラムでは，平方根を計算する関数sqrt を使っている．この関数は，C 言語のシステムが標準で持っている関数であるが，それを用いるためには，こられらの関数の型やパラメータに関して定義してあるヘッダファイルmath.h を取り込む．

2. `float a, b, c, s, area;`
   float 型とdouble 型は，浮動小数点を値とする変数の型である．

float	有効数字がおよそ 6 桁．指数は $10^{-37} \sim 10^{38}$ の範囲．
double	float よりも有効数字，範囲ともに大．
long double	もっと大．

3. `printf("Enter three lengths of the edges of a triangle.  \n");`
   2 重引用符 ”で囲まれた文字列を画面に出力する．\n は改行を表す記号．

4. `scanf("%f %f %f", &a, &b, &c);`
   キーボードからの三つの実数値を変数a，b，c に読み込む．float 型の変換仕様は%f．また，double 型の変換仕様は，%lf である．よって，変数a，b，c がdouble 型の時には，
   scanf("%lf %lf %lf", &a, &b, &c);

5. `s =(a+b+c)/2;`
   この式は，浮動小数点数(a+b+c) と整数の定数 (2) が混じっている混合演算．この場合，整数が浮動小数点数に変換されてから演算される．

6. `area = sqrt(s*(s-a)*(s-b)*(s-c));`
   sqrt は平方根の数学関数．

7. 標準ライブラリの数学関数には，以下のようなものがある．

`double cos( double x )`	余弦
`double acos( double x )`	逆余弦
`double cosh( double x )`	双曲線余弦
`double sin( double x )`	正弦
`double asin( double x )`	逆正弦
`double sinh( double x )`	双曲線正弦
`double tan( double x )`	正接
`double atan( double x )`	逆正接
`double atan2( double y, double x )`	y/x の逆正接
`double tanh( double x )`	双曲線正接
`double fabs( double x )`	絶対値
`double exp( double x )`	指数
`double log( double x )`	自然対数
`double log10( double x )`	常用対数
`double pow( double x, double y )`	x^y
`double sqrt( double x )`	$\sqrt{x}$
`double ceil( double x )`	x より小さくない最小の整数
`double floor( double x )`	x より大きくない最小の整数

8. `printf("The area of the triangle is %f.\n", area);`
`%f` は浮動小数点数のための変換仕様．`The area of the triangle is` と表示した後に，`%f.\n` に従い，`area` の値を表示してピリオド．を打ち，改行`\n` する．

記号`%` が変換仕様`f` を識別するのに使われている．`%` 自体を表示したいときには`%%` と書く．

```
> gcc float.c -lm
> a.out
Enter three lengths of the edges of a triangle.
3.1 4.2 5.3
The area of the triangle is 6.506613.
```

図 9: 実行の様子

図 9で注意すべき点は，`sqrt` のような数学関数を使ったプログラムをコンパイルするときには，ライブラリを指定する必要があるので，

```
% gcc float.c -lm
```

のように，`-lm` を指定することである．

ヘッダーファイルは，関数の型やパラメータに関する情報を含んでいるが，実際の関数値を計算するプログラムはライブラリに入っている．コンパイルに際し，上記のように`-lm` と指定することによってライブラリがリンクされる．

5　マクロ命令 / 前処理命令

ある容積の水が含む分子数を計算するプログラムを考える．ここでは，水 1 分子当たりの質量を表す定数 (2.99×10^{-23}) を計算に使っている．このような意味のある定数には名前を付ける．名前を付けないで，いきなり定数をプログラムに書いたものはマジックナンバーといって，プログラムの読み易さをそこなう．

```
1  /* water.c -- calculate the number of water molecules */
2  #include <stdio.h>
3  #define GRAM 1000 /* grams of 1 liter */
4  #define MASS 2.99e-23 /* mass of a single water molecule */

5  void main(void)
6  {
7    float water, molecule;
8    printf("Enter the amount of water in liter.\n");
9    scanf("%f", &water);
10   molecule = water*GRAM/MASS;
11   printf("The number of water molecules in %f liter(s) is %e.\n",
12  water, molecule);
13 }
```

図 10: water.c

```
> gcc water.c
> a.out
Enter the amount of water in liter.
1
The number of water molecules in 1.000000 liter(s) is 3.344482e+25.
```

図 11: 実行例

1. `#define GRAM 1000`　記号定数

   `#define` という マクロ命令 によって，これ以降，プログラム中の`GRAM` という文字列はすべて`1000` という整数の定数に置き換えられる．同様に，`#define MASS 2.99e-23` は，`MASS` という文字列を，`2.99e-23`，すなわち 2.99×10^{-23} という浮動小数点の値に置き換える．

   よって，　`molecule = water*GRAM/MASS;` という代入文は，
   `molecule = water*1000/2.99e-23`　に置き換えられる．

2. プリプロセッサ
   C 言語のプログラムがコンパイルされる時には，プログラムは，まずプリプロセッサと呼ばれるものによって処理される．プリプロセッサは，#include や#define などのマクロ命令に従って，ヘッダファイルを読み込んだり，記号定数の置換を行なう．置換された結果は、gcc -E water.c として見ることができる。

   なお，マクロ命令は C 言語の文ではないので，行末にセミコロン; はいらない．

3. プログラム中の定数は，なるべく記号定数にして，名前をつけ，マジックナンバーを残さないのがよい．慣習により，記号定数は大文字で書く．

4. %e 浮動小数点の変換仕様
   printf(... in %f liter(s) is %e.\n", water, molecule); において，二つ目の変換仕様%e は，浮動小数点の値を，3.344482e+25 のように仮数部と指数部で表示する．

5. 整数の定数 (整定数) の書き方

10 進数	0,1,2,3,4,5,6,7,8,9 の列	123
8 進数	0,1,2,3,4,5,6,7 の列．先頭に 0 を付ける．	0777
16 進数	0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F の列. 先頭に 0X または 0x をつける．	0X1234, 0XABCD

6 文字の処理

整数，浮動小数点数に加えて，もうひとつの基本的な型が文字型である．

```
1  /* char.c -- display character code */
2  #include <stdio.h>

3  void main(void)
4  {
5    char ch;
6    printf("Enter a character.\n");
7    scanf("%c",&ch);
8    printf("The character code for %c is %d.\n", ch, ch);
9  }
```

図 12: char.c

```
> gcc char.c
> a.out
Enter a character.
a
The character code for a is 97.
```

図 13: 実行例

1. 文字コード
   計算機の内部では，文字は文字コードという整数値によって表現されている．例えばa という文字は97(0x61) という整数で表される．

   文字コードは通常規格によって決められている．代表的なものに ASCII や JIS コードがある．最近，複数の言語の文字を統一的に表現できるコードが開発されている． ASCII コード表を下記に示す．
2. `char ch;` 文字型
   変数ch は一つの ASCII 文字コードを表すことができる．つまり， 0 〜 127(7 bits) の文字コードを表す．
3. `scanf("%c",&ch);`
   %c は文字型の値の読み込みのための変換仕様.
4. `printf("The character code for %c is %d.\n", ch, ch);`
   文字型の値の表示のための変換仕様.

%c　変数ch を文字として表示する．
%d　変数ch の文字コード (整数) を表示する．文字コードは整数であるので，%d はそれを 10 進表示する．

5. 文字定数
'a' のように，1 文字を' でくくったものを文字定数という．

これに対して，"abc"のように，文字の並びを"でくくったものを文字列という．

6. エスケープシーケンス
文字コードの中には，改行とか，バックスペースのように，画面に表示出来ない文字がある．これらを文字定数や文字列で使うには，例えば"Hello!\n"における \n のように書く．これをエスケープシーケンスという．

\a　ベルを鳴らす　　\b　バックスペース (後退)
\f　改ページ　　\n　改行
\r　行の先頭に戻る (復帰)

またその他の特別な書き方には，次のようなものがある．

\\　バックスラッシュ\　　\'　引用符'
\"　2 重引用符"　　\?　疑問符?
\ooo　8 進数 (例 '\7')　　\xhh　16 進数 (例 '\xD')

7. ASCII コード表

HEX	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
1	DEL	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	EC	FS	GS	RS	US
	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
2	SP	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
6	‘	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

この表の右上の HEX とは 16 進数 (hexa-decimal number) を表す。16 進数では、$0, 1, 2, \cdots, 9, A, B, C,$ によって、0 から 15 までの整数を表す。例えば、B は 11(10 進) を表し、$5C$ は、$16\times5+12 = 92$(10 進) を表す。

7　繰り返し

プログラムの実行を制御する構造の代表的なものに繰り返しと条件分岐がある．C には次の三つのタイプの繰り返しがある．

1. while
2. for
3. do while

ここでは，目の子平方を例としてこれらを説明する．目の子平方とは，正の整数の平方根の整数部分を求める方法である．それは，正の整数 a から，1,3,5,7, …と奇数を引けるだけ引いてゆく．その時の引くことができた回数 (引くことができた奇数の個数) が a の平方根の整数部分となる．例えば，13 ならば，$13-1-3-5$ のように三つの奇数を引くことができるので，13 の平方根の整数部分は 3 であることが分かる．

アルゴリズムとしては，繰り返しを使って，a の値が正の間，a から奇数を小さい順に繰り返し引いてゆく．その繰り返しの回数を数える．

7.1　入口で条件を調べる繰り返し (while 文)

```
1  /* menoko1.c -- menoko-heiho (while version) */
2  #include <stdio.h>

3  void main(void)
4  {
5    int x;
6    int odd = 1;      /* set the first odd number */
7    int count = 0;    /* clear the counter count */

8    printf("Enter a positive integer.\n");
9    scanf("%d", &x);
10   printf("The integral part of the root %d", x);
11   while (x - odd >= 0){
12     x = x - odd;
13     count = count + 1;
14     odd = odd + 2;
15   }
16   printf(" is %d\n", count);
17 }
```

図 14: menoko1.c

1. アルゴリズム

```
> gcc menoko1.c
> a.out
Enter a positive integer.
13
The integral part of the root 13 is 3
```

図 15: 実行例

- ここではwhile を使って繰り返しを行い、繰り返し回数はcount で表す．最初に0 を代入しておき，繰り返す度に1 ずつ増やす．つまり
  `count = count + 1`
- 奇数はodd で表し，最初に1 を代入しておき，繰り返す度に2 ずつ増やす．
  つまり`odd = odd + 2`
- 平方根を計算したい値をscanf でキーボードから変数x に読み込む。以下で、x の値は変更されてしまうので、その前に画面に`printf("The integral part of the root %d", x)`で表示しておく。この変換仕様には\n が無いので改行されない。したがって、プログラムの最後のprintf の出力は、同じ行に行われる。
- x から順々に奇数を引いてゆくには，繰り返す度に，`x = x - odd` とする．
- そして，`x - odd >= 0` の間，これらを繰り返す．

2. `int odd = 1;` 初期値設定を伴う変数宣言
   変数odd が宣言され、その初期値が1 になる。
3. `while (x - odd >= 0){ x = x - odd; count = count + 1; odd = odd + 2; }`
   while 文の一般形は，`while（条件）文`.
   ここで，条件は，`x - odd >= 0` という関係式であり，文は`{ }` で括られたブロック (複文)．この関係式が成り立つ間，ブロックの文が繰り返し実行される．
4. 関係式
   値の大小を比較する式．比較をするのが関係演算子で，次の 6 種.
   `< > <= => == !=` (それぞれ $<, >, \leq, \geq, =, \neq$)
5. 関係演算子の優先順序: 強い (`< > <= =>`)，弱い (`== !=`)
   同じ強さならば，左から右に．(例:`p!=q==r` は`(p!=q)==r` と同じ．
6. 関係演算子は，算術演算子 (`+,-,*` など) より弱く，代入演算子`=` より強い．
   例 `x+y==b` は`(x+y)==b` に同じ．`x=y==b` は`x=(y==b)` に同じ．
7. 代入演算子と代入式
   代入`x=y` は，`x=y;` のように通常は文として現れるが，`=` を演算子 (代入演算子) と考え，`x=y` を，`x+y` のような式と同等と考える．式であるから値をもつが，その値は代入に用いられた値である．これを代入式と言う．例) `(x=y)==b`, `x=y=z;`
   独立した文が現れるべきところに代入式がある場合には，代入文と解釈される．

8. 真偽値
C 言語では， Pascal の Boolean 型のような真偽値を表す型がない．偽を整数の0 で表し，真は0 以外の値としている．関係式は，真の時は整数の1 を値とし，偽の時は0 を値とする．例えば，x が 0 でない間繰り返す while は，`while (x != 0)` でもよいし，あるいは，もっと簡単に`while( x )` でよい．また`while(1)` は無限ループとなる．脱出には
`break`，`goto`，`return` などを使う．

7.2 C らしい簡略な記法: 代入演算子など

```
while (x - odd >= 0){
  x -= odd;
  count++;
  odd += 2;
}
```

図 16: C らしい簡略な記法をつかったプログラム

1. `x -= odd`
`-=` `+=` などのように 演算子 = の形をした 代入演算子 .
`x -= odd` は，`x = x - odd` と同じ．
2. `count++`
`++` `--` インクリメント演算子とデクリメント演算子
`count++` と `++count` は，被演算子 (count) の値に 1 加える．つまり、 `count=count + 1` と同じ。また、 `count--` と `--count` は，被演算子 (count) の値から 1 引く．
(注意) これらの演算子は変数にしか使えない．
3. 後置演算子の場合と前置演算子
`count++` のように変数の後につくものを後置演算子といい，`++count` のように前につくものを前置演算子という．

   前置演算子: 変数の値を (代入などに) 使う前に， 1 加えたり引いたりする．
   後置演算子: 変数の値を (代入などに) 使った後に， 1 加えたり引いたりする．
   例) `x = 1; y = ++x;` とすると，y には2 が代入されるが， `x = 1; y = x++;` とすると，1 が代入される．どちらも代入後のx の値は2.
4. (注意) `n*n++` という式は，`n++` をn よりも先に評価するかどうかで値が変わってくるので，一つの式に 2 度以上現れる変数に対してはインクリメント演算子やデクリメント演算子は使わないようにする．同様に，`f(n,n++)` のように関数の引数に 2 度以上現れる場合も同じ．

7.3 入口で条件を調べる繰り返し (for 文)

同じ例題を，今度は繰り返しの入り口 (各繰り返し毎の最初) で条件を調べる`for 文`で計算してみる．

```
1  /* menoko2.c -- menoko-heiho (for version) */
2  #include <stdio.h>

3  void main(void)
4  {
5    int x;
6    int odd;      /* set the first odd number */
7    int count = 0;    /* clear the counter count */

8    printf("Enter a positive integer.\n");
9    scanf("%d", &x);
10   printf("The integral part of the root %d", x);
11   for(odd = 1; x - odd >=0; odd += 2){
12     x -= odd;
13     count++;
14   }
15   printf(" is %d\n", count);
16 }
```

図 17: menoko2.c

1. `for(odd = 1; x - odd >=0; odd += 2){ x -= odd; count++; }`
   `for 文`の一般形は，for(初期設定; 条件; 更新) 文
   これは，ちょっとややこしい格好をしている．for の後の括弧の中には，三つの要素 (初期設定，条件，更新) が指定され，その後に繰り返し実行される文が指定される．

   このプログラムでは，

   - 初期設定はodd = 1 という代入式
   - 条件は，x - odd >= 0 という関係式
   - 更新はodd += 2 という式
   - 文は{ x -= odd; count++; } という複文 (ブロック)

   意味は，まず初期設定の式を評価し，次に条件を調べ，条件が真であれば文を実行する．実行が終わったら，更新の式を評価し，条件を調べる．真であれば，また文を実行する．これを条件が偽になるまで繰り返す．

   - このプログラムでは，odd の初期設定をfor 文でやっているので，変数宣言のところでは，初期設定を行なっていない．

- for 文のodd = 1; は代入文のように見えるが，この; は，初期設定と条件を区切るセミコロンであり，(代入文と似ているが) odd = 1; は代入式である．

2. コンマ式
   odd の初期化がfor で行えるなら，count = 0 の初期設定もfor で行ないたい．そのためには，複数の式をコンマ , でつないで一つの式として扱うコンマ式を使う．これを使うと，

   ```
   for(count = 0,odd = 1; x - odd >=0; odd += 2){....}
   ```

   このコンマを コンマ演算子 という．コンマ式は左から右へと評価される．また，更新式にもコンマ式を使えるので，count++ もfor の中に入れてしまい，次のようにfor ループの文を空にすることもできる．

   ```
   for(count = 0, odd = 1; x - odd >=0; x-=odd, count++, odd += 2) ;
   ```

   ここで，更新式のコンマ式の順序に注意.

   しかし，これはあまり勧められない．なぜなら，プログラムの意図が分かりにくくなるから．ここでは，回数を数えるということを明確にし，条件のところに代入式を使った次の例が適当と思われる．

   ```
   for(count = 0, odd = 1; (x -= odd) >=0; odd += 2) count++;
   ```

   ここでは，x - odd >=0; x-=odd を一つにして，(x -= odd) >=0 にしている．なお，-= の優先度は>= の優先度よりも低いので，(x -= odd) のようにカッコで括る必要がある．つまり，x -= odd >=0 はダメ．また，この場合は，繰り返す文は 1 つ (count++) なので，{ } でくくる必要はない．

3. for(初期設定; 条件; 更新) において，初期設定や条件，更新は不要ならば書かなくてもよい．その場合でも ; だけは書く．例えば，for(;;) は無限ループを表わす．

式に関するまとめ

1. 演算子 (算術演算子，代入演算子，関係演算子，コンマ演算子)
2. 演算子の強さ
   強い ( ) > + - ++ --(単項演算子) > * / % > + -
   > < > <= => > == != > = *= /= %= += -= > , 弱い
3. 式の値

算術式	評価 (計算) 結果の値
関係式	真の式は 1，偽の式は 0
代入式	= の左辺に代入される値
コンマ式	右はじの式の値

7.4 出口で条件を調べる繰り返し (do while 文)

```
 1  /* menoko3.c -- menoko-heiho (do-while version) */
 2  #include <stdio.h>

 3  void main(void){

 4    int x;
 5    int odd=1; /* set the first odd number */
 6    int count=0; /* clear the counter count */

 7    printf("Enter a positive integer.\n");
 8    scanf("%d", &x);
 9    printf("The integral part of the root %d", x);
10    do {
11      x -= odd;
12      ++count;
13      odd += 2;
14    } while (x >=0);
15    --count;
16    printf(" is %d\n", count);
17  }
```

図 18: menoko3.c

1. `do 文 while (条件)`
   条件をループの最後におき，各繰り返しの後 (各繰り返し毎の文の実行の後) にその条件を調べる．各繰り返しの後で条件を調べるので，最初の 1 回は必ず実行される．

2. プログラムについての注意

   - `++count` と`count++`
     (変数の値の参照を伴わない) 単独の `++count` は，`count++` と書いても同じ．
   - このプログラムではx の値が負になった時に繰り返しは終了する．つまり，1 回余計に繰り返し`count` の値が 1 だけ大きくなるので，15 行目で`--count` として帳尻を合わせる．

3. セミコロンについて

   - C ではセミコロンは文の終了を示す (Pascal では文の区切り (セパレータ) であった).
   - `++count` などの式は，その後にセミコロンをつけると文になる．

式	`++count`	`x=1`	`printf("hello\n")`
文	`++count;`	`x=1;`	`printf("hello\n");`

   - 複文を表す`{ }` の右括弧`}` の後にはセミコロンは付けない．次に例を示す．

```
do
  ++count;
while (count < 10);
```

```
do {
  ++count; printf("*");
} while(count < 10);
```

8 応用: データ数が予め特定できない場合のデータ処理

```
 1  /* mean.c -- mean and standard deviation */
 2  #include <stdio.h>
 3  #include <math.h>

 4  void main(void)
 5  {
 6    int n=0;
 7    float x;
 8    float mean, sd;
 9    float sum=0, square=0;

10    printf("Enter floating point data. \n");
11    printf("Enter q to finish the input. \n");
12    while (scanf("%f",&x)==1){
13      sum += x;
14      square += x*x;
15      ++n;
16    }
17    mean = sum/n;
18    sd = sqrt( (double)((square - sum*mean)/(n-1)));
19    printf("The number of data read is %d.\n", n);
20    printf("The mean value and the standard deviation of those data");
21    printf(" are\n %10.2e and %10.2e.\n", mean, sd);
22  }
```

図 19: mean.c

```
> gcc mean.c -lm
> a.out
Enter floating point data.
Enter q to finish the input.
3.14 9.81 273.15 8.31 22.41 q
The number of data read is 5.
The mean value and the standard deviation of those data are
   6.34e+01 and   1.17e+02.
```

図 20: 実行例

1. N 個のデータ (x_i) の平均と標準偏差.
   総和 $S = \Sigma x_i$，平均 $\bar{x} = S/N$，標準偏差 $= \sqrt{(\Sigma x_i^2 - S\bar{x})/(N-1)}$
2. このプログラムでは，データの個数が予めわからないので，キーボードから数値データを入力してゆき，最後のデータを入力したら，データ入力終了の意味で文字q を入力するようにしている.
3. 各繰り返しで，数値をscanf によって変数x に入力し，その和と自乗の和を，`sum += x;` と `square += x*x;` によって求めてゆく．またデータを読み込む度に，n++ によってカウントする.
4. 変数の宣言において，`sum`， `square`，`n` は，全て 0 に初期化されている.
5. `while (scanf("%f",&x)==1)`
   このwhile の繰り返しの終了条件について説明する．このプログラムでは，数値データの入力が終ったら，数値の代わりにq を入力する．q は数値ではなく文字なので，変換仕様%f が与えられたscanf は，q を数値に変換することができない．このような場合，scanf は，読み込む (変換する) ことができた項目 (変数) の数を返す．この場合は項目は&x だけなので，変換が出来ない (読み込めない) 場合，scanf は0 を返し，また数値が入力されて変換が出来た (読み込むことができた) 場合には，1 を返す．(関数の返す値を返戻値という).
6. EOF の利用
   より一般的な方法としては， EOF(End Of File)(CTRL-D, MS-DOS ではCTRL-Z) を入力する方法がある． EOF を読みとると，scanf は返戻値として，stdio.h で
   #define されたEOF(通常は -1) という値を返す．よって，データの最後まで読みとるという繰り返しは，次のように書くことができる． `while (scanf("%f",&x)!=EOF)`
7. キャスト (cast) 演算子による型変換
   `sd = sqrt( (double)((square - sum*mean)/(n-1)));`
   関数sqrt の引数はdouble 型を仮定している．ここでは，square，sum，mean などの値はfloat 型なので，式(square - sum*mean)/(n-1) の計算結果も
   float 型になる．そこで，その式の前に （double） というキャスト演算子をつけて，値の型をdouble 型に変換する.
8. 変換仕様%10.2e
   %e に修飾子10.2 をつけたもの．数値を全体として最大 10 桁表示し，そのうち小数を 2 桁表示するという意味．例: 1.17e+02

8.1 型変換の規則のまとめ

1. char とshort が一つの式に現れた時，自動的にint に変換されるか，必要ならばunsigned int に変換される.
2. 二つの異なる型を含む演算では『より広い』方の型へと変換される．これを格上げ(promotion) という.

   広い long double > double > float > unsinged long > long > unsigned int > int 狭い
3. より狭い型への代入は，問題が生じる．ただし，浮動小数点から，整数への代入の場合は，小数点以下が切り捨てられるのでエラーにはならない.
4. 関数の実引数は，仮引数の型へ自動的には変換されない.

9 応用: 関数のグラフ

繰り返しを使って，曲線のグラフを書いてみる．

1. 曲線は折れ線として近似する．例えば，関数 $f(x)$ を，$0 \leq x \leq 1$ の範囲で表示するには，$x_0 = 0, x_1 = 0.01, x_2 = 0.02, \cdots$ のように，x の値を少しずつ増やしながら，$f(x)$ を計算する．点列 $(x_i, f(x_i))$ を結ぶ折れ線が，関数 $f(x)$ の近似曲線になっている．

2. グラフを画面に表示するには，xgraph というツールを使う．これは，点列を読み込むと，自動的に座標軸などをつけてグラフを書いてくれるツールである．

3. 作成するプログラムは，関数 $y = f(x)$ に対して，x_i, y_i の値を次々と計算して出力するようにし，その出力をxgraph に入力する．プログラムは図 21のようになる．

   graph.c で計算した結果は，図 22のように，パイプで，xgraph コマンドに入力する．

```
 1  /* graph.c -- drawing a graph for sin function */
 2  #include <stdio.h>
 3  #include <math.h>
 4  #define PI 3.1415926

 5  void main(void)
 6  {
 7    double x = 0.0;
 8    while (x < 2*PI)
 9      {
10        printf("%e  %e\n", x, sin(x));
11        x += 0.1;
12        }
13  }
```

図 21: graph.c

```
> gcc graph.c -lm
> a.out | xgraph            画面にウィンドウが開いて表示される．
>
```

図 22: graph.c の実行例

10 条件分岐 (if 文，if else 文)

繰り返しに加えて，もう一つの実行制御構造が条件分岐である．条件分岐にはif 文とswitch 文がある．最初の例は，入力された整数が奇数かどうか判定するものである．

```
1  /* if1.c -- an odd number? */

2  #include <stdio.h>
3  void main(void)
4  {
5    int a;

6    printf("Enter a number: ");
7    scanf("%d", &a);
8    if ( (a%2) == 1) printf("Odd number\n");
9  }
```

図 23: if1.c

1. `if ( (a%2) == 1) printf("Odd number");` if 文
   関係式 (`(a%2) == 1`) が真 (非ゼロ) ならば文 (`printf("Odd number");`) を実行する．複数の文を実行するには，それらを`{}` で括ってブロック (複文) にする．

```
1  /* if2.c -- odd or even? */

2  #include <stdio.h>
3  void main(void)
4  {
5    int a;

6    printf("Enter a number: ");
7    scanf("%d", &a);
8    if ( (a%2) == 1) printf("Odd number\n");
9    else printf("Even number\n");
10 }
```

図 24: if2.c

if else 文

1. `if ( (a%2) == 1) printf("Odd number\n"); else printf("Even number\n");`
   `if` (条件) 文 $_1$ `else` 文 $_2$

条件が真 (非ゼロ) ならば文 $_1$ を実行し，そうでなければ文 $_2$ を実行する．

if 文の補足

1. if 文の入れ子: if 文や上記のif else 文で実行される文の中に，またif 文やif else 文が現れる場合がある．
2. else は，その前にあって一番手前のif と結びつけられる．

```
例) if (P) if (Q) x; else y;

if (P)              if (P)              if (P) {
  if (Q) x;           if (Q) x;           if (Q) x;}
else y;             else y;             else y;
```

左と真中は同じプログラムだが，段付け (indentation) (空白を入れて段を付けること) は真中がよい．
右は他の二つとは別のプログラムで，if (Q) x; を{} で囲んでいるので，else は最初のif に結び付く．} の後にはセミコロンは不要．

3. 条件演算子
条件文 if (x > 0) y = x; else y = -x; をy = (x > 0) ? x: -x; のように書くことができる．これを条件式という．条件式は一般に，式 $_1$? 式 $_2$: 式 $_3$ と書き，式 $_1$ の値が真 (非零) ならば式 $_2$ の値をもち，そうでなければ，式 $_3$ の値をもつ．? : を条件演算子という．

11 応用:2 次方程式の解法

else-if 構造を連続して使って、場合分けをすることができる。

```
if (P) a
else if (Q) b;
else if (R) c;
else if (S) d;
else e;
```

図 25は，2 次方程式の解法プログラムで、プログラムの"if(det == 0)"で始まるif 文は，この場合分の構造になっている．また、プログラム中のfabs は絶対値をとる関数．

```
1  /* quad1.c -- quadratic equation */

2  #include <stdio.h>
3  #include <math.h>

4  void main(void)
5  {
6    int a, b, c;
7    int det;
8    float d;

9    printf("Enter the three coefficients");
10   printf("of a quadratic equation.\n");
11   scanf("%d %d %d", &a, &b, &c);
12   if (a != 0){
13     det = b*b - 4*a*c;
14     if( det == 0)
15       printf("This equation has one real root: %10.2f.\n",
16      -b/(2.0*a));
17     else if(det > 0){
18       d = sqrt((double)det);
19       printf("This equation has two real roots:");
20       printf("%10.2f and %10.2f.\n",
21      (-b+d)/(2.0*a), (-b-d)/(2.0*a));
22     } else if(det < 0){
23       d = sqrt(fabs((double)det));
24       printf("This equation has two complex roots.\n");
25       printf("The real part is %10.2f ",-b/(2.0*a));
26       printf("and the imaginary part is %10.2f.\n ",d/(2.0*a));
27     }
28   }
29 }
```

図 25: quad1.c

12 応用: ワードカウントプログラム

この例は，Unix のwc コマンドと同じように，テキストファイルに含まれている字数，語数，行数を求めるプログラムである．ファイルからプログラムにデータを読み込む方法については，まだ説明していないが，キーボード (標準入力) からデータを読みこむプログラムは，a.out < winner.txt のようにリダイレクションを使うことによって，ファイルからも読み込める (実行例参照).

```
 1  /* wc.c -- word count */
 2  #include <stdio.h>
 3  #include <ctype.h>
 4  #define YES 1
 5  #define NO 0

 6  void main(void)
 7  {
 8    int c;
 9    long chars = 0;
10    int lines = 0;
11    int words = 0;
12    int inWord = NO;

13    while( (c=getchar()) !=  EOF ) {
14      ++chars;
15      if (isspace(c)){
16        inWord = NO;
17        if( c == '\n') ++lines;
18      }
19      else if (inWord == NO){
20        inWord = YES;
21        ++words;
22      }
23    }
24    printf("The number of characters, words, and lines are ");
25    printf("%ld, %d, and %d.\n", chars, words, lines);
26  }
```

図 26: wc.c

どのようにして文字数，行数，単語数を数えるかがポイントである．

1. `while( (c=getchar()) != EOF )`

   - `getchar()` 関数

```
> gcc wc.c
> a.out < winner.txt          (適当なファイル名を < の後に書く)
The number of characters, words, and lines are 543, 103, and 17.
```

図 27: 実行例

関数getchar() は，標準入力 (キーボードまたは，リダイレクションの場合はファイル) から 1 文字ずつ読んでゆく．尚，1 文字出力する関数にputchar(c) がある．

- EOF
  リダイレクションの場合に，ファイルの最後まで読んでしまうと，getchar はEOF という値を返す． EOF は，ヘッダーファイルstido.h で -1 という値に定義されている．
- c=getchar()
  getchar の返戻値を代入する変数c はint 型である．getchar() は文字を読み込む関数なので，char 型でもよさそうであるが，char 型では，0 から 255 の値しか扱うことができないため，EOF(-1) の値を扱えないので，int 型にしてある．

2. `++chars`
   読み込んだ文字数は， chars という変数をカウンターにして， 1 文字読み込む毎にインクリメント (+1) する．文字数は大きくなるので，char はlong 型にしてある．EOF まで読み込むことができた文字の数が文字数となる．
3. `if( c == '\n') ++lines;`
   行数はどのようにして数えるのだろうか? ファイルでは，行と行の間 (行の区切り) のところには，'\n' という文字 (改行記号) が入っている．getchar は，行の区切りに来ると，この'\n' という値を戻すので，その数を数えれば，それが行数になる．ここでは，lines という変数をカウンターにして，'\n' という値の度にインクリメントする．
4. さて，もっとも難しいのが単語数の数え方である．

   - まず，単語とはスペースで区切られた文字列 (空白を含まない文字の並び) と考える．単語は，スペース以外の文字が最初に現れたところから，次にスペースが現れた所までの文字列である．よって，words という変数をカウンターにして，単語が始まるところを数えていく．

     では，スペースとは何か? 空白はスペースであるが，空白だけがスペースではない．例えば，

     ```
     As the rose is the flower of flowers, so is the
     house of houses.
     ```

     のようなファイルがあったときに， 1 行目の最後の the と 2 行目の house の間には，改行記号が入っている．これも単語を区切るスペースと考えるべきである．タブ記号 (後述) もスペース文字である．

- このプログラムでは，ある文字がスペース文字かどうかを判定するのに，ctype.hで定義されているisspace(c) 関数を使っている．引数の文字がスペースであれば真 (1)，そうでなければ偽 (0) を返す関数で，具体的には次のように定義されている．

  if (isspace(c)) ⟺ if (c==' '||c=='\n'||c=='\t')

  ここで|| は，論理演算子で，論理和 (OR) を表す．他の論理演算子として，論理積&&と，否定! がある．

  また，'\t' はタブ (tab, CTRL-I) を表わす．これは，ある決まった位置まで空白を開けるもので，表などを作成するために使うことがある．

- また，isspace と同種のものに，isalpha，isdigit などがあるが，これらも，ヘッダーctype.h で定義されている．

ここでは単語数を数えるのに，inWord という変数を用意し，単語が始まった所で値をYES にし，終ったところでNO にする．終った所とは，読み込んだ文字がスペースである (isspace が真である) 所であり，また単語が始まる所とは，その直前までinWord がNO で，かつスペース以外の文字を読み込んだ所 (isspace が偽) である．尚，inWord の初期値はNO とする．

```
          The winner  is always a part of the answer.\n
inWord: NYYYNYYYYYYNNYYNYYYYYYNYNYYYYNYYNYYYNYYYYYYYN
words:  01111222222223334444444556666677788889999999
```

このinWord のように，状態に応じて真偽 (YES/NO) をセットして使う変数をフラグ (flag) という．

5. printf の変換仕様 %ld は，long int 型に対する変換仕様である．

13 switch 文による条件分岐

上記のプログラムではif 文によって場合分けをしていたが，switch 文を用いると，すっきりと書ける．プログラムの"if (isspace(c))"から始まる部分は，図 28に示すようなswitch 文を使って書き直せる．

1. この例でみると，switch(c) は，case の直後に示された定数式 (ラベル) とc の値とが等しい場合に，ラベルの後の文を実行するものである．定数式は整数型か文字型に限られる．尚，ラベルの後にはコロン: をつける．

   switch 文の一般形は次のようになる．

```
switch (式) {
  case ラベル: 文 … 文
  case ラベル: 文 … 文
  default: 文 … 文
  }
```

2. 各case の後に続く文の中に，break 文が指定してある．このbreak 文は，そこでswitch 文の実行を終了する．逆にbreak がないと，次のcase の文も実行してしまう．例えば，このプログラムのcase ' ': の部分には，文が何も書かれていないが，何も行なわないのではなく，その次のcase '\t': の部分の文を実行する．何もしないようにするには，case ' ': break; とする．

```
switch(c) {
  case '\n':
    inWord = NO;
    ++lines;
    break;
  case ' ':
  case '\t':
    inWord = NO;
    break;
  default:
    if (inWord == NO){
      inWord = YES;
      ++words;
    }
    break;
  }
```

図 28: switch の例

3. どのラベルにも合致せず，しかもdefault がある場合には，そのdefault: の後の文が実行される．

4. 一番最後のcase あるいはdefault に続く部分ののbreak は必ずしも必要でないが慣習で付ける．

14　関数

これまではmain 関数一つだけからなるプログラムを紹介してきたが，ここでは多くの関数を定義し，それらを呼び出して計算を行うようなプログラムについて説明する．

14.1　関数と引数

1. 図 25の 2 次方程式の根を求めるプログラムを，関数を用いて書いたものを図 29に示す。関数名はquad で， 3 つの引数a，b，c を持つ．
2. プロトタイプ宣言
   関数を定義するには，プログラムの最初の方で，かつヘッダファイルの指定の次のあたりで，これから定義しようとする関数の型 (返戻値の型)，関数の名前，関数の引数などを宣言する．これをプロトタイプ宣言という．

   quad 関数の場合は，プログラムに示したように， `void quad(int a, int b, int c);` となる． `void` は，関数 `quad` が値を返さないことを示す．このような関数を手続きともいう．
3. 関数定義
   関数定義の一般的な形は，次のようになる．プログラムにあるquad の定義を参照しつつ確認してほしい．

```
返戻値の型 関数名(仮引数１の型 仮引数名１，仮引数２の型 仮引数名２，…，仮引数ｎの型 仮引数名ｎ)
{
  宣言 … 宣言                    関数の本体
  文 … 文
  }
```

4. 関数の呼び出しと引数
   定義された関数quad は，main 関数の中で，`quad(a, b, c);` のようにして呼び出される．このようにして呼び出すのは値を戻さない関数 (手続き) か，値は戻すがそれを使わない場合である．(これは実はすでにscanf 関数の例で見ている．`scanf("&d", &a)` とする場合と，`scanf("&d", &a)==1` のように関係式の中で値を使う場合があった．) 値を戻す関数の場合には，式中で呼び出すこともできる．

   引数については， Pascal などと基本的な考え方は同じである．`quad(a, b, c);` のa，b，c の値 (実引数) が`void quad(int a, int b, int c)` のa，b，c(仮引数) に渡される．この場合はたまたま実引数と仮引数が同じ名前になっているが，実引数は一般に式でよいし，仮引数も好きな名前を使ってよい．
5. 変数の有効範囲
   関数の本体は`{ }` で囲まれたブロックとなっている．このブロックで宣言された変数 (これ例ではdet，d) は，そのブロックの中でしか使えない (“有効である” という)．また，関数の引数も，その関数の中だけで有効である．このようにある変数を使用できる範囲をその変数の有効範囲 (scope) という．有効範囲が異なる変数は，たとえ同じ名前をしていても，異なる変数である．例えばmain の中の変数a,b,c は，quad の引数a,b,c とは別のものである．

```
 1  /* quad2.c -- quadratic equaiton (function version) */
 2  #include <stdio.h>
 3  #include <math.h>

 4  void quad(int a, int b, int c);

 5  void main(void)
 6  {
 7    int a,b,c;

 8    printf("Enter the three coefficients");
 9    printf("of a quadratic equation.\n");
10    scanf("%d %d %d", &a, &b, &c);
11    if (a != 0)
12      quad(a, b, c);
13    else
14      printf("The value of an a must not be zero.\n");
15  }

16  void quad(int a, int b, int c)
17  {
18    int det;
19    float d;

20    det = b*b - 4*a*c;
21    if( det == 0)
22      printf("This equation has one real root: %10.2f.\n",
23             -b/(2.0*a));
24    else if(det > 0){
25      d = sqrt((double)det);
26      printf("This equation has two real roots:");
27      printf("%10.2f and %10.2f.\n",
28     (-b+d)/(2.0*a), (-b-d)/(2.0*a));
29    } else if(det < 0){
30      d = sqrt(fabs((double)det));
31      printf("This equation has two complex roots.\n");
32      printf("The real part is %10.2f ",-b/(2.0*a));
33      printf("and the imaginary part is %10.2f.\n ",d/(2.0*a));
34    }
35  }
```

図 29: quad2.c

14.2 値を戻す関数

ユークリッドの互除法を例題として，値を戻す関数について説明する．このユークリッドの互除法というのは，ゼロでない 2 つの整数 a, b の最大公約数 (GCD, greatest common divider)

を求めるものである．いま，$a > b$ のとき，a, b の最大公約数は，$a - b$ と b との最大公約数になる．x, y の最大公約数を (x, y) と表すと，
$(24, 18) \Rightarrow (18, 24 - 18) = (18, 6) \Rightarrow (18 - 6, 6) = (12, 6) \Rightarrow (12 - 6, 6) = (6, 6) = 6$
　このプログラムを図 30に示す。

```
 1  /* gcd.c -- greatest common divisor */
 2  #include <stdio.h>

 3  int gcd(int a, int b);

 4  void main(void)
 5  {
 6    int x, y, z;
 7    int gcd(int, int);
 8    printf("Enter two non-zero integers.\n");
 9    scanf("%d %d", &x, &y);
10    if( x > 0 && y > 0){
11      z = gcd(x, y);
12      printf("The GCD of %d and %d is %d\n", x, y, z);}
13    else
14      printf("Both x and y must be positive.\n");
15  }

16  int gcd(int a, int b)
17  {
18    while (a != b){
19      if (a > b)
20        a -= b;
21      else
22        b -= a;
23    }
24    return a;
25  }
```

図 30: gcd.c

1. `int gcd(int a, int b);`
   これは，int 型の引数を二つもち，int 型の値を返す関数gcd のプロトタイプ宣言．引数名は書いても書かなくてもよい．書かない場合には，
   `int gcd(int, int);`
2. 7 行目の `int gcd(int, int);`
   プロトタイプ宣言とは別に，関数において，その関数で利用する関数の名前と引数を，変数などの宣言に加えて宣言しても良い．
3. `if( x > 0 && y > 0)` && 論理演算子
   && の論理積 (AND) を表す．論理和は|| で，否定は!
4. `return a;`
   関数gcd は最大公約数を求めて，それを返戻値とする．`return 文` は，そこで関数の実行

```
> gcc gcd.c
> a.out
Enter two non-zero integers.
24 18
The GCD of 24 and 18 is 6
>
```

図 31: 実行例

を終了し，呼び出された所に関数の返戻値としてa の値を返すことを意味する．値を返さない関数 (void 型) の場合は，return; と書くか，または何も書かない (例: 図 29 の関数quad).

5. 関数定義と関数宣言
   3 行目の プロトタイプ宣言 は，関数の返戻値の型や引数の型などを指示するものであり，19 行目からの 関数定義 は，関数の実際のプログラムを記述するものである．

6. 関数呼び出し
   このプログラムでは，gcd 関数は代入文z = gcd(x, y); の式中に現れている．この式が評価される時，gcd 関数が呼び出される．

   このように式の中で関数を呼び出す時には，gcd(a,b) のように，関数名の後に() でパラメタを括って呼び出す．この引数を 実引数 という．

   これに対して，関数定義における引数を 仮引数 という．実引数は，一般に式でよいが，式を評価した結果の値の型は，対応する仮引数の型に一致していなければならない．ただし整数については，int⇒long の 格上げ (promotion) が行なわれ，float はdouble に変換される．

15　変数の値とポインタ

```
 1  /* neg1.c   change the sign of an integer*/
 2  /* THIS PROGRAM IS WRONG!! */
 3  #include <stdio.h>

 4  void neg1(int);

 5  void main(void)
 6  {
 7    int a=19;

 8    printf("Initially,  a = %d\n", a);
 9    neg1(a);
10    printf("After neg1, a = %d\n", a);
11  }

12  void neg1(int x)
13  {
14    printf("In neg1, initially, x = %d\n", x);
15    x = -x;
16    printf("In neg1, finally,   x = %d\n", x);
17  }
```

図 32: neg1.c

```
> gcc neg1.c
> a.out
Initially,  a = 19
In neg1, initially, x = 19
In neg1, finally,   x = -19            <-- OK
After neg1, a = 19                     <-- NG
>
```

図 33: 実行例

変数の値の符合を反対にする関数，つまり＋を－に，－を＋にする関数について考えてみる．変数x の符合を反転するには，代入文を使って `x = -x;` とすればよい．そこで，これを使って関数neg1 を図 32のように定義したが，図 33に示すようにうまくいかなかった．つまり，関数neg1

の中では，確かにx の符合は19 から-19 になって反転しているのに，neg1 の実行終了後に，関数main に戻ってみると，x の値は19 のままなのである．

これを解決するには図 34のneg2 のようなプログラムにする必要がある．

```
 1  /* neg2.c   change the sign of an integer*/
 2  #include <stdio.h>

 3  void neg2(int *x);

 4  void main(void)
 5  {
 6    int a=19;

 7    printf("Initially,  a = %d\n", a);
 8    neg2(&a);
 9    printf("After neg2, a = %d\n", a);
10  }

11  void neg2(int *x)
12  {
13    printf("In neg2, initially, x = %d\n", x);
14    *x = -*x;
15    printf("In neg2, finally,   x = %d\n", x);
16  }
```

図 34: neg2.c

```
> gcc neg2.c
> a.out
Initially,  a = 19
In neg2, initially, x = -268437052     <-- ?
In neg2, finally,   x = -268437052     <-- ?
After neg2, a = -19                    <-- OK
>
```

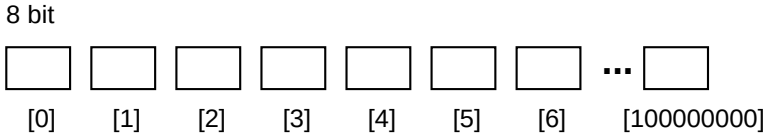
図 35: 実行例

neg1 とneg2 の違いは，引数の渡し方にある．それは，値渡し (値引数) と アドレス渡し (変数引数) の違いであるが，この違いを理解するには，計算機の中で変数がどのように扱われているかを知る必要がある．しばらくその話をしよう．

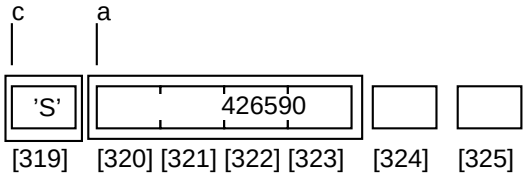
1. メモリ
   計算機の内部には，データやプログラムを記憶するためのメモリがある．メモリの最小単位は，2 進数 1 桁，つまり，0 か 1 かを記憶するもので，これを 1 ビット という．これを 8 個ならべて，2 進数 8 桁分のメモリ，つまり 8 ビットを 1 バイトと呼ぶ．1 バイトのメモリは，2 進数の00000000 から11111111，10 進数でいうと，0 から 255 までの値を表すことができる．

2. アドレス
   計算機の中には記憶装置があって，この 1 バイトを単位としてメモリーが一列に並んでいる．最近のワークステーションでは，数十メガバイト (1 千万個から一億個) 程度は持っている．

   この記憶単位に先頭から 0,1,2,3,... のように番号を付ける．これを アドレス (番地) という．計算機はアドレスを指定することによって，メモリの内容を参照したり，書き変えたりする．下図では，アドレスを[1] のように括弧付きで示す．



3. 計算機がプログラムを実行する時，このメモリに変数の値が格納される．別の言い方をすると，メモリを変数に対して 割り当てる． 例えば下図で，char 型の変数c には，1 記憶単位 (1 バイト) が割り当てられ，'S' などの文字 (文字コード) が記憶される．またint 型の変数a には，4 バイト (処理系によっては 2) が割り当てられ，整数値が表現される．(一般に，ある型にどれくらいのバイト数が割り当てられるかは，sizeof 関数で調べることができる．)



4. このように変数には，値とアドレスという二つの性質がある．変数a の値は，a+1.2 のように式の中で"a" によって表される．そして，アドレスは アドレス演算子& を使って，&a で得られる．a のアドレスのことを，a への ポインタ ともいう．

5. 値渡し
   前のプログラムでは，neg1 にはa の値が渡され，neg2 にはa のポインタが渡されている．では，neg1 について詳しく検討してみよう．関数呼び出しneg1(a) が実行されると，実引数a の値が，関数neg1 の仮引数x に渡される．変数の値を渡すので，これを 値渡し という．仮引数x は，整数値を受けとるので，int 型に宣言されている．

   関数neg1 の中では，代入文x = -x; によって変数x の値の符合を変えている．しかし，肝心の変数a の方は値を渡しただけなので，いくらx の値を交換しても，a の値は変らない．

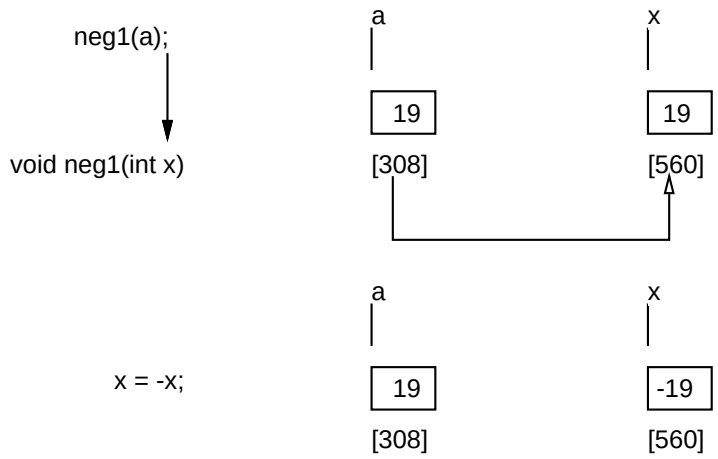


図 36: neg1 の場合の値渡し

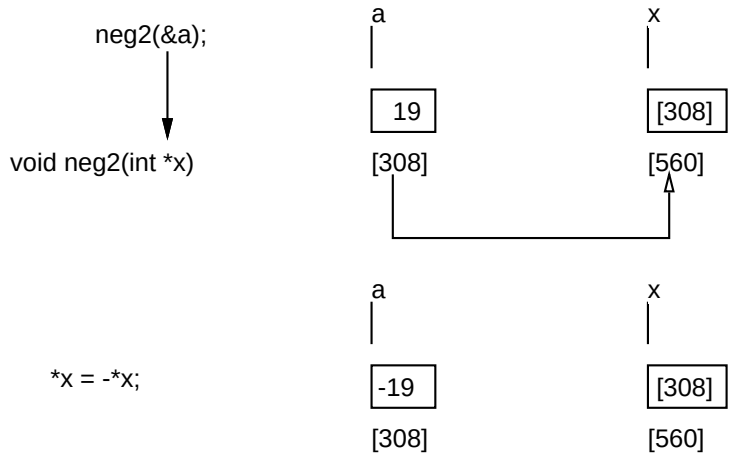


図 37: neg2 の場合のアドレス (ポインタ) 渡し

6. アドレス (ポインタ) 渡し
   a の値を変更するには，変数a の値を渡すのではなく，それらのアドレスを渡してやる．これがneg2 である．

   この場合，仮引数のx は，int 型の値を受けとるのではなく，int 型の変数のアドレス (&a) を受けとるので，x の型はint 型ではまずい．それは，int 型の値をもつ変数のアドレス (int 型の値へのポインタ) を表す型でなければならない．このような変数は，`int *x` のように宣言される．

   このint *x という宣言は，『*x(ポインタ型の変数x の表すアドレスに格納されているデータ) がint 型である』とか，『x は，int 型を指す (*) ポインタ型の変数である』と読むことができる．

7. neg2 では，ポインタ型の変数x は，変数a のアドレスが渡される．このx を使って，変数a の値を参照したり，値を代入したりすることができる．まず，*x のように，

`*` 演算子 (間接演算子) を，ポインタ型の変数x の前につけることによって，x の表すアドレスに格納されているデータ (つまり変数a の値) を参照することができる．また`*x = 0` のようにも書くことができ，その場合にはx の表すアドレス (つまり変数a) への代入となる．neg2 では，代入`*x = -*x;` によって，変数a の値を書き変えることに成功している．

8. neg2 の実行例にもあるように，ポインタ型変数の値 (アドレスの値) を%d で表示させると，-268437052 のような変な値になるが，これは計算機によっても，プログラムを実行した状態によっても違ってくる．ポインタの値を表示するには，%d ではなく，%p を用いるのがよい．

記号

C 言語では様々な記号が使われ，それがプログラムをコンパクトにしていると同時に，C を初心者にはとっつきにくいものにしている．特に* は，掛け算を表わす演算子と，ポインタ変数の宣言と，さらには間接演算子という三つの目的に使われている．言語仕様はすでに決まってしまっているので，今更どうしようもないのだが…

16 応用: 値の交換

二つの変数の値を交換することはよく行われるので，関数として用意しておくと便利かもしれない．例えば，x，y 変数の値の交換は，次のようにして行える．

```
t = x;
x = y;
y = t;
```

これを使って関数swap1 を図 38のように定義したが，前節のプログラムneg1 と同じ理由でうまくいかない．交換を行うには、図 39のようにする。

```
1  /* swap1.c   swap values of two variables */
2  /* THIS PROGRAM IS WRONG!! */
3  #include <stdio.h>

4  void swap1(int, int);

5  void main(void)
6  {
7    int a=19, b=39;

8    printf("Initially,   a = %d and b = %d.\n", a, b);
9    swap1(a, b);
10   printf("After swap1, a = %d and b = %d.\n", a, b);
11 }

12 void swap1(int x, int y)
13 {
14   int t;
15   t = x;
16   x = y;
17   y = t;
18 }
```

図 38: swap1.c

```
 1  /* swap2.c   swap values of two parameters */

 2  #include <stdio.h>

 3  void swap2(int *, int *);

 4  void main(void)
 5  {
 6    int a=19, b=39;

 7    printf("Initially,   a = %d and b = %d.\n", a, b);
 8    swap2(&a, &b);
 9    printf("After swap2, a = %d and b = %d.\n", a, b);
10  }

11  void swap2(int *x, int *y)
12  {
13    int t;
14    t = *x;
15    *x = *y;
16    *y = t;
17  }
```

図 39: swap2.c

17　変数の有効範囲

17.1　関数ブロック

次のプログラム (部分) は，図 29の 2 次方程式の根を求めるものである．

```
void main(void)
{
  int a,b,c;
    省略
  scanf("%d %d %d", &a, &b, &c);
  if (a != 0)
    quad(a, b, c);
  else
    printf("The value of an a must not be zero.\n");

}

void quad(int a, int b, int c)
{
  int det;
  float d;

  det = b*b - 4*a*c;
  省略
}
```

このように関数 quad の本体は{ }で囲まれた ブロック となっている．このブロックで宣言された変数 (この例ではdet，d) は，そのブロックの中でしか使えない (有効である，という)．つまりブロックの中だけで変数の値を参照したり，変更したりすることができる．また，関数の引数 (この例ではa，b，c) も，その関数のブロックだけで有効である．このようにある変数を使用できる範囲をその変数の 有効範囲 (scope) という．

有効範囲が異なる変数は，たとえ同じ名前をしていても，異なる変数である．例えばmain の中の変数a,b,c は，quad の引数a,b,c とは別のものである．

17.2　外部変数

関数の中で宣言された変数の有効範囲がその関数の中だけになることは既に説明した．では，二つの関数で一つの変数を使うにはどうすればよいのだろうか．例えば，二つの関数inc() とdec() を用意して，inc() は変数x の値を 1 だけ増やし，dec() は減らすようにしたい．プログラムはおおよそ次のようになるだろう．

```
void main(void){
  略

void inc(void){
  ++x ;
}
```

```
void dec(void){
  --x ;
}
```

では，この変数x はどこで宣言すればよいのだろうか? 次のように，それぞれの関数で宣言すると，

```
void main(void){
  略

void inc(void){
  int x;
  ++x ;
}
void dec(void){
  int x;
  --x ;
}
```

これらのx は，それぞれの関数を有効範囲とするので，名前は同じでも，全く別の変数となってしまう．つまり，inc() で++x によってx の値を増やしても，dec() 関数のx の値は増えないのである．

関数inc() とdec() のx を**同じ**変数にするには，これらの関数の**外**で宣言を行う．例えば，次のプログラムのようにmain 関数の前で宣言をする．

```
#include <stdio.h>
int x=0;

voind main(void){
   ...
}

void inc(void){
  ++x ;
}

void dec(void){
  --x ;
}
```

このように関数の外で宣言された変数を 外部変数 といい，その有効範囲はそのプログラム全体 (正確にはそれが宣言されたところから，そのファイルの最後まで) になり，その有効範囲内のどの関数からでも参照することができる．

これを利用したプログラムを図 40に示す．

```
 1  /* counter1.c */
 2  #include <stdio.h>
 3  void inc(void);
 4  void dec(void);
 5  void main(void);

 6  int x=0;

 7  void main(void){
 8  char c;
 9  while(1) {                      /* infinite loop */
10    scanf("%c", &c);
11    switch (c){
12      case 'i': inc(); break;
13      case 'd': dec(); break;
14      case 'q': break;
15     }
16    printf("counter = %d\n", x);
17    if (c == 'q') break;           /* exit while loop */
18    }
19  }

20  void inc(void){
21    ++x ;
22  }
23  void dec(void){
24    --x ;
25  }
```

図 40: counter1.c

18 記憶クラス

変数について理解が深まったところで，変数のもう一つ別の性質について説明しよう．それは，変数には「もの覚えのよい変数と，忘れっぽい変数」があるということである．次のプログラム片を考えてみよう．

```
void main(void) {
  int x;
  x := exp2();
  printf("%i",x);
}

int exp2(void) {
```

```
  int c = 2;
  c = c*c;
  return c;
}
```

exp2(void) は，2 に初期化された変数c を自乗した値 (4) を戻すので，このプログラムを実行するとx の値は 4 になる．では，exp2 を 2 回呼び出す次のプログラムはどうだろうか?

```
void main(void) {
  int x;
  x = exp2();
  x = exp2();
  printf("%i",x);
}
int exp2(void) {
  int c = 2;
  c=c*c;
  return c;
}
```

今度はexp2() が 2 回呼ばれている．変数c の値はexp2() が呼び出される度に自乗されるので，2 回目にexp2() が呼び出された時には 16 を戻すのだろうか? 実はそうはならない．exp2() が呼び出される度にc は 2 に初期化されるので，exp2() は 2 回目も 4 を戻す．何度exp2() を呼び出しても事情は同じで，exp2() は常に 4 を戻す．

つまり，変数c の値はexp2() によって一旦は自乗されるのに，次にexp2() が呼び出される前に，その値を忘れてしまうのである．言い方を変えると，変数c はexp2() が呼び出される度に新しく生成され，関数の実行が終わると消されてしまうのである．このような変数を 自動変数 という．

これに対して，関数の実行が終わっても，その変数を消さずに，その値を覚えておくようにすることができる．そのためには変数宣言の頭に `static` というキーワードを付ける．次のプログラムを見てみよう．

```
void main(void) {
  int x;
  x = exp2();
  x = exp2();
  printf("%i",x);
}
int exp2(void) {
  static int c = 2;
  c=c*c;
  return c;
}
```

このようにすると，変数c の値は関数exp2() の実行が終わっても保存される．例えばこの例では 2 度目にexp2() が呼び出された時のc の値は，前に呼び出された時の最後の値が残っているので，4 であり，その結果 2 度目のexp2() は 16 を返す．(この場合c = 2 という初期化は行われないことに注意) このように値を覚えている変数を 静的変数 という．

図 41はこの静的変数を使ったカウンタープログラムの例である．c を入力する度に値が 1 ずつ増える．

```
1  /* counter2.c */
2  #include <stdio.h>
3  int count(void);
4  void main(void);

5  void main(void){
6  char c;
7  while(1) {                      /* infinite loop */
8    scanf("%c", &c);
9    switch (c){
10     case 'c': printf("%d\n",count()); break;
11     case 'q': break;
12    }
13   if (c == 'q') break;          /* exit while loop */
14   }
15 }

16 int count(void){
17   static int c=0;
18   return ++c;
19 }
```

図 41: counter2.c

自動変数と静的変数の違いは，変数がメモリに保持される期間の違いと考えることができる．これを保持期間といい，有効範囲とともに変数のもう一つの性質である．自動変数や仮引数として宣言された変数は，その関数の実行の開始から終了までを保持期間とし，また静的変数や前述の外部変数は，プログラムの実行の開始から終了までを保持期間とする．

18.1 記憶クラスのまとめ

有効範囲と保持期間の二つのによって変数は分類することができる．これを記憶クラスといい，自動記憶クラス，静的記憶クラス，外部記憶クラスなどがある．

1. 自動記憶クラス
   関数 (ブロック) の内部で定義された変数は，特に指定がなければ自動記憶クラスとなる．これを自動変数と言う．仮引数もこのクラス．
   有効範囲: それが宣言された関数だけ．保持期間: 関数の呼び出しから 終了まで (終了後，消滅)．
   自動変数は，ブロックの先頭で定義することができる．その場合，有効範囲はそのブロック内で，保持期間はそのブロックの実行中．

2. 静的記憶クラス
   有効範囲は自動変数と同じだが，保持期間は，その関数の実行が終了しても消滅せず，次の関数呼び出しにも，その値が生き残っているもの．これを静的変数と言う．静的変数は，`static` をつけて，`static int x;` のように宣言する (初期値は最初だけ有効).
   例) `int counter(){static int c=0; return c++;}` は`count()` が呼び出される度に，$0, 1, 2, 3, 4$ という値を返す.

3. 外部記憶クラス
   関数の外側で定義された変数は，外部記憶クラスに属する．これを外部変数という．
   有効範囲: 全ての関数．保持期間: プログラムの実行中.

19　数値計算の誤差

19.1　機械エプシロン

浮動小数点型 (float，double) の値は，計算機内では，$0.1234567\cdots \times 10^{12}$ のように仮数部 (0.1234567) と指数部 (10^{12}) に分けて表現する．それらは，2 進数で表現される．しかし，それぞれにある決まった大きさの記憶領域が割り当てられるので，仮数部も指数部も，ある範囲の数値しか表すことができない．特に，仮数部は実数であるのに，有限の桁数しか表現できないので，無限少数は原理的に表現できない．

例えば，仮数部が表現できる桁数が 10 桁しかないと仮定すると，1.0×10^1 に 1.0×10^{-10} を加えると，1.0000000001×10^1 となり，仮数の桁数が 11 桁を越え，11 桁以上の部分は捨てられるので，加えた結果は 1.0 となってしまう．これを**情報落ち誤差**と呼ぶ (後述)．

一般に，1.0 に，ある小さい ϵ を加えた時に，$1.0 = (1.0 + \epsilon)$ となる最大の ϵ を機械エプシロンという．図 42のプログラムは，double 型に対する機械エプシロンを求める求めるもので，機械エプシロンを表すe の値を 1.0 から，1/2 倍ずつ小さくして行き (2 進数だから)，1.0 == (1.0+e) が成立するe を求めている．なお，while (!(1.0 + e == 1.0)); の! は否定を表し，また printf("Size of e = %d \n", sizeof e); は，変数e に割り当てられた記憶量 (sizeof e) を表示させている．

図 43の結果からも分かるように，double 型は，8 バイトの記憶量をもち，その機械エプシロンは，10^{-16} のオーダーである．

一方，float 型については，図 44と 45に示す．float 型は，4 バイトの記憶量を持ち，その機械エプシロンは，10^{-7} のオーダーである．つまり，float 型は約 8 桁程度の精度を持ち，またdouble 型は 17 桁程度の精度を持つことがわかる．

プログラムを作成する場合には，数値計算が必要とする精度に応じてfloat 型とdouble 型を使い分ける必要がある．理想的にはdouble 型の方が良いのであるが，データ数が大きい場合のデータ量の問題や，計算時間の問題が生じる．逆に，次節以降で述べる他の数値誤差の問題もあるので，double 型を使ったからと言って，精度の良い計算が必ずしもできるわけではない．

なお，図 44で，ところどころに(float) とキャストが付いているのは，これを付けないと計算がdouble で行なわれるからである．

19.2　丸め誤差

浮動少数点は，有限の桁数しか仮数部を表現できないので，当然，無限小数を表すことはできない．無限小数は，仮数部の桁内に入る最後の桁が丸められて表現される．実際の丸めは 2 進数で行なわれるが，考え方としては，10 進数の四捨五入に似ている．この丸めによって数値誤差が生じる．

例えば，1/20 や 1/30 は，2 進数表現では無限小数となる．図 46に示すプログラムで，それを実際に確かめて見ると，図 47のように誤差が生じているのが分かる．

```
/* deps.c Machine Epsilon */
#include <stdio.h>

void main(void)
{
  double e=1.0, e0;

  printf("Size of e = %d \n", sizeof e);

  do {
    e0 = e;
    e = e/2.0;
    printf("%.40e\n", e);
  } while (!(1.0 + e == 1.0));
  e = 1.0 + e0;
  printf("Machine Eps = %.40e\n", e-1.0);
}
```

図 42: deps.c

```
> gcc deps.c
> a.out
Size of e = 8
5.0000000000000000000000000000000000000000e-01
2.5000000000000000000000000000000000000000e-01
1.2500000000000000000000000000000000000000e-01
6.2500000000000000000000000000000000000000e-02
3.1250000000000000000000000000000000000000e-02
1.5625000000000000000000000000000000000000e-02
7.8125000000000000000000000000000000000000e-03
3.9062500000000000000000000000000000000000e-03
                    中略
8.8817841970012523233890533447265625000000e-16
4.4408920985006261616945266723632812500000e-16
2.2204460492503130808472633361816406250000e-16
1.1102230246251565404236316680908203125000e-16
Machine Eps = 2.2204460492503130808472633361816406250000e-16
>
```

図 43: 実行例

```
/* feps.c Machine Epsilon */
#include <stdio.h>

void main(void)
{
  float e=1.0, e0;

  printf("Size of e = %d \n", sizeof e);

  do {
    e0 = e;
    e = e/(float)2.0;
    printf("%.40e\n", e);
  } while (!((float)1.0 + e == (float)1.0));
  e = (float)1.0 + e0;
  printf("Machine Eps = %.40e\n", e-1.0);
}
```

図 44: feps.c

```
> gcc feps.c
> a.out
Size of e = 4
5.0000000000000000000000000000000000000000e-01
2.5000000000000000000000000000000000000000e-01
1.2500000000000000000000000000000000000000e-01
6.2500000000000000000000000000000000000000e-02
3.1250000000000000000000000000000000000000e-02
1.5625000000000000000000000000000000000000e-02
7.8125000000000000000000000000000000000000e-03
3.9062500000000000000000000000000000000000e-03
                    中略
4.7683715820312500000000000000000000000000e-07
2.3841857910156250000000000000000000000000e-07
1.1920928955078125000000000000000000000000e-07
5.9604644775390625000000000000000000000000e-08
Machine Eps = 1.1920928955078125000000000000000000000000e-07
>
```

図 45: 実行例

```
/* round.c  rounding error */
#include <stdio.h>

void main(void){
  double x;

  x = 1.0/2.0;
  printf("x = %.40e\n", x);

  x = 1.0/20.0;
  printf("x = %.40e\n", x);
}
```

図 46: round.c

```
> gcc round.c
> a.out
x = 5.0000000000000000000000000000000000000000e-01
x = 5.0000000000000002775557561562891351059079e-02
>
```

図 47: 実行例

19.3 桁落ち誤差

値がほぼ等しい 2 つの数の差を作ると，計算結果において絶対値が小さくなった分だけ相対誤差が大きくなってしまうことを桁落ち誤差という．例えば，$1.0016 - 1.0 = 0.0016$ では，有効桁数が 2 桁に減っている．

もう少し複雑な例では，$x = 0.0031384$ に対し，式

$$1 - \frac{1}{\sqrt{1+x}} = \frac{\sqrt{1+x}-1}{\sqrt{1+x}}$$

の右辺と左辺の値を計算すると，真の値 0.0015879 に対して
左辺 $= 1 - 1/\sqrt{1.0032} = 1 - 1/1.0016 = 1 - 0.99840 = 0.0016000$
右辺 $= (1.0016 - 1)/1.0016 = 0.0016/1.0016 = 0.0015974$
となり，両辺ともに計算結果の有効数字は 2 桁程度しかない．

桁落ちを防ぐには，桁落ちを起こす引算がないように式を変形する．この場合，右辺は次のように引き算の無い形に変形でき，その結果桁落ち誤差がなくなる．

$$\text{右辺} = \frac{x}{\sqrt{1+x}(\sqrt{1+x}+1)} = 0.0031834/(1.0032 + 1.0016) = 0.0015879$$

19.4 情報落ち誤差

絶対値の大きく異なる 2 数の加減算を行なうと，小さい方の値の大きい方の有効桁数内に入る部分が少なくなるために起こる誤差．

例) $X = 0.48362 \times 10^5$ と $Y = 0.51323 \times 10^2$ の和 Z は，有効桁数を 5 桁とすると，

$$\begin{aligned} Z &= 0.48362 \times 10^5 + 0.51323 \times 10^2 \rightarrow 0.48413323 \times 10^5 \\ &\rightarrow 0.48413 \times 10^5 + 0.323(\text{誤差}) \\ &= 0.48413 \times 10^5 \end{aligned}$$

例) $X = 0.48362 \times 10^5$ と $Y = 0.51323$ の和 Z は，

$$\begin{aligned} Z &= 0.48362 \times 10^5 + 0.51323 \rightarrow 0.4836251323 \times 10^5 \\ &\rightarrow 0.48362 \times 10^5 + 0.51323(\text{誤差}) \\ &= 0.48362 \times 10^5 \\ &= X(\text{Z の値には Y が加えられなかった !}) \end{aligned}$$

情報落ち誤差によって思いもよらない問題が起きることがある．図 48のプログラムは，`x` の値を少しずつ増やしながら計算をするという，よくあるプログラムであるが，`x=1.0` に機械エプシロンより小さい値を繰り返し加えているために`x` の値が増えず，その結果無限ループになる．(プログラムを止めるには，`CTRL-C` を入力する．)

19.5 条件判定

以上見てきたように，浮動小数点の計算には多くの誤差が伴うことを常に考慮しなければならない．したがって，例えば`if` 文で浮動小数点の値`x` の値が 0 かどうかを判定するのに，

```
if( x == 0.0 )
```

のようにしても意味がないことは理解できよう．理論的には 0 になるはずでも，計算過程で様々

```
 1  /* ierr.c  Infinte Loop */
 2  #include <stdio.h>

 3  void main(void)
 4  {
 5    float x=1.0;
 6    while (x <= 2.0) {
 7      printf("%.12e\n", x);
 8      x += 1.0e-8;
 9    }
10  }
```

図 48: ierr.c

な誤差が含まれてしまい，ぴったりと 0 にはならない場合のほうが普通だからである．このような場合，閾値を設定し，x の絶対値がその値よりも小さければ 0 と判定する．つまり，

```
if( fabx(x) < 0.0001 )
```

のようにする．この閾値の値をどの程度にするかは，x の含みうる**誤差の見積り**を行なわなければならず，数値計算のプログラムを作成する上で重要な課題となる．

19.6 打ち切り誤差: 級数の例

e^x を $x = 0$ で Taylor 展開すると，次のようになる．

$$e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots \frac{x^{k-1}}{(k-1)!} + \frac{x^k}{k!} + \cdots$$

この式を使って，e^x を求めるプログラムを考える．これは無限級数であるので，計算を途中の項までで打ち切る必要がある．第 k 項の絶対値は，k が大きくなるにつれて，0 に近付くので，十分大きな k 項まで計算すれば，級数の値はよい近似を与えるはずである．

そこで，第 k 項までの和を S_k で表し，次のような打ち切り条件を用いる．

$$\frac{|S_k - S_{k-1}|}{|S_{k-1}|} < \epsilon$$

このように，計算を途中で打ち切ることによって生ずる誤差を一般に**打ち切り誤差**と呼ぶ．プログラム例と実行例を図 49と 50に示す．計算結果を見てみると，$\exp(1)$ や $\exp(10)$，$\exp(100)$ などの値は大変精度よく求まっている．一方，マイナスの値に対する $\exp(-20)$ や $\exp(-40)$ の値は大きくずれており，しかも符合まで異なっている．

これは，上述の桁落ち誤差によるものである．この級数の一般項 $\frac{x^k}{k!}$ の分子は，x が負の場合，k が偶数の時には正，奇数の時には負となる．図 51に $x = -40$ の時の各項の様子を示す．$k = 40$ の前後の，項の絶対値が急激に増加する区間においては，その項の値が級数のなかでも支配的になっており，その結果，ほぼ絶対値の等しい正の数と負の数を交互に加算することになる．そのため，加算において桁落ち誤差が発生し，それが累積することによって大きな誤差を生じている．また，x が正の時に比べ収束が遅いために，加算の回数も増えている．

```
 1  /* exp.c -- exponential function */
 2  #include <stdio.h>
 3  #include <math.h>
 4  #define EPS 1.0e-16

 5  double myexp(double x);

 6  void main(void)
 7  {
 8    double x;

 9    printf("Enter a real number.\n");
10    printf("Enter q to quit.\n");
11    while(scanf("%lf", &x) == 1)
12      printf("Exp of %e is %.15e (%.15e)\n", x, myexp(x), exp(x));
13  }

14  double myexp(double x)
15  {
16    double a=1.0, s=1.0, s0;
17    int n=1;

18    do {
19      s0 = s;
20      a = a * x / (double) n;
21      s = s + a;
22      n++;
23    }  while(fabs(s0 - s) > EPS);
24    return s;
25  }
```

図 49: exp.c

このように x が負の場合には，次のような工夫を行なう．

- $\exp(-20)$ や $\exp(-40)$ のように， x が負の整数の場合は， $e = 2.718281828459046$ を $|x|$ 回乗じて，その逆数をとる．
- $\exp(-20.12)$ のように x が負の実数の場合は， $\exp(-20.12) = \exp(-20) \times \exp(-0.12)$ のように x の整数部分と小数部分に分けて，桁落ちの心配のない小数部分は級数によって求め，整数部分は上記の方法によって求め，それらを掛け合わせる．

```
> gcc exp.c -lm
> a.out
Enter a real number.
Enter q to quit.
1
Exp of 1.000000e+00 is 2.718281828459046e+00 (2.718281828459046e+00)
10
Exp of 1.000000e+01 is 2.202646579480671e+04 (2.202646579480672e+04)
100
Exp of 1.000000e+02 is 2.688117141816134e+43 (2.688117141816136e+43)
-1
Exp of -1.000000e+00 is 3.678794411714424e-01 (3.678794411714423e-01)
-10
Exp of -1.000000e+01 is 4.539992962302762e-05 (4.539992976248485e-05)
-20
Exp of -2.000000e+01 is 5.621884467407823e-09 (2.061153622438558e-09)
-40
Exp of -4.000000e+01 is -3.165731894063124e+00 (4.248354255291589e-18)
q
>
```

図 50: 実行例

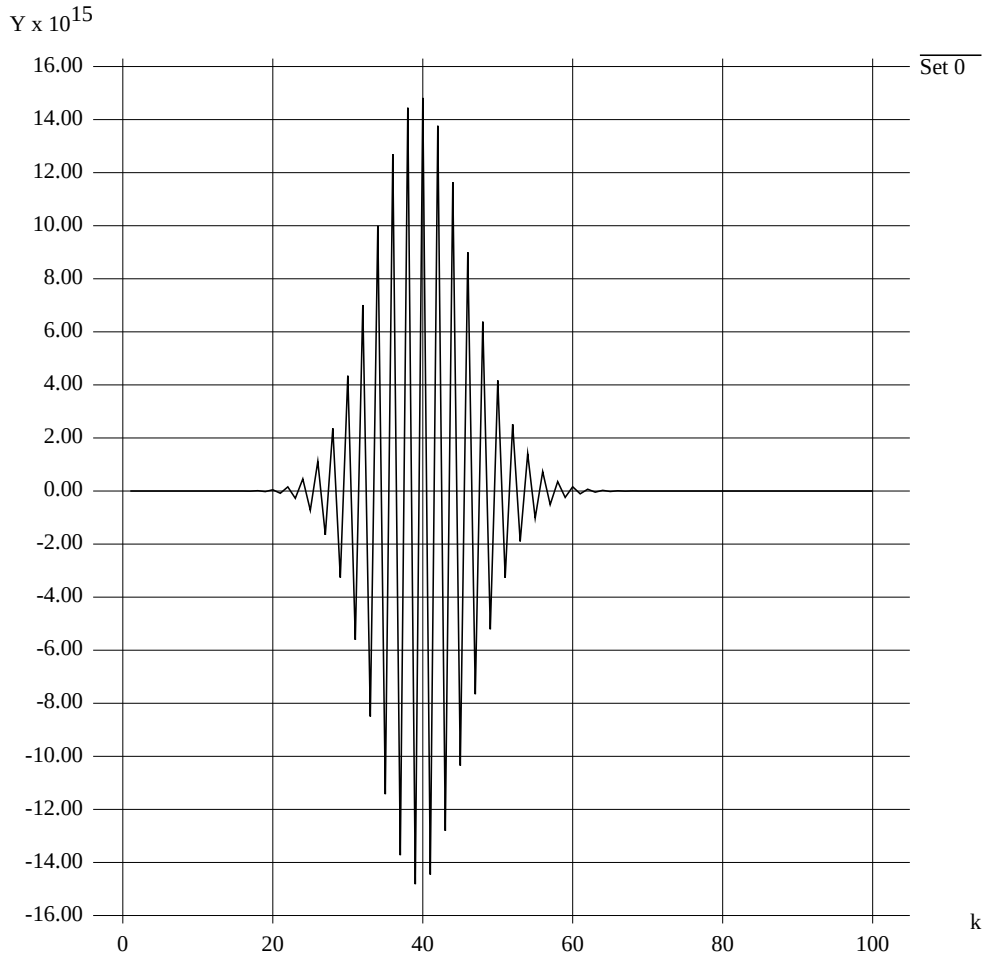


図 51: 級数の各項 ($x = -40$ の場合

20 配列

他の多くの言語と同じ様にC言語でも配列を使える．まず1次元配列の基本的な例題を紹介しよう．

```
 1  /* array.c -- small example for array */
 2  #include <stdio.h>
 3  #define MAX 100

 4  void main(void)
 5  {
 6    int a[MAX];
 7    int sum = 0;
 8    int n = 0;
 9    int i;

10    printf("Input whole numbers up to %d.\n", MAX);
11    printf("Enter EOF to finish the input.\n");
12    while( n < MAX && scanf("%d",&a[n])!=EOF) ++n;
13    for (i = 0; i < n; i++)
14      sum += a[i];
15    printf("The sum of the input data is %d.\n", sum);
16  }
```

図 52: array.c

```
> gcc array.c
> a.out
Input whole numbers up to 100.
Enter EOF to finish the input.
1 2 3 4
^D                                  EOF(End Of File) CTRL-D の入力
The sum of the input data is 10.
```

図 53: 実行例

1. `int a[MAX];` 配列の宣言
   これは，整数型の要素をMAX個だけもつ配列を宣言する．MAXは記号定数(100)である．これはint a[100];でもよい．整数型の変数の宣言と異なるのは，変数名の後に[MAX]のように配列のサイズを付けることだけである．

2. 配列のi 番目の要素は，a[i] で表す．注意が必要な点は，添え字が 0 から始まる約束なので，要素は，
   a[0], a[1], ... , a[MAX-1]
   となる． Pascal では任意の整数の範囲を添え字にすることができたが， C では 0 から始まるように固定されている．

3. `while( n < MAX && scanf("%d",&a[n])!=EOF) ++n;`
   — 配列の要素にデータを読み込むには，n の値を 0 から 1 ずつ増やしながら，第n 番目の要素にscanf でデータを読み込んでゆく．繰り返し条件は，読み込んだデータ数が配列の最大要素数MAX を越えず，かつ (&& は論理積) 読み込んだデータがEOF でないことである．++n を繰り返すことによって，a[0], a[1], a[2],... の順でデータを読み込む．

4. `for (i = 0; i < n; i++) sum += a[i];`
   さらに総和を求めるには，i の値を 0 からn まで変えながら， sum += a[i] を繰り返す．この文は sum = sum+a[i] のことであり，これによってsum という変数に，和が求まる．ループ変数のi が配列要素の添え字になっており，for 文の初期化部 (i=0) で 0 に初期化され，更新部i++ で， 1 ずつ増える．

21 応用: ソーティング

21.1 直接挿入法

配列の例題として，その要素を大きい順や小さい順に並べかえるソーティングを紹介する．これは，アルゴリズムとして最もよく研究されたものと言えよう．ここでは最も簡単なアルゴリズムを紹介するが，興味のある人はアルゴリズムの教科書を参考にして勉強して欲しい．

1. アルゴリズム
   データ列 $a_1, a_2, a_3, \ldots a_N$ を大きい順に (降順．小さい順のことを昇順という) 並べ直す．そのために， i の値を $2, 3, \ldots, N$ と変えながら，それぞれの値に対して，次のことを行なう．すなわち， a_i より前にあるデータ $(a_1, a_2, a_3, \ldots a_{i-1})$ の中で， a_i 以上の要素があったら，その前に a_i を挿入する．ここで予め a_0 には a_i を入れておき， $(a_1, a_2, a_3, \ldots, a_{i-1})$ に a_i 以上の要素がない場合には (a_i が最大の時)， a_i は a_1 に挿入される． a_0 のような役割を果たすものを番兵という．

番兵 a_0	a_1	a_2	a_3	a_4	a_5	a_6	a_7
87	23	[87]	19	30	11	89	87
19	87	23	[19]	30	11	89	87
30	87	23	19	[30]	11	89	87
11	87	30	23	19	[11]	89	87
89	87	30	23	19	11	[89]	87
87	89	87	30	23	19	11	[87]
	89	87	87	30	23	19	11

   $i = 2, 3, \ldots, N$ と処理を繰り返してゆくので，ある i について処理を行なう時には，$(a_1, a_2, a_3, \ldots a_{i-1})$ は既にソーティングされている (降順にならんでいる) ことに注意する．

2. 上の図から， a_i についての処理を行なう場合に， a_i よりも前の要素で， a_i よりも値が小さいものは， a_i の挿入によって，配列上の位置が一つだけ後ろに下がることがわかる．今 $x = a_i$ とすると， a_i の前の要素 $a_j (j = i-1, i-2, i-3, \ldots, 1)$ のうち， x よりも値が小さいものについては，一つだけ後ろにずれるので， $a_{j+1} \Leftarrow a_j$ とする．そして， $a_j \geq x$ なる j に対して， a_j の後ろに x を挿入，すなわち $a_{j+1} \Leftarrow x$ とする．配列をa とすると，この部分のプログラムは，以下のようになる．

```
for (i = 2; i <= n; i++){
  x = a[i];
  a[0] = a[i];
  j = i-1;
  while(x > a[j]){
    a[j+1] = a[j];
    j--;
    }
  a[j+1]=x;
}
```

21.2 配列の引数

直接挿入法によるソートのアルゴリズムが分かったところで，そのプログラムを作成してみよう．ここではソートを行う部分や配列にデータを読み込む部分を関数として定義し，それらに配列を引数で渡すことにする．一般の変数と同様に配列を引数とすることができる．

1. 関数readArray
   これは，配列にデータを読み込み，その個数を戻す．関数の引数としては，配列名と配列のサイズを渡す．

2. `int readArray(int a[], const int limit);`
   この関数のプロトタイプ宣言．配列を受けとる仮引数の宣言`int a[]` では，`a[]` によって，仮引数`a` が配列であることを示し，`int` によって配列の要素の型を宣言している．配列のサイズは書かいても書かなくてもよいが，仮引数の配列のサイズは，それに対する実引数の配列のサイズで決まってしまうので，ここにサイズを書いたとしても意味がない．

   そこで，配列の具体的なサイズは，別の変数`limit` によって関数に渡す必要がある．この変数は，配列のサイズを表しているので，その値が変わることはない．つまり定数であるので，`const` 修飾子をつけている．こうすると，この関数の中で`limit` の値を変えることができない．

3. 関数`readArray` は，配列にデータを読み込み，そのデータ数を返す．`a[0]` は番兵として使うので，`a[1]` 以降に読み込む．

4. 関数`printArray` は配列の内容を表示する． `if( i%10 == 0) putchar('\n');` によって要素を 10 個表示するごとに改行する．また，データ数がちょうど 10 の倍数でない時には，全てのデータを表示した後にも改行する必要がある．そこで，データの数を N とすると，全てを表示し終った後で，`i` の値は $N+1$ になっているので， `if( i%10 != 1) putchar('\n');` によって，データ数がちょうど 10 の倍数でない時に改行を行なう．

5. 関数`sort` は，直接挿入方によって大きい順に (降順) ソートする．

21.3 配列の初期値の設定

1. 変数と同じ様に配列にも記憶クラスがあり，それぞれ自動配列，外部配列，静的配列がある．この内，外部配列と静的配列には初期値を設定できる．
2. 初期値はコンマで区切って並べ，全体を`{}` でくくる．
   例) `int a[6]={0, 2, 4, 6, 8, 10}`
   要素の数よりも初期値の数が多くてはいけない．少ない時は，`a[0]` から順次設定され，残りには 0 が補われる．
3. 配列のサイズを自動的に初期値の個数にすることもできる．
   例) `int a[]={0, 2, 4, 6, 8, 10}` .
4. プログラムの中で配列のサイズを知るには，配列の記憶領域の大きさをその 1 要素当たりの記憶領域の大きさで割ればよい．例えば，`a` のサイズを知るには， `sizeof` 演算子を使って， `sizeof a / sizeof(int)` で得られる．

   `sizeof` 演算子は，被演算子が配列名の時は，配列が占める領域のバイト数となる．また，`int` のような型指定子の時は，`sizeof (int)` のように() でくくる．

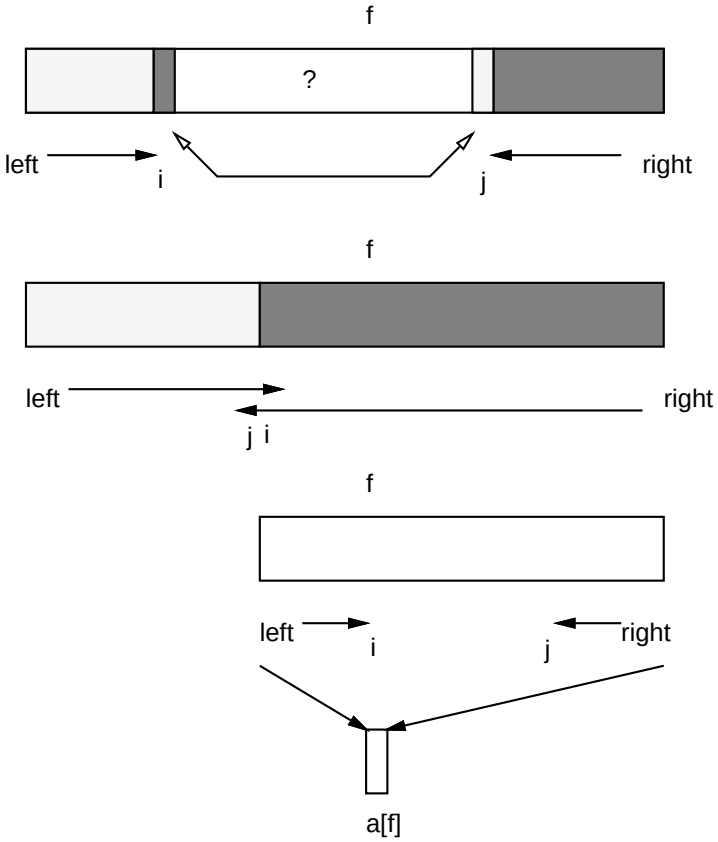
```
 1  /* sort.c -- straight insertion sort */
 2  #include <stdio.h>
 3  #define MAX 100
 4  int readArray(int a[], const int n);
 5  void printArray(int a[], int n);
 6  void sort(int a[], int n);
 7  void main(void)
 8  {
 9    int n, a[MAX+1];
10    n = readArray(a, MAX);
11    sort(a, n);
12    printArray(a, n);
13  }
14  int readArray(int a[], const int limit)
15  {
16    int n = 1;
17    printf("Input whole numbers up to %d.\n", limit);
18    printf("Enter EOF to finish the input.\n");
19    while(n < limit+1 && scanf("%d",&a[n])!=EOF) ++n;
20    return --n;
21  }
22  void printArray(int a[], int n)
23  {
24    int i;
25    for(i=1; i<=n; i++){
26      printf("%d ", a[i]);
27      if( i%10 == 0) putchar('\n');
28    }
29    if( i%10 != 1) putchar('\n');
30  }
31  void sort(int a[], int n)
32  {
33    int i,j,x;
34    for (i = 2; i <= n; i++){
35      x = a[i];
36      a[0] = a[i];
37      j = i-1;
38      while(x > a[j]){
39        a[j+1] = a[j];
40        j--;
41      }
42      a[j+1]=x;
43    }
44  }
```

図 54: sort.c

22 応用: 中央値の探索

データ列 $a_i, (i = 1, 2, \ldots, n)$ の中央値 ($(n+1)/2$ 番目に大きいデータ) を見つけるプログラムを考える．データ列は，配列a[i] に入っているものとする．まず，中央値ではなく，より一般にa[1] からa[n] までのn 個のデータの中からf 番目に大きいデータを見つける関数を用意し，これを使ってf = (n+1)/2 番目のデータを求めればこれが中央値となる．

f 番目に大きな要素を求めるための基本的な考え方は，配列の左の方にある要素と右の方にある要素を適当に交換して，a[f] の左側 (a[1] からa[f-1]) にはa[f] よりも小さいものが来るようにし，またa[f] の右側 (a[f+1] からa[n]) にはそれよりも大きいものがくるようにすることである．



1. x = a[f] とする．
2. 配列を左と右の両方向からサーチし，それぞれx 以上のデータとx 以下のデータを探す．この時，探す添え字の範囲を[left，right] で表わす．最初は，[1,n] である．
3. 左と右から配列を一つずつ調べてゆき，前述のようにそれぞれx 以上のデータとx 以下のデータを探す．
4. 両方見つかった所で，それぞれの位置をi，j とし，a[i] とa[j] を交換する．
5. これを繰り返してゆくと，ある時点で，i，j が交差する．すなわちi > j となる．この時点で分割が終了し，交差した所 (i 番目 (あるいはj) 番目，これを分割点という) の右側にはx 以上の値のデータが並んでおり，左側にはx 以下の値のデータが並んでいる．つまり，x は，前からi 番目 (あるいはj) 番目に大きいデータであることがわかる．
6. もしf が分割点よりも右側ならば (f > j)，f 番目の値はi と，right の間にある．また，左側ならば (f < i)，f 番目の値はj と，left の間にある．そこで，それぞれの区間に対

して，上記の手続きをもう一度行う．つまり，前者の場合にはleft = i として，後者の場合にはright = j として，2 へ行く．

7. これを繰り返していくと，サーチする区間 [left, right] がだんだん小さくなっていき，ある時点で f の所で交差する (left >= right のとき)．この状態で，交差したところの左には，f 番目に大きいデータよりも小さいものが並んでおり，右にはそれよりも大きいデータが並んでいる．したがって，交差したところの配列の要素a[f] が求めるデータになる．

図 55にプログラムを示す．

1. プロトタイプ宣言では，引数名を省略できる．配列やポインタの場合は，次のように書く．
   `int readArray(int [], int);` または， `void swap(int *, int *);`

マクロとその展開

プログラムを高速化する簡単な方法に，呼び出し回数が多く，かつ，比較的短い関数をマクロ化するという方法がある．例えば，図 55のswap.

1. マクロ
   swap をより高速に行なうには，それをマクロで書き直す．マクロを定義するには，記号定数の定義でも使った#define を使う．例えば#define MAX 100 のような記号定数の定義では単にMAX という文字列が100 という文字列に置き換えられたが，定義のなかに変数を含むことができる．この場合には，#define MAX 100 の所に，次の定義を追加する．
   `#define SWAP( x,y ) {int t; t = x; x = y; y = t;}`
   ここで，(x,y) は関数の引き数のようなパラメータであり，例えばプログラム中で，
   `SWAP(a[i], a[j]);` などの表現があると，それを，a[i] をx に対応させ，かつa[j] をy に対応させて，さらに{int t; t = x; x = y; y = t;} に従って
   {int t; t = a[i]; a[i] = a[j]; a[j] = t;} に置き換える．このことを，マクロを展開する，という．

   プログラム中の手続き呼び出し `swap(&a[i], &a[j]);` を `SWAP(a[i], a[j]);` に書き変えることによって， `SWAP(a[i], a[j])` は，
   すべて {int t; t = a[i]; a[i] = a[j]; a[j] = a[i];} にマクロ展開される．その結果，関数の呼び出しが不要となり，計算の高速化が期待される．図 56に，図 55のswap をマクロ化したプログラムを示す．

   なお， {int t; t = a[i]; a[i] = a[j]; a[j] = a[i];} のint t; のように，自動変数は適当な所 (ブロックの先頭など) で宣言してよい．

2. マクロ使用上に当たって注意しなければならないことがある．例えば条件演算子? を用いて絶対値のマクロを定義する．
   `#define ABS( x ) (x > 0) ? x : -x` これは `y = ABS(a);` などでは，
   `y = (a > 0) ? a : -a;` のようにうまく展開されるが， `y = ABS(a+b);` は，
   `y = (a+b > 0) ? a+b : -a+b;` となり (a+b が負の場合に-a+b を返してしまい具合が悪い．そこで， マクロの引数は括弧で括っておく のがよい．
   `#define ABS( x ) ((x) > 0) ? (x) : -(x)`
   こうすると， `y = ABS(a+b);` は，`y = ((a+b) > 0) ? (a+b) : -(a+b);` となる．

3. 補足: マクロの定義が複数行に渡る時は，途中の行末に\ をつける．

```
 1  /* median1.c -- find */
 2  #include <stdio.h>
 3  #define MAX 100
 4  int readArray(int [], int);
 5  void swap(int *, int *);
 6  int find(int [], int, int);

 7  void main(void)
 8  {
 9    int n, a[MAX];
10    n = readArray(a, MAX);
11    printf("The median is %d.\n", find(a, n, (n+1)/2));
12  }

13  int readArray(int a[], const int limit)
14  {
15    int n = 1;
16    printf("Input whole numbers up to %d.\n", limit);
17    printf("Enter EOF to finish the input.\n");
18    while(n < limit+1 && scanf("%d",&a[n])!=EOF) ++n;
19    return --n;
20  }

21  void swap(int *x, int *y)
22  {
23    int t;
24    t = *x; *x = *y; *y = t;
25  }

26  int find(int a[], int n, int f)
27  {
28    int left, right, i, j;
29    int x;

30    left=1;  right=n;
31    while(left < right){
32      x = a[f];
33      i = left;  j=right;
34      while( i <= j ){
35        while( a[i] < x ) i++;  while( x < a[j] ) j--;
36        if(i <= j){
37          swap(&a[i], &a[j]);
38          i++;  j--;
39        }
40      }
41      if( j < f ) left = i;   if( f < i ) right = j;
42    }
43    return a[f];
44  }
```

図 55: median1.c

```
1  /* median2.c -- find */
2  #include <stdio.h>
3  #define MAX 100
4  #define SWAP(x, y) {int t; t = x; x = y; y = t;}

5  int readArray(int [], int);
6  int find(int [], int, int);

7  void main(void)
8  {
9    int n, a[MAX];
10   n = readArray(a, MAX);
11   printf("The median is %d.\n", find(a, n, (n+1)/2));
12 }

13 int readArray(int a[], const int limit)
14 {
15   int n = 1;
16   printf("Input whole numbers up to %d.\n", limit);
17   printf("Enter EOF to finish the input.\n");
18   while(n < limit+1 && scanf("%d",&a[n])!=EOF) ++n;
19   return --n;
20 }

21 int find(int a[], int n, int f)
22 {
23   int left, right, i, j;
24   int x;

25   left=1;  right=n;
26   while(left < right){
27     x = a[f];
28     i = left;  j=right;
29     while( i <= j ){
30       while( a[i] < x ) i++;  while( x < a[j] ) j--;
31       if(i <= j){
32         SWAP(a[i], a[j]);
33         i++;  j--;
34       }
35     }
36     if( j < f ) left = i;   if( f < i ) right = j;
37   }
38   return a[f];
39 }
```

図 56: median2.c

23 応用: 多項式の計算

23.1 Horner 法

多項式 $y = 5x^3 + 4x^2 - 7x + 1$ の計算する代入文は，次の二通り考えられる．

```
y = 5*x*x*x + 4*x*x - 7*x + 1;      (乗算 6 回 加算 4 回)
y = ((5*x + 4)*x - 7)*x +1          (乗算 3 回 加算 3 回)
```

一般に，計算回数を比べてみると，
$y = a_n x^n + a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + \cdots a_1 x^1 + a_0$　N(N+1)/2 回の乗算と N 回の加算
$y = (\cdots((a_n x + a_{n-1})x + a_{n-2})x \cdots a_1)x) + a_0$　N 回の乗算と N 回の加算
となり，後者の方が効率がよい．後者の計算方法を Horner 法という．また， Horner 法は桁落ちなどにも強いことが分かるだろう．

Horner 法で多項式を計算するには，係数 a_i を配列a の要素a[i] で表し，上式の括弧の中に相当する項をs で表すと，

```
s := a[n]
s := s・x + a[n-1]
s := s・x + a[n-2]
- - - - - - - - - -
s := s・x + a[2]
s := s・x + a[1]
s := s・x + a[0]
```

つまり，s=a[n] と初期化しておき，s=s*x+a[i] をi=n-1 からi=0 まで繰り返し計算すればよい．

```
s = a[n];
for(i=n-1; i >= 0; i--)  s = s*x + a[i];
```

23.2 Horner 法によるプログラム

図 57は， Horner 法によるプログラムで，変数x のn 次多項式 (係数a[i]) を計算する関数 double polynom(a[], n, x) を示している．

24 応用:Newton Raphson 法

高次の代数方程式や超越方程式の解を数値的に求める方法として，解の適当な推定値から出発して繰返し計算によって正解に収束させる反復法がある．その一つとして， Newton-Raphson 法がある．

24.1 理論

$f(x) = 0$ の解 x の近似値を x_k として，その近くで $f(x)$ を Taylor 級数展開の 1 次の項までで近似する．つまり， $f(x)$ を， x_k の近くで x_k における接線によって近似する．従って $f(x) = 0$ は,

$$f(x) \approx f(x_k) + f'(x_k)(x - x_k) = 0$$

```
 1  /* polynom.c -- polynomial */
 2  #include <stdio.h>
 3  #define N 50

 4  double polynom(double a[], const int n, double x);

 5  void main(void)
 6  {
 7    double a[N], x;
 8    int n,i;

 9    printf("Enter the degree of a polynomial: ");
10    scanf("%d", &n);
11    printf("Enter coefficients:\n");
12    for(i=n;i >= 0; i--) {
13      printf("a[%d]=",i);
14      scanf("%lf",&a[i]);
15    }
16    printf("Enter values for polynomial.\n");
17    printf("To stop, enter EOF.\n");
18    while(scanf("%lf", &x) != EOF)
19      printf("Polynomial = %e\n", polynom(a,n,x));
20  }

21  double polynom(double a[], const int n, double x)
22  {
23    double s;
24    int i;
25    s = a[n];
26    for (i = n-1; i >= 0; i--) s = s*x + a[i];
27    return s;
28  }
```

図 57: polynom.c

この方程式を解いて，解 x のより良い近似 x_{k+1} が得られる．

$$x_{k+1} = x_k - f(x_k)/f'(x_k)$$

このようにして，適当な初期値 x_0 から出発し，上式を用いて $x_0 \to x_1 \to x_2,,,$ と繰返し計算を進め，収束させる．

収束性については，以下のような定理が証明できる．

定理　逐次近似列 $x_0, x_1, x_2, \cdots$ 及び解 α を含む区間で，$F(x) = x - f(x)/f'(x)$ の導関数 $F'(x)$ の最大値を K とする．$K < 1$ ならば逐次近似列 $x_0, x_1, x_2, \cdots$ は α に収束する．

24.2　収束判定

実際のプログラムでは，収束計算を無限回繰返すこともできないし，また数値計算の誤差によってむやみに計算を繰返しても真の解に収束するとも限らない．そこで，関数 $f(x_k)$ が十分小

多項式 $y = x^3+3x^2+3x+1$ を計算した実行例

```
> gcc polynom.c
> a.out
Enter the degree of a polynomial: 3
Enter coefficients:
a[3]=1
a[2]=3
a[1]=3
a[0]=1
Enter values for polynomial.
To stop, enter EOF.
0
Polynomial = 1.000000e+00
1
Polynomial = 8.000000e+00
2
Polynomial = 2.700000e+01
-1
Polynomial = 0.000000e+00
^D
>
```

図 58: polynom.c の実行例

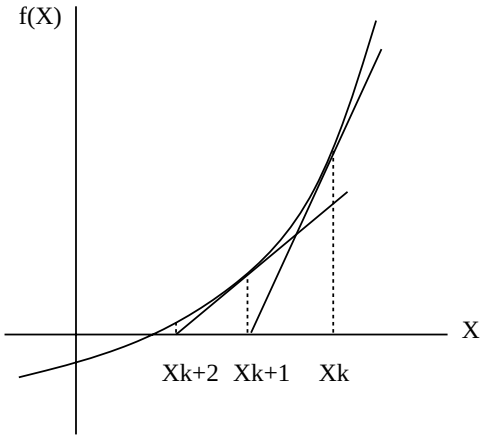


図 59: Newton-Raphson 法

さくなった時，すなわちある $\epsilon > 0$ に対して $f(x_k) < \epsilon$ が成立した時，収束したと判断し計算を打切り，その時の x_k を解とする．

ϵ の値が小さすぎると収束せず，大きすぎると解の精度が上がらない．適当な ϵ の値を決めるためには，

1. $f(x_k)$ の値は解の近くでどの程度の値になるのか．

2. $f(x_k)$ の値がどの程度の数値計算誤差を含むのか.

を見積もる必要がある．特に (2) で， $f(x)$ が解の近くで δ だけの数値計算誤差を持つとすれば，δ よりも小さい ϵ を与えても意味がない． ϵ の正しく見積もるのは難しいので， ϵ のを変えて計算を行ない，収束の様子を調べてみる必要がある．

24.3 アルゴリズム

方程式 $f(x) = 0$ における，f とf の導関数の関数定義をそれぞれf(x)，df(x) とする．

```
const double eps = 1.0e-8;
double x;
scanf("%lf", &x);            適当な初期値を読み込む
do                           繰返し計算
  x = x - f(x)/df(x);        解の修正
while(fabs(f(x)) > eps);     収束判定 (|f(x)| < ε)
printf("x = %e\n", x);       解の出力
```

関数の性質が悪い場合などにも，f(x) が中々 0 に近づかない場合がある．そのような場合に，上記のプログラムでは無限ループになってしまう可能性があるので，繰返し回数の上限を決めて，それ以上は繰返しを行なわないようにする．

```
const double eps = 1.0e-8;
double x;
int i=0;                                  繰返し回数カウンターのゼロクリア
const int max = 100;
scanf("%lf", &x);
do {
  x = x -f(x)/df(x);
  ++i; }                                  計算回数のカウント
while((fabs(f(x))>eps) && (i>max))        収束判定 (|f(x)| < ε) かつ
                                          繰返し回数> 最大繰返し数
if (i <= max) printf("x = %e\n", x);;
else printf("not converge\n");
```

24.4 n 次方程式の解法

第 23節で作成したpolynom を使って， n 次方程式を解くプログラムを作成する．つまり，方程式

$$\begin{aligned} f(x) &= \sum_{i=0}^{n} a_i x^i = 0 \\ &= a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0 = 0 \end{aligned}$$

を解く．導関数は,

$$f'(x) = \sum_{i=0}^{n} i a_i x^{i-1} = n a_n x^{n-1} + (n-1) a_{n-1} x^{n-2} + \cdots + a_1$$

$b_i = (i+1)a_{i+1}$ とおくと,

$$\begin{aligned} &= b_{n-1}x^{n-1} + b_{n-2}x^{n-2} + \cdots + b_0 \\ &= \sum_{i=0}^{n-1} b_i x^i \end{aligned}$$

つまり， $f'(x)$ は $b_i = (i+1)a_{i+1}$ を係数とする $(n-1)$ 次多項式となる.

よって，関数f(x) もdf(x) も関数polynom によって書ける．係数をa,b という配列で表すとpoly を用いて，f(x) はpolynom(a,n,x)，df(x) はpolynom(b,n-1,x) によって計算できる.

方程式 x^3-9x^2+26x-24=(x-2)(x-3)(x-4)=0 の解を計算した例

```
> gcc newton.c -lm
> a.out
Enter the degree of a polynomial: 3
Enter coefficients:
a[3]=1
a[2]=-9
a[1]=26
a[0]=-24
Enter an initial value:
To stop, enter EOF.
0
Solution = 2.000000e+00
100
Solution = 4.000000e+00
3.2
Solution = 3.000000e+00
^D
>
```

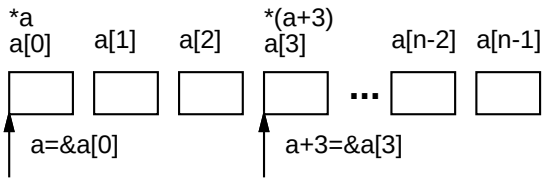
図 60: newton.c の実行例

```
 1  /* newton.c -- Newton Raphson Method */
 2  #include <stdio.h>
 3  #include <math.h>
 4  #define N 50

 5  double polynom(double a[], const int n, double x);
 6  double newton(double a[], const int n, double x);

 7  void main(void)
 8  {
 9    double a[N], x;
10    int n,i;

11    printf("Enter the degree of a polynomial: ");
12    scanf("%d", &n);
13    printf("Enter coefficients:\n");
14    for(i=n;i >= 0; i--) {
15      printf("a[%d]=",i);
16      scanf("%lf",&a[i]);
17    }
18    printf("Enter an initial value:\n");
19    printf("To stop, enter EOF.\n");
20    while(scanf("%lf", &x) != EOF)
21      printf("Solution = %e\n", newton(a,n,x));
22  }

23  double polynom(double a[], const int n, double x)
24  {
25    double s;
26    int i;
27    s = a[n];
28    for (i = n-1; i >= 0; i--) s = s*x + a[i];
29    return s;
30  }

31  double newton(double a[], const int n, double x)
32  {
33    double eps = 1.0e-10;
34    double b[N];
35    int i;

36    for(i=n; i >= 1; i--) b[i-1] = i*a[i];
37    do
38    x = x - polynom(a,n,x)/polynom(b,n-1,x);
39    while(fabs(polynom(a,n,x)) > eps);
40    return x;
41  }
```

図 61: newton.c

25 配列とポインタ

1. a[i] などの配列の表記は，ポインタによって書き変えることができる．
2. int a[MAX]; のように宣言された配列名a は，配列の先頭要素のアドレスを表す定数である (配列の先頭要素へのポインタ定数)．すなわち， a == &a[0] ．
3. ポインタに 1 加えることは，そのポインタが指す型のサイズをポインタの示すアドレスに加えることに等しい．例えば， a+1 は，a にint 型の要素のサイズ，すなわち 4(2) バイトを加えたものになり，それはa[1] のアドレスである．つまり，a+1 == &a[1] ．
   また，a+3 == &a[3] である[2]．一般に，a+i == &a[i].



4. 間接演算子* を使うと，配列の要素の値を参照することができる．a[3] の値は *(a+3) に等しい．(注意: *a + 3 の値は最初の要素a[0] に3 を加えたもの．) 一般に，a[i] == *(a+i).
5. 同様にして，ポインタから 1 引くことは，そのポインタが指す型のサイズをポインタから引くことに等しい．配列の場合には一つ前の要素のアドレスになる．
6. 配列を引数として関数に渡す時に注意すべき点は，引数としては，配列へのポインタしか渡せないことである．したがって，実際にポインタが配列を指しているのか，単なるポインタなのかは，関数側では区別できない．そのため，配列を引数として関数に渡す時には，配列のサイズを一緒に渡す必要がある．(sizeof で調べても，ポインタ変数のサイズしかわからない．)
7. 図 19のプログラムでは，配列の要素の和を，for (i=0; i < n; i++) sum += a[i]; で求めたが，これをポインタを使って書くと，for (i=0; i < n; i++) sum += *(a + i); と書ける．

   また，ここでは，a はポインタ定数であったが，下の例のようにa が仮引数の場合は，a は配列の先頭要素へのポインタ変数となるので，
   for (i=0; i < n; i++) {sum += *a; a++;} あるいは，
   for (i=0; i < n; i++) sum += *a++; のようにa を直接インクリメントすることができる．

```
int total(int a[], int n)
{ int sum=0, i;
  for (i=0; i < n; i++) sum += *a++;
  return sum;}
```

   このa++ は，1 だけ増えるのではなく，a の指す型の大きさだけ増えることに注意．

[2] &a[3] では，演算子[] の方が & よりも強い．

26　多次元配列：ガウスの消去法による連立一次元方程式の解法

1 次元の配列以外にも，2 次元の配列や必要ならばそれ以上の多次元の配列を使うことができる．ここでは 2 次元配列を使ってマトリクスを表現し，連立一次元方程式を解いてみよう．図63にプログラムを示す．

1. 連立一次方程式の解法としては最も簡単なガウスの消去法をを例題にする．これは連立方程式の係数行列を掃き出しによって上三角行列に変換する前進消去と，解を求める後退代入からなる．掃き出しを行う対角要素 (ピボット要素) の値が小さいときは，行の交換を行う．
2. `double a[MAX][MAX+1];`　多次元配列
   添え字を二つ以上もつ配列を多次元配列という．この例では係数行列として 2 次元の配列a を定義している．その要素は， `a[i][j]` のように書く．
3. `scanf("%lf",&a[i][j]);`
   変換仕様 `%lf` はdouble 型の浮動小数点の読み込みの変換仕様である．
4. 図 62に実行例を示す．これは，次の連立方程式を解いたものである．

$$\begin{pmatrix} 3 & -4 & 2 \\ 2 & 5 & 3 \\ -2 & 3 & -2 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} 0 \\ 6 \\ 1 \end{pmatrix}$$

```
> gcc gauss.c
> a.out
Enter the number of the simultaneous equation.
3
Enter coefficients in row-wise.
3 -4 2 0
2 5 3 6
-2 3 -2 1
x[1] =  2.00  x[2] =  1.00  x[3] = -1.00
>
```

図 62: gauss.c の実行例

```
 1  /* gauss.c -- gaussian elimination */
 2  #include <stdio.h>
 3  #include <stdlib.h>
 4  #define SWAP_DOUBLE( x, y ){double t; t = x; x = y; y = t;}
 5  #define MAX 31

 6  int readArray2(double [][MAX+1], const int);
 7  void gauss(double [][MAX+1], double [], const int);
 8  void printArray2(double [], const int);

 9  void main(void)
10  {
11    double a[MAX][MAX+1];
12    double x[MAX];
13    int n;
14    n = readArray2(a, MAX);
15    gauss(a, x, n);
16    printArray2(x, n);
17  }

18  int readArray2(double a[][MAX+1], const int limit)
19  {
20    int n;
21    int i,j;
22    printf("Enter the number of the simultaneous equation.\n");
23    scanf("%d", &n);
24    while( n <= 0 || n > (limit - 1)){
25      printf("The number must be positive");
26      printf("and less than or equal to %d.\n", limit-1);
27      printf("Enter, again\n");
28      scanf("%d", &n);
29    }
30    printf("Enter coefficients in row-wise.\n");
31    for(i = 1; i <= n; i++)
32      for(j = 1; j <= n+1; j++)
33        scanf("%lf",&a[i][j]);
34    return n;
35  }

36  void gauss(double a[][MAX+1], double x[MAX], const int n)
37  {
38    int i,j,k,largest;
39    double t;
40    for(i = 1; i < n; i++){
41      for(largest = i, j = i+1; j <= n; j++)
42        if(abs( a[i][j] ) > abs( a[largest][i] ))
43          largest = j;
44      for(k = i; k <= n+1; k++)
45        SWAP_DOUBLE( a[largest][k], a[i][k]);
46      for( j = i+1; j <= n; j++)
47        for( k = n+1; k >= i; k--)
48          a[j][k] = a[j][k]-a[i][k]*a[j][i]/a[i][i];
49    }
50    for(i = n; i >= 1; i--){
51      for(t = 0, j=i+1; j <= n; j++)
52        t = t + a[i][j]*x[j];
53      x[i] = (a[i][n+1] - t)/a[i][i];
54    }
55  }

56  void printArray2(double x[MAX], const int n)
57  {
58    int i;
59    for(i = 1; i <= n; i++){
60      printf("x[%d] = %5.2lf  ",i,x[i]);
61      if( i%3 == 0 ) printf("\n");
62    }
63    if( i%3 != 1) printf("\n");
64  }
```

図 63: gauss.c

27　多次元配列とポインタ

1. まず，1 次元配列`int a[3];` の意味を考えてみよう．

   (a) `int a[3] ;` 要素`a[0]`, `a[1]`, `a[2]` は`int` 型．

   (b) `int a [3] ;` `a` は`int` が`[3]` 個並んだ配列．つまり`a` は`int [3]` 型．
   ⇒ `a` は，その配列の先頭要素`a[0]`(`int` 型) のアドレス，つまりポインタ (定数)．
   `a == &a[0];　*a == a[0];`
   ⇒ `a` は，`int *` 型の変数に代入可能．(`&a` は`&a[0]` に等しい (ANSI C))．

   (c) 配列表記とポインタ表記の関係

配列`a[N]` に対して，	`a[i]`⟺`*(a + i)`
	特に，`a[0]`⟺`*a`.
	`&a[i]`⟺`a + i`

ポインタ`p` に対して，	`*(p + i)`⟺`p[i]`

2. 多次元配列`int a[3][2];` の意味

   (a) `int a[3][2] ;` 要素`a[0][0]`, `a[0][1]`, `a[0][2]` … `a[2][1]` は`int` 型．

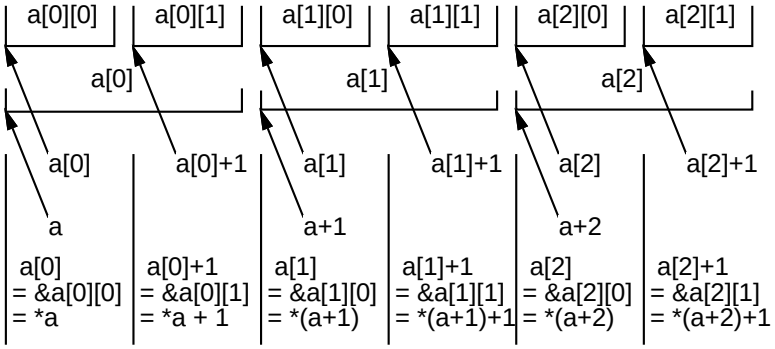
   (b) `int a[3] [2] ;` `a[0]`, `a[1]`, `a[2]` は，それぞれ`int` が`[2]` 個並んだものの 1 次元配列．例えば`a[0]` の要素は，{ `a[0][0]`, `a[0][1]`}．
   ⇒ `a[0]`, `a[1]`, `a[2]` は`int [2]` 型で，1 次元配列の先頭要素 (`int` 型) へのポインタ．つまり`int *` 型
   `a[0] == &a[0][0]`, `a[1] == &a[1][0]`, `a[2] == &a[2][0]`

   (c) `int a [3][2] ;` : `a` は`int [3][2]` 型．
   ⇒ `a` は`int [2]` が`[3]` 個並んだ”1 次元” 配列と考えられる．つまり配列の配列．
   ⇒ `a` は，その配列の先頭要素`a[0]`(`int [2]` 型) へのポインタ (`&a[0]`)．さらに`a[0]` は`int` 型へのポインタだから，`a` は”`int` 型へのポインタのポインタ” となる (2 重間接)．
   つまり，`int **` 型

```
          a          ⟶     a[0]     ⟶  a[0][0]
    int [3][2] 型          int [2] 型      int 型
int [2] 型へのポインタ        int *
       int **
```

   `a` は`int **` 型のポインタ定数であるから，値を`int **` 型の変数や仮引数に代入したり，値を渡したりできる．

   (d) `a` が表す`&a[0]` も，`a[0]` が表す`&a[0][0]` も，両方とも配列`a` の先頭のアドレスとなる．アドレスそのものは同じでも，その意味が異なる．しかし，前者は`int [2]` のアドレスであり，後者は`int` のアドレスである．この違いは，例えば`a + 1` と`a[0] + 1` の違いとなって現われる (`a+1` は，先頭アドレスに対して，`int [2]` の長さ分だけ増えているの対し，`a[0]+1` は`int` の長さ分だけ増える)．



(e) 2 次元配列の要素の配列表記とポインタ表記

`a[i][j] == *(*(a + i) + j);`

実際のアドレスを計算するには,

1 次元配列 `int a[N];` の場合：`&a[i]` は 先頭 ＋ `i*sizeof(int)`

2 次元配列 `int a[N][M];` の場合： `&a[i][j]` は 先頭 `+(i*M + j)*sizeof(int)`

よって 2 次元配列を引数で渡す時には，その行のサイズ (二つ目の次元の要素の数`M`) を渡す必要があることがわかる．(例題プログラム`gausss` では，`M+1` を渡している．)

一般に，多次元では，一つ目の次元以外は，すべてそのサイズを明記する．

(f) 2 次元配列の初期値設定

```
static int a[2][3] = {{35, 75, 79},{23, 59, 82}};
```

28 文字列

1. 一つ以上の文字 (char) の並びを 文字列 (character string) という.
2. 文字列は，文字型char の配列 (char [] 型) で表す．例えば， char s[10]; は最大 10 文字の長さの文字列．但し，文字列の終りを示す ナル文字'\0 を最後の文字として入れる約束なので，実質的には 9 文字以下の文字列となる．
   s = 123456789\0　　s = abcde\0␣␣␣␣
3. s は，配列の先頭の要素へのポインタ定数となる．先頭要素はchar 型であるから，s は (char *) の定数となる.
4. 文字列中の第 i 番目の文字は，s[i] で表す.

```
 1  /* reverse.c -- string reverse */
 2  #include <stdio.h>
 3  #include <string.h>
 4  #define SWAP_CHAR( x, y ) {char c; c = x; x = y; y = c;}
 5  #define MAX 81
 6  void reverse(char []);

 7  void main(void)
 8  {
 9    char text[MAX];

10    printf("Enter a character string.\n");
11    gets(text);
12    reverse(text);
13    puts(text);
14  }

15  void reverse(char t[])
16  {
17    int i,j;
18    for(i = 0, j = strlen(t)-1; i < j; i++, j--)
19      SWAP_CHAR(t[i], t[j]);
20  }
```

図 64: reverse.c

図 64のプログラムは入力された文字列を逆順に表示するプログラム.

1. #include <string.h>
   strlen などの文字列処理を行なう関数のためのヘッダーファイル.

2. gets(char []) 関数
   文字列の読み込み．gets(text); は，キーボードから，文字列text に，改行までの文字列を読み込む．改行記号は捨てられ，その変わりにナル文字を最後につけて文字列とする．text に割り当てられた文字列の長さを越えていないかどうかのチェックはしない．gets は返戻値として，文字列 (文字へのポインタ) を返す．つまり，char *gets();.

   読み込みにエラーがあったり， EOF に出合うと，返戻値としてNULL(ナルポインタ，値 0)を返す．scanf でも，変換仕様%s によって文字列を読み込むことができる．この場合は空白が区切りとなる．

3. puts(char []) 関数
   文字列を表示する関数．関数printf でも，変換仕様%s によって表示できる．

4. void reverse(char []);
   このプロトタイプ宣言は，関数reverse は，値を返さない (void) 関数で，その引数は，文字の配列，つまりchar [] 型であることを示す．

5. void reverse(char t[]); はvoid reverse(char *t); としてもよい．

6. strlen
   文字列の長さ (ヌル文字まで (含む) の長さ) を返す関数．

7. 文字列定数
   ""で囲んだ文字列が文字列定数であり，文字列の変数に代入すると，最後にナル文字をつけて代入される．

   "を文字列に含めたい時には，バックスラッシュを付けて\"のようにする．

8. 文字列の初期設定
   配列の初期設定の方法や，文字列定数を使う．

```
char s1[5] = {'a', 'b', 'c'};          × ナル文字がないので文字列にならない．
char s2[5] = {'a', 'b', 'c', '\0'};    ○
char s3[3] = {'a', 'b', 'c', '\0'};    × 文字列のサイズを越えているのでエラー．
char s4[] = {'a', 'b', 'c', '\0'};     ○
char s5[] = "abc";                     ○ 文字列定数
static char s5[5] = "abc";             ○ 文字列定数 (静的変数)
char *s6 = "abc";                      ○ 文字へのポインタ変数
```

28.1 文字列とポインタ

図 65のプログラムは，文字列をコピーする関数の例．copy0 とcopy1，copy2 は同じ処理を行う関数である．

1. copy0 は，文字列to とfrom を受け取り， 1 文字ずつコピーしてゆく．そのためにfor 文で i の値を 0 から 1 ずつ増やしながら，文字列from のi 番目の文字 (from[i]) を文字列to のi 番目の文字 (to[i]) に代入してゆく．この繰り返しは，from[i] がヌル文字になるまで繰り返される．このfor 文が終了した段階では，文字列to には，文字列の終わりを示すヌル文字が代入されていないので，to[i]='\0'; によって，それを代入しておく．

```
 1  /* strcopy.c  string copy */
 2  void main(void){
 3    char x[] ="abcdefg";
 4    char y[10];
 5    copy2(y,x);   /* using copy2 */
 6    puts(y);
 7  }

 8  void copy0(char to[], char from[])
 9  {
10    int i;
11    for(i = 0; from[i]!='\0'; i++) to[i] = from[i];
12    to[i] = '\0';
13  }

14  void copy1(char *to, char *from)
15  {
16    while(*from != '\0') *to++ = *from++;
17    *++to = '\0';
18  }

19  void copy2(char *to, char *from)
20  {
21    while( *from ) *to++ = *from++;
22    *++to = '\0';
23  }
```

図 65: strcopy.c

2. copy1 では文字列をポインタで扱っている．これは，配列をポインタで扱うのと同じようにして行う (25節参照).
   `void copy1(char *to, char *from)`
   ここで仮引数to とfrom は，文字型へのポインタとして宣言されている．これに実引数x，y，すなわち文字列x，y の先頭の文字へのポインタが渡され，ポインタto とfrom は，それぞれ文字列x，y の先頭の文字を指す.

3. 間接演算子* を使って
   `*to = *from;`
   とすることによって，ポインタfrom の指す文字が，ポインタto の指す文字に代入される．さらに，
   `to++;  from++;`
   とすると，ポインタto, from は，それぞれ文字列の 2 番目の文字を指すようになる．そこで，再び上記の代入文を行うことによって 2 番目の文字が代入される．これらの代入文は，
   `*to++ = *from++;`

のように一つにまとめることができる (copy2 参照). ここで, ++ は後置演算子なので, 間接演算子* によって, to, from の値が参照された後に, to とfrom の値 (文字型を指すポインタ) はインクリメントされ, それぞれ次の文字を指す.

4. この代入文をfor 文で繰り返す. 繰り返しの終了条件は, from の指す文字がヌル文字になることである. while(*from != '\0') は, ナル文字の値が 0 であることを利用して, while(*from) と簡潔に書ける.

28.2 文字列に関する注意

文字列を扱う時には次の点に注意.

- 文字列の長さを越えていないかどうか.
- 文字列の領域がきちんと確保されているかどうか.

特に二つ目については注意が必要である. 例えば, 図 66を考えて見よう. 文字列を表す変数としてs が宣言されている. しかし, この宣言だけでは, s は文字列を格納するための有効な記憶領域を指していないので, このプログラムは図 67のようなエラーを出して異常終了する.

```
1  #include <stdio.h>

2  void main(void)
3  {
4    char *s;
5    /* WRONG */
6    gets(s);
7    printf("%s\n", s);
8  }
```

図 66: str1.c

```
> gcc str1.c
> a.out
abc
Segmentation fault
>
```

図 67: 誤りプログラム str1.c の実行例

これを修正するには, 図 68のように, 文字列を配列として宣言し, 記憶領域を確保する.

```
1  #include <stdio.h>

2  void main(void)
3  {
4    char s[20];
5    /* GOOD */
6    gets(s);
7    printf("%s\n", s);
8  }
```

図 68: str2.c

また，別の方法としては，記憶領域を確保する関数malloc を使うものがある．図 69に示すように，malloc(20) は，必要な記憶領域の大きさ (バイト単位，この場合は 20.) を受けとり，計算機のメモリー領域の中にその大きさだけの領域を確保して，その先頭のアドレス (ポインタ) を戻す．(char *) は，キャスト演算子で，このmalloc の戻すポインタ (void *) を文字型へのポインタに変換している．malloc については，第 36節で詳しく述べる．

```
1  #include <stdio.h>

2  void main(void)
3  {
4    char *s;
5    /* OK */
6    s = (char *)malloc(20);
7    gets(s);
8    printf("%s\n", s);
9  }
```

図 69: str3.c

さて，図 70を見てみよう．これは，文字列s に char *s="12345678901234567890"; によって初期化を行なったものである．初期化によって，s には，この文字列分だけの記憶領域が割り当てられるから，このプログラムは一見うまく動きそうであるが，実はエラーが発生する．

それは， char *s="12345678901234567890"; によってs は，文字列データ "12345678901234567890"の格納されているアドレスを指すことになるのだが，この領域は定数を表す領域，つまり値を書き変えることが禁止されている領域である．したがって，gets(s); によって，読み込んだ文字列をそこに書き込もうとしてエラーになる．

28.3 文字列関数

文字列を操作するいくつかの関数がある．

```
1  #include <stdio.h>

2  void main(void)
3  {
4    char *s="1234567890123456789";
5    /* WRONG */
6    gets(s);
7    printf("%s\n", s);
8  }
```

図 70: str4.c

1. `int strcmp(char s1[], char s2[])`
   二つの文字列を比較し，同じなら 0 を返す．それ以外の時は，辞書式順序 (アルファベット順) で，s1<s2 ならば負数， s1>s2 ならば正数を返す．(s1 == s2 は，ポインタの値を比較しているだけなので，文字列の比較にはならないことに注意)

2. `char *strcpy(char s1[], char s2[])` :s2 をナル文字も含めてs1 にコピー．返戻値はs1.

3. `char *strcat(char s1[], char s2[])` :s2 をs1 に連結．返戻値はs1.

4. `int atoi(char [])`: 数字からなる文字列を，対応する整数に変換．例:"123"を 123 に変換.

5. `double atof(char [])`: 同様にdouble に変換.

6. `long atol(char [])`: 同様にlong に変換.

29 応用: パタンマッチプログラム

```
 1  /* patern.c -- pattern search */
 2  #include <stdio.h>
 3  #include <string.h>
 4  #define MAX 81
 5  #define NOT_FOUND -1

 6  int search(char [], char []);

 7  void main(void){
 8    char text[MAX], pattern[MAX];
 9    int i;

10    printf("Enter the text.\n");
11    gets(text);
12    printf("Enter a pattern.\n");
13    gets(pattern);
14    if((i = search(text, pattern)) == NOT_FOUND){
15      printf("No string matched the pattern:");
16      printf("%s\n",pattern);
17    } else {
18      printf("The string which started at the %dth character ",i);
19      printf("matched the pattern.\n");
20    }
21  }

22  int search(char text[], char pattern[]){
23    int i,j;
24    int patternLength, textLength;

25    i = 0; j = 0;
26    patternLength = strlen(pattern);
27    textLength = strlen(text);
28    while( ( j < patternLength ) && ( i < textLength ) )
29      if( text[i] == pattern[j] ){
30        i++; j++;
31      } else {
32        i -= j-1; j = 0;
33      }
34    if( j >= patternLength )
35      return( i - patternLength );
36    else
37      return NOT_FOUND;
38  }
```

図 71: pattern.c

図 71は，二つの文字列text とpattern を入力して，text がpattern を含んでいるかどうかを調べるプログラム．

30 再帰関数

関数の定義の中に，自分自身が現れる関数を再帰関数という．他の多くの言語と同様に，Cでは，一般の関数と変わることなく定義できる．

30.1 ユークリッドの互助法

例題として，ユークリッドの互助法の再帰的アルゴリズムを考えよう．すなわち，整数 a と b の最大公約数を $gcd(a,b)$ とすると，

$a > b$ の場合は，$gcd(a,b) = gcd(a-b,b)$
$a < b$ の場合は，$gcd(a,b) = gcd(a,b-a)$
$a = b$ の場合は，$gcd(a,b) = a = b$ (停止条件)

1. 再帰関数
   再帰呼び出し (recursive call) を含む関数のこと．図 72は，ユークリッドの互助法の再帰関数版．関数gcd が再帰関数.

```
 1  /* rgcd.c --- recursive version of gcd() */
 2  int gcd(int a, int b);

 3  void main(void)
 4  {
 5    int x, y;
 6    printf("Enter two positive intergers: ");
 7    scanf("%d %d", &x, &y);
 8    printf("%d\n", gcd(x, y));
 9  }

10  int gcd(int a, int b)
11  {
12    int z;
13    if( a != b)
14      if( a > b ) z = gcd(a-b,b);
15      else z = gcd(a,b-a);
16    else
17      z = a;
18    return z;
19  }
```

図 72: rgcd.c

実行の様子は図 73のようになる．

2. 関数が呼び出されるたびに，関数の局所変数 (z) や仮引数 (a，b) は，” 新しいもの” が用意される．gcd(呼び出し側) がgcd(呼ばれる側) を呼び出す場合，呼び出し側のgcd の使う

```
gcd(24,18)
   |z = gcd(18,6)
        gcd(a, b)
           |z = gcd(12,6)
                gcd(a, b)
                   |z = gcd(6,6)
                        gcd(a,b)
                           |z = 6;
                           |return z(=6)
                   |return z(=6)
           |return z(=6)
   |return z(=6)
```

図 73: 再帰関数 gcd の実行の様子

これらの変数と，そこから呼び出されるgcd の使うこれらの変数は別のものとなる (新しく作られる)．従って，呼び出されたgcd がz などの値を書き換えても，呼び出した方のgcd のz には何ら影響がない．

3. 再帰呼び出しを行なう関数では，必ず再帰を打ち切る停止条件が必要である．

4. この関数のようにreturn の直前で再帰を行なうものを端末再帰 (tail recursive) という．端末再帰のプログラムは繰り返しで書き直すことができる (参考: 図 30のプログラム)．

30.2 再帰から戻りながら処理するプログラム

1. 2 進数変換
   正の整数 (10 進数) を読み，対応する 2 進数に変換するプログラム (toBinary) を図 74に示す．4(10 進数) の 2 進数表現は 100 となるが，それを求めるには，2 での割算 /2 と，余り %2 の計算を繰り返す．例えば，4 の 2 進数表現 100 を求める手順は，

n	n % 2	
4	4 % 2 = 0	1 の位 (左から一つ目の位)
4 / 2 = 2	2 % 2 = 0	2 の位 (左から二つ目の位)
2 / 2 = 1	1 % 2 = 1	4 の位 (左から三つ目の位)

2.
```
if( n != 0 ){
  r = n%2;
  toBinary( n/2 );
  printf("%1i",r);
```

   再帰関数において，再帰呼び出しよりも後ろにある文は，関数が呼び出されたのと逆の順序で実行される．つまり，停止条件によって再帰的な呼び出しがそれ以上起こらなくなり，順次一つ前の再帰呼び出しのところへと戻っていくが，それぞれ一つ前の呼び出し点に戻るところでこれらの文は，実行される．

   n=4 の場合の実行の様子は図 75のようになる．

3. printf("%1i",r); の%1i は，整数を 1 桁で出力するための指定である．

```
 1  /* binary.c -- print positive interger in binary notation */
 2  #include <stdio.h>
 3  void toBinary(int);

 4  void main(void)
 5  {
 6    int n;
 7    printf("Enter a positive number: ");
 8    scanf("%d", &n);
 9    toBinary(n);
10    putchar('\n');
11  }

12  void toBinary(int n)
13  {
14    int r;
15    if( n != 0 ){
16      r = n%2;
17      toBinary( n/2 );
18      printf("%1i",r);
19    }
20    return;
21  }
```

図 74: binary.c

```
toBinary(4)
   |r = 4 % 2 = 0
   |toBinary(4/2 = 2) の呼び出し
       toBinary(2)
          |r = 2 % 2 = 0
          |toBinary(2/2 = 1) の呼び出し
              toBinary(1)
                 |r = 1 % 2 = 1
                 |toBinary(1/2 = 0) の呼び出し
                     toBinary(0)
                        |return
                 |printf(r) 1 の表示
                 |return
          |printf(r) 0 の表示
          |return
   |printf(r) 0 の表示
   |return
```

図 75: 再帰関数 toBinary の実行の様子

31　関数ポインタ

31.1　台形則による数値積分

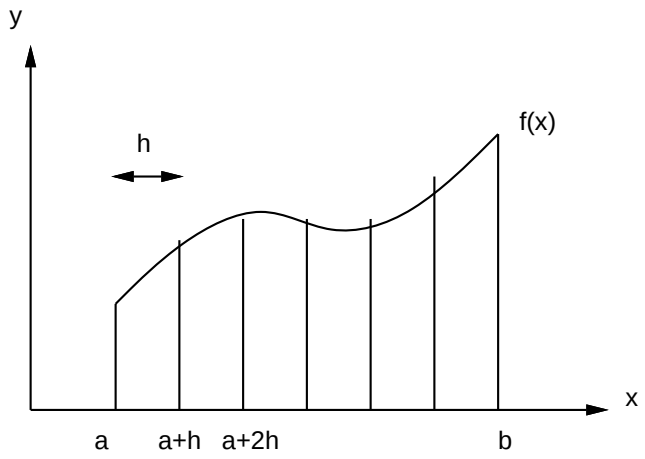
1. 定積分を数値的に計算する方法として，関数を折れ線近似し，折れ線によって構成される台形の面積の和として，定積分を近似計算する台形則がある．定積分 $\int_a^b f(x)dx$ に対し，積分区間 $[a,b]$ を n 等分し，その刻みを $h(=(b-a)/n)$ とする．また，分点 x_i を x の小さい方から，$x_0=a$, $x_1=a+h$, $x_2=a+2h$, …, $x_n=a+nh=b$ とする．

   図の様に関数 $f(x)$ を，小区間 $[x_i, x_{i+1}]$ において点 $(x_i, f(xi))$ と点 $(x_{i+1}, f(x_{i+1}))$ を結ぶ直線で近似すると，この小区間における定積分 $\int f(x)dx$ は，台形の面積 $h/2\cdot(f(x_i)+f(x_{i+1}))$ で近似できる．これをすべての小区間について足しあわせれば，

$$\begin{aligned}\int_a^b f(x)dx &\approx \sum_{i=0}^{n-1}\int_{x_i}^{x_{i+1}} f(x)dx = \sum_{i=0}^{n-1} h/2(f(x_i)+f(x_{i+1}))\\ &= h/2(f(x_0)+f(x_1)) + h/2(f(x_1)+f(x_2)) + \cdots + h/2(f(x_{n-1})+f(x_n))\\ &= h(1/2\cdot f(x_0) + f(x_1) + f(x_2) + \cdots + f(x_{n-1}) + 1/2\cdot f(x_n))\\ &= h(A+B)\end{aligned}$$

   但し，

$$\begin{aligned}A &= 1/2\cdot f(x_0) + 1/2\cdot f(x_n) = 1/2(f(a)+f(b))\\ B &= f(x_1)+f(x_2)+\cdots+f(x_{n-1}) = \sum_{i=1}^{n-1} f(x_i) = \sum_{i=1}^{n-1} f(a+ih)\\ h &= (b-a)/n\end{aligned}$$



2. プログラム例: 積分区間の下限と上限をそれぞれ変数`a`,`b` で表し，分割数を`n`，刻みを`h`，積分値を`s` とし，また被積分関数が`double f(double);` で定義されているとすると，

```
h = (b - a)/n;                                  刻み h の計算
s = (f(a) + f(b))/2.0;                          A の計算
for(i = 1; i <= n-1 ; i++)  s += f(a + i*h);    A に B を加える
s = s*h;
```

31.2 関数へのポインタ

台形則で積分を行なうプログラムを図 76のように作成した．ここで，関数trape が数値積分を行なう関数であるが，その被積分関数は目的によって変わるので，被積分関数そのものも関数の引数として与えられるようにしてある．

1. `printf("Result = %e\n", trape(a,b,g));`
   ここで， trape(a,b,g) の三つ目の引数は関数名g で，関数へのポインタを表す．一方これを受けとるtrape の仮引数は次のように宣言される．
   `double trape(double a, double b, double(*f)(double))`
   最後のdouble(*f)(double) が，関数へのポインタを受けとる仮引数f の宣言である．

   例えば，宣言 `double h1(double);` は，h1 が，double 型の引数を一つとり，double 型の値を返す関数であることを示す．つまり，関数h1 は，`double (double)` という関数型であるといえる．そうすると，double(*h2)(double) というのは，(*h2) が `double (double)` 型であること，つまり，h2 がそのような型へのポインタであることを表す．例題の関数g も，double 型の引数を一つとり，double 型の値を返す関数であるので，実引数g はdouble (double) 型の関数へのポインタを表し，仮引数f に渡される．

2. このようにして得た関数へのポインタf を使って，関数を呼び出すには，
   `s += (*f)(a + i*h);` あるいは，単に `s += f(a + i*h);` のようにする． C では前者が慣例である．

32 * と () と [] の結合則

- 宣言に現われる* () [] の結合規則:
  1. [] と() の結合の強さは同じ．
  2. [] と() のほうが* より強い．
  3. [] と() との結合は左から右．
  4. カッコ() を用いて結合の順序を指定できる．カッコのついた結合が最も強い．
- 例:
  1. `int *color[7];` color は七つの要素をもつ配列で，要素はint * 型．
  2. `int cards[4][13];` cards は四つの要素をもつ配列で，その要素が 13 の要素をもつ配列． 13 の要素はint 型．
  3. `int (*ptr)[10];` ptr はポインタである．それは， 10 個のint 型の要素をもつ配列へのポインタである．
  4. `int *func();` func は関数であり，それはint * 型つまり，int へのポインタを返す．
  5. `int (*foo)();` foo はポインタである．それは，関数へのポインタであり，その関数はint を返す．つまり，func は，int を返す関数へのポインタ．
  6. `int (*clock[4])();` clock は四つのポインタを要素とする配列である．個々の要素は，int 型を返す関数である．clock は，int 型を返す関数へのポインタを要素を四つもつ関数である．

```
 1  /* integ.c trapezoidal numerical integration */
 2  #include <stdio.h>
 3  #include <math.h>

 4  #define N 50

 5  double trape(double, double, double(*)(double));
 6  double g(double);

 7  void main(void)
 8  {
 9    double a,b;

10    printf("Enter a, b ?\n");
11    scanf("%lf %lf",&a,&b);
12    printf("Result = %e\n", trape(a,b,g));
13  }

14  double trape(double a, double b, double(*f)(double))
15  {
16    double h, s;
17    int i;

18    h = (b - a)/N;
19    s = ((*f)(a) + (*f)(b))/2.0;
20    for(i = 1; i <= N-1 ; i++)  s += (*f)(a + i*h);
21    return s*h;
22  }

23  double g(double x)
24  {
25    return (sqrt(4 - x*x));
26  }
```

図 76: integ.c

33 構造体

1. 複素数は実部と虚部という二つの実数で構成され，また， 3 次元空間の点は (x,y,z) という三つの実数の組で表現される．

   このような複数の要素をもつデータを表現するには，配列を用いることができるだろう．例えば，複素数 x を表現するには，次のように 2 個の要素を持つ配列を定義する．
   `double x[2];`

   そして， `x[0]` には実部を表す実数を入れ， `x[1]` には虚部を表す実数を入れることにする．

2. さて，このように 1 単位の情報が複数の要素データから構成されることは一般によくあることである．例えば，学生名簿を考えると，名簿には一人一人の学生に関するデータとして，氏名，年齢，住所，電話番号などが必要となる．これらの一つ一つの項目は，氏名ならば文字列，年齢ならば整数型などで表現できる．

   では，複素数の時のように，学生に関するデータ項目を**ひとまとまり**にして表現するにはどうしたらよいだろうか．複素数の場合には，実部と虚部という要素がすべて同じ実数型であったので，配列を使うことができた．しかし，学生の場合には氏名は文字列であるし，年齢は整数型であるから，配列を使うことはできない．

3. 構造体とメンバー
   このように複数のデータ項目から構成されている情報に関して，それらのデータ項目を一まとめにして表現するのには，構造体を用いる．

   例えば学生データを表すには，次のような構造体を定義する．

```
struct student {
  char name[51];
  int age;
  char address[101];
  char phone[16];
  };
```

   ここで，student は，この構造体の名前 (識別子) であり，タグ という．
   また，name，age，address，phone は，この構造体を構成する要素であり，これを構造体のメンバーという．

   複素数を表す構造体は，メンバーを re と im とすれば，次のように宣言できる．

```
struct complex {
  double re;
  double im;
  };
```

   これらのstruct 宣言は，その構造体がどのような要素から構成されているかを宣言する．これを，構造体のテンプレートを宣言している，という．

   宣言の中では，同じ名前のメンバー名が現れてはならない．メンバー名やタグ名は他の変数名などと同一でもよい．

4. 構造体変数の定義
さて，実際に個々の複素数や学生のデータを表現するには，構造体テンプレートを使って，変数を定義する必要がある．複素数を表現する二つの変数 x と y を定義するには，
`struct complex x, y;`
のようにする．つまり，`struct タグ名 変数名, …;`
このように `struct complex` は，`int` などの型名と同等である．
例えば，`struct complex` の配列を宣言するには，
`struct complex z[10];`
のようにする．

5. その他の定義方法
テンプレートと変数定義を一緒にすることもできる．また，下の右の例のようにテンプレートを一度しか使わないときには，タグを省略することもできるが，あまり好ましい方法ではない．

```
struct complex {
  double re;
  double im;
  } x, y;
```

```
struct {
   double re;
   double im;
  } x, y;
```

6. メンバ演算子 .
例えばstruct complex x; で宣言された複素数を表す構造体変数x は，その” 内部” に，実部re と虚部im のデータを持っている．これらの値を読み出したり，代入したりするには，メンバ演算子 . を使い，変数名の後にメンバ名をつける．
例えば， x を複素数 $1.2 + 3.9i$ にするには，

```
x.re = 1.2;
x.im = 3.9;
```

x.re とx.im などは，double 型の変数である．構造体の配列の時には，z[5].im のように書く．

7. 図 77は，ある複素数を読み込んで，その共役複素数を表示するプログラム．

34 構造体と関数

1. 構造体は関数の引数としても，関数の結果の値 (返戻値) としても用いることができる．次の関数getComplex は，構造体の値を返す関数であり，関数の型がstruct complex として宣言されている．

```
struct complex getComplex(void)
{
  struct complex x;
```

```
 1  /* conj1.c  conjugate complex number */
 2  #include <stdio.h>

 3  struct complex {
 4    double re;
 5    double im;
 6  };

 7  void main(void)
 8  {
 9    struct complex x, y;
10    printf("Enter an imaginary number\n");
11    scanf("%lf %lf", &(x.re), &(x.im));

12    y.re = x.re;
13    y.im = -x.im;
14    printf("Conjugate complex is %lf %lf\n", y.re, y.im);
15  }
```

図 77: conj1.c

```
    printf("Enter imaginary number\n");
    scanf("%lf %lf", &(x.re), &(x.im));
    return x;
  }
```

次の関数conjugate は，構造体の値を仮引数p に受けとる．これは値渡しであるので，例えばconjugate(x) のように呼び出すと，構造体x の値が，p にコピーされる．そして，p の共役をとり，その値 (構造体) を返す．

```
struct complex conjugate(struct complex p)
{
   p.im = - p.im;
   return p;
}
```

これを使ったプログラムを示す．

補足

(a) 宣言における構造体の初期化

```
struct complex x={1.2, 2.3};
```

```
 1  /* conj2.c  conjugate complex number */
 2  #include <stdio.h>

 3  struct complex {
 4    double re;
 5    double im;
 6  };

 7  struct complex getComplex(void);
 8  struct complex conjugate(struct complex);

 9  void main(void)
10  {
11    struct complex x, y;
12    x = getComplex();
13    y = conjugate(x);
14    printf("Conjugate complex is %lf %lf\n", y.re, y.im);
15  }

16  struct complex getComplex(void)
17  {
18    struct complex x;
19    printf("Enter imaginary number\n");
20    scanf("%lf %lf", &(x.re), &(x.im));
21    return x;
22  }

23  struct complex conjugate(struct complex p)
24  {
25     p.im = - p.im;
26     return p;
27  }
```

図 78: conj2.c

(b) 構造体の代入: 構造体から構造体への代入ができる.

```
struct complex x={1.2, 2.3};
struct complex y;
y = x;
```

(c) 構造体同士の比較は出来ない. 例) × `x == y`

35　構造体へのポインタ

関数に構造体の値ではなく，構造体へのポインタを渡すこともできる．前述の関数conjugateを変更してみよう．

1. このとき，関数が受けとるものは，struct complex 型へのポインタである．つまり，struct complex * 型になる．つまり，関数の頭部は次のようになる．
   void conjugate(struct complex *p)
   関数がvoid 型になっているが，それは，この場合にはポインタを受けとって直接に構造体の値を変更できるので，関数自体としては値を返さないようにしているからである．

2. 引数p は，構造体ではなくて，構造体へのポインタであるから，その虚部や実部のメンバーにアクセスするのに，単にメンバー演算子を使ってp.im とやっただけでは駄目．つまり，ポインタが示すアドレスに格納されている値が構造体データであるから，まず間接演算子*を用いて，*p として構造体の値を得，それに対してメンバー演算子を適用する．すなわち，(*p).im のようにする．つまり，

```
void conjugate(struct complex *p)
{
  (*p).im = - (*p).im;
}
```

   この表現は頻繁に現われるので，C ではより簡潔に書くために，間接メンバ演算子-> が用意されており，p->im のように書ける．これを用いると，関数conjugate は，次のようになる．

```
void conjugate(struct complex *p)
{
  p->im = - p->im;
}
```

   いまx が複素数の場合，それを共役複素数にするには，conjugate(&x) とする．

3. 図 79のプログラムは，ポインタ渡しによるプログラムの例である．

4. 構造体の値が関数に渡されるときには，他の変数と同様にその構造体の値のコピーが作られる．しかし，構造体のメンバーの数や大きさが増え，そのデータ量が大きくなると，関数を呼ぶたびにコピーを作るのは効率のよいことではないので，その場合にはポインター渡しにする．

36　自己参照構造体

1. リスト構造
   2 次元平面上の点列を表すことを考える．個々の点は次の構造体を用いて表現する．

```
 1  /* conj3.c -- conjugate complex number (pointer version) */
 2  #include <stdio.h>

 3  struct complex {
 4    double re;
 5    double im;
 6  };

 7  struct complex getComplex(void);
 8  void conjugate(struct complex *);

 9  void main(void)
10  {
11    struct complex x;
12    x = getComplex();
13    conjugate(&x);
14    printf("Conjugate complex is %lf %lf\n", x.re, x.im);
15  }

16  struct complex getComplex(void)
17  {
18    struct complex x;
19    printf("Enter imaginary number\n");
20    scanf("%lf %lf", &(x.re), &(x.im));
21    return x;
22  }

23  void conjugate(struct complex *p)
24  {
25    p->im = - p->im;
26  }
```

図 79: conj3.c

```
struct point {
  double x;
  double y;
  };
```

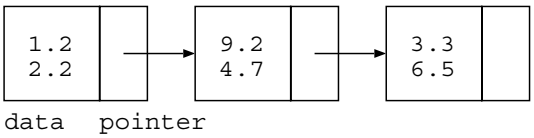
点列は点の並びであるから，例えば配列を用いて，

```
struct point rp[100];
```

のようにすれば，100 個までの点を持つ点列 rp を表現することができる．

しかし，処理する点の数があらかじめわからない時には，このような方法は不便である．つまり，配列を用いていると，プログラムを書いた (コンパイルした) 時点で，処理できる最大の点の数のが決まってしまう (静的記憶割付け) からである．そうすると，計算機にはまだたっぷりとメモリーが残っているのに，プログラムを書き直さなければ，どうやってもそれ以上の多くの点を扱うことができないということがおきてしまう．

そこで必要に応じてメモリー領域を割り当てていく (動的記憶割付け) 方法を用いる．リスト構造はそのような 動的データ構造 の一種であり，データをポインタによって鎖状につないだものである．必要に応じて新しい要素をこのリストに次々とつないでゆく．



2. 自己参照構造体

上図のようにリスト構造の一つの箱 (要素) は，データ部とポインタ部からなっている．上で点をstruct point {double x; double y;}; という構造体で表したように，この要素も構造体 (名前を plist とする) で表すことを考えてみよう．

まず，先頭の要素に注目すると，データ部には 1.2, 2.2 のように二つの座標値が入っており，ポインタ部には 2 番目の要素へのポインタが入っている．つまり，データ部を表すためには，二つのdouble 型のメンバー (名前を x，y とする) を用意すればよく，またポインタ部には，2 番目の要素を指すポインタが入るメンバを一つ用意すればよい．このメンバは，” 次の” 要素を指すので next という名前にする．

これで構造体 plist には三つのメンバー x，y，next があり，最初の二つはdouble 型であることが分かった．三つ目のポインタ next について考えてみよう．

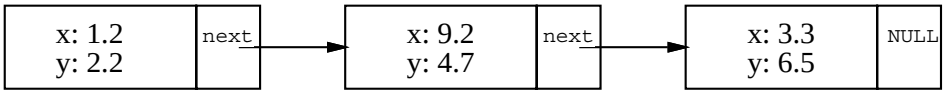
リスト構造では，同じ型の要素が並んでいるから，当然先頭要素と 2 番目の要素は同じ型の構造体，つまり plist 型の構造体である．したがって，この next ポインタが指す 2 番目の要素も plist 型の構造体である．つまり，メンバ next は，plist 型の構造体をさすポインタで，次のように宣言される．

```
struct plist *next;
```

まとめると，構造体 plist は次のように定義される．

```
struct plist {
  double x;
  double y;
  struct plist *next;
  };
```

next は，ここで定義されている自分自身，つまりplist という型への参照であるので，このような構造体を自己参照構造体という．



3. malloc 関数による記憶領域の確保

struct 宣言は，前述のようにテンプレートを宣言しただけなので，データを格納するための記憶領域はまだ確保されて (割付けられて) いない．ここでは，プログラムの実行時において，キーボードから座標値を読み込むたびに，一つずつ頂点の構造体のためのデータ領域を生成し，それをポインタでつないでいく．

実行時に，メモリの中にデータ領域を確保するには，malloc 関数を使う．malloc 関数は，必要なメモリ領域の大きさを渡すと，システムのメモリ中から適当な領域を確保し，その先頭アドレスをポインタとして返してくる．

1 個の plist 構造体のデータを記憶するのに必要なメモリー領域の大きさは，
sizeof(struct plist) によって計算される．したがって，malloc(sizeof(struct plist)) とすると，malloc 関数はメモリー領域にsizeof(struct plist) だけの領域を確保し，その領域へのポインターを返してくる．

ただし，malloc 関数は，その領域がどのような型のデータとして使われるかには関与せず，void 型へのポインタとしてポインタを返す．ここでは得られた領域をstruct plist 型として使うので，このポインタを，struct plist 型へのポインタ型，すなわち，
(struct plist *) 型に型変換してから使う．型変換はキャストによって行なう．そのためには，この型名(struct plist *) をキャスト演算子として用いて次の様にする．

```
p = (struct plist *)malloc(sizeof(struct plist));
```

このようにすると， plist 構造体の領域が一つ生成され，変数p は，その領域へのポインタとなるので，例えば，

```
p->x = 1.2;
p->y = 2.2;
```

のようにして座標値を格納することができる．

以上で説明した部分を使ったプログラムを図 80に示す．

補足

(a) malloc 関数を使用する時には，#include <stdlib.h> を忘れてはならない．

(b) 計算機のメモリーが不足し，malloc が領域を確保できないときには，NULL という値を返す．図 80のプログラムはこのチェックを行なっていないので，暴走する可能性がある．次のようにするのが望ましい．

```
 1  /* list.c List structure for polygon vertexes */

 2  #include <stdio.h>
 3  #include <stdlib.h>

 4  struct plist {
 5    double x;
 6    double y;
 7    struct plist *next;
 8    };

 9  void main(void)
10  {
11    struct plist *p;
12    double xpos, ypos;

13    printf("Enter X and Y: ");
14    scanf("%lf %lf", &xpos, &ypos);

15    p = (struct plist *)malloc(sizeof(struct plist));
16    p->x = xpos;
17    p->y = ypos;

18    printf("(%lf %lf)\n", p->x, p->y);
19  }
```

図 80: list.c

```
p = (struct plist *)malloc(sizeof(struct plist));
if(p == NULL){
  printf("Memory allocation error!\n");
  exit(-1);
  };
```

ここで， exit(-1); はプログラムを異常終了させるものである．

(c) calloc: malloc と同様に記憶領域を割付ける関数で，配列を対象としたものにcallocがある．calloc は，配列の要素の個数と要素のサイズを与えて，その配列のための記憶領域を確保する．

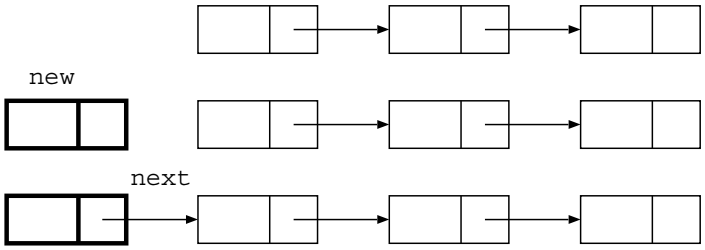
```
int *ip;
ip = (int *)calloc( 3, sizeof( int ));
*ip = 1; *(ip+1)=2; ip[2] = 3;
```

これによって， int 型の要素 3 個を持つ配列 ip が生成され，
ip[0]=1; ip[1]=2; ip[2]=3; となる．

1. リストの生成

リストの要素の生成の仕方が分かったところで，その要素をポインタでつないでリストを構成してゆく方法について示そう．

リストを生成する基本的な考え方は，新しい要素を生成しては，それまでのリストをそれにつなぐというものである．つまり，新しい要素はリストの先頭 (一番最初の要素) の前に挿入される．



リストの先頭を start で表すと，まず，新しい要素を生成し，その要素の next メンバに，start をセットする．

この新しい要素はmalloc によって生成し，それをポインタ new にセットする．リストを構成する操作の核は以下のようなものになる．

```
start = NULL;

while(scanf("%lf %lf", &xpos, &ypos) != EOF ){
  new = (struct plist *)malloc(sizeof(struct plist));
  new->x = xpos;  new->y = ypos;
  new->next = start;
  start = new;
}
```

図 81にアルゴリズムの様子を示す．

(a) `start = NULL;`
最初に start = NULL と初期化する． NULL はヌルポインタといい，” どこも指さないポインタ” を表す．後述のようにリストの終りを示すのに使うポインタ値であり，start = NULL は start を空のリスト (要素のないリスト) に初期化するという意味である．

(b) `new = (struct plist *)malloc(sizeof(struct plist));`
ここからは while による繰り返しで，座標値を読み込み，malloc で新しい要素 new を生成し， new->x = xpos; new->y = ypos; で読み込んだ座標値をセットする．

(c) `new->next = start;`
次に new を先頭の要素にするために， new の next に現在のリストの先頭へのポインタ start をセットする．

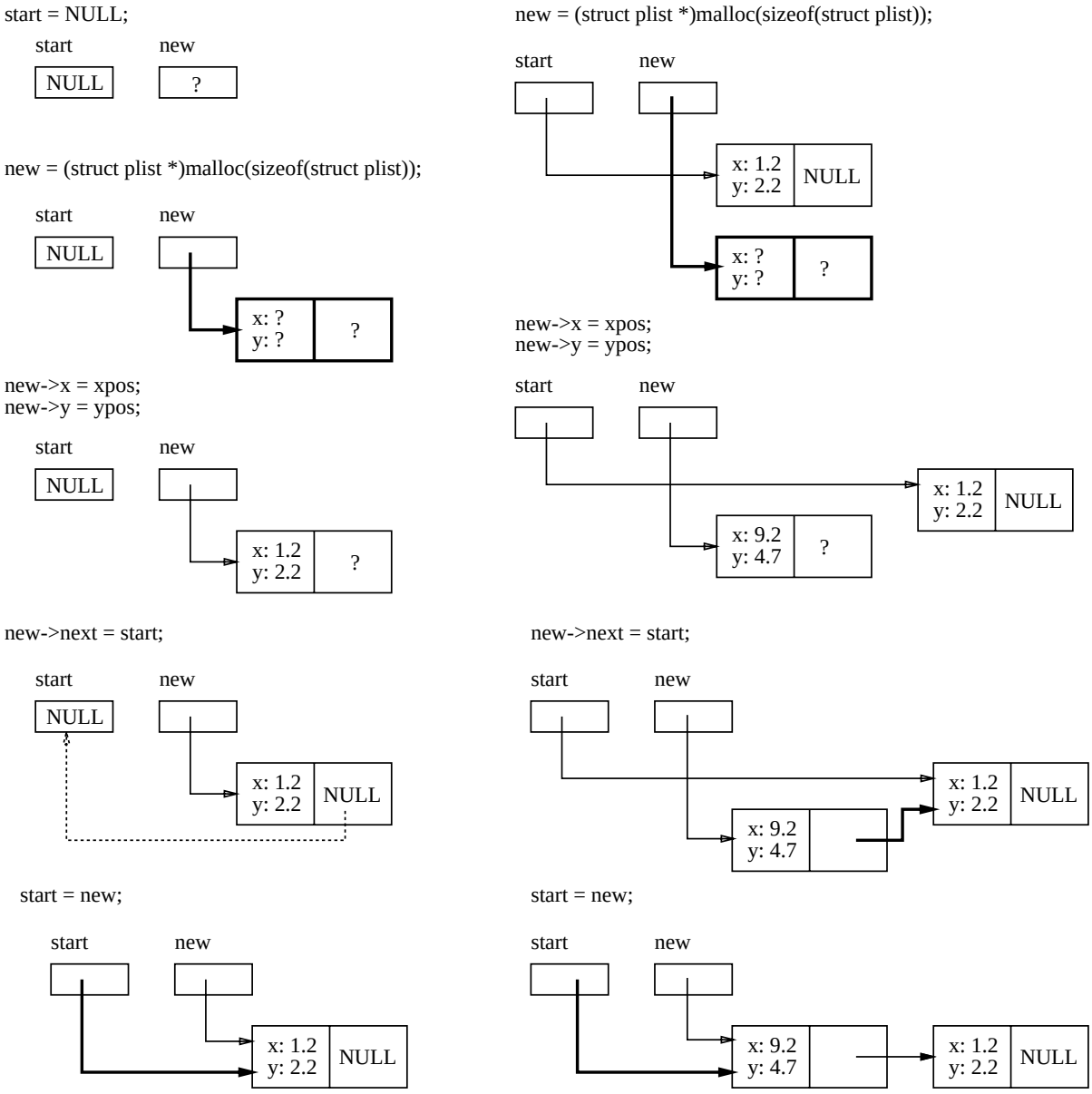


図 81: リストの生成

(d) `start = new;`
`new` が先頭要素になったので，`start` を `new` にする．

以上を繰り返す．最初に挿入された要素の前に一つずつ要素が追加されるので，入力されたのと逆の順序でリストが生成される．最初に挿入された要素が，リストの最後の要素となり，その `next` は `NULL` になっていることに注意せよ．これによってリストの終端を示す．

2. リストのなぞり
生成されたリストを先頭要素から表示することを考える．先頭要素へのポインタは，`start` で得られる．これを変数 `p` に代入すれば，`p->x`，`p->y` によって，先頭要素の座標値は得られる．

2 番目の要素は `p->next` で得られるので，`p = p->next;` として `p` を更新し，処理を繰り返す．最後の要素では，この値が `NULL` になるので，`p` が `NULL` でない間処理を繰り返す．

while を使ったプログラムは次のようになる.

```
p = start;
while( p != NULL ) {
  printf("(%lf %lf)\n", p->x, p->y);
  p = p->next;
  }
```

3. 以上をまとめたプログラムを図 82に示す.

37 typedef

1. typedef によって，プロトタイプなどの型に名前をつけることができる．例えば,
   typedef char[] STRING;
   これは，char[] 型 (文字列へのポインタ) にSTRING という名前をつける．これを使うと，
   変数p を宣言するのに，char p[]; ではなくSTRING p; と書くことができる．
   STRING p; は char p[]; と同じ.

2. struct に対して名前をつけると，記述が簡潔になる．struct plist に対しては，次のように書ける.

```
typedef struct plist * PLIST_PTR;
typedef struct plist {
  double x;
  double y;
  PLSIT_PTR next;
  } PLIST;
```

   これによって，これまで struct plist と書いてきたところをPLIST で,
   またstruct plist * と書いてきたところを，PLIST_PTR と書くことができる.

3. 再帰関数の利用
   自己参照データ型は，自分自身を参照している再帰的なデータ型なので，これを処理するには再帰関数を利用するのが便利である．例えば，上記のように要素を表示するには，次の再帰関数が使える.

   関数 printlist はリスト p をもらって，その先頭要素を表示し，さらに先頭要素を除いたリスト p を引数として自分自身を呼びだす． p が NULL(空リスト) の時には，再帰呼び出しを打ち切る.

```
void printlist(PLIST_PTR p)
{
  if(p != NULL) {
    printf("(%lf %lf)\n", p->x, p->y);
```

```
1  /* poly1.c  List structure for polygon vertexes */

2  #include <stdio.h>
3  #include <stdlib.h>

4  struct plist {
5    double x;
6    double y;
7    struct plist *next;
8    };

9  void main(void)
10 {
11   struct plist *start, *new, *p;
12   double xpos, ypos;

13   start = NULL;
14   printf("Enter X and Y: ");
15   while(scanf("%lf %lf", &xpos, &ypos) != EOF ){
16     new = (struct plist *)malloc(sizeof(struct plist));
17     if(new == NULL){
18       printf("Memory allocation error!\n");
19       exit(-1);
20     };

21     new->x = xpos;
22     new->y = ypos;
23     new->next = start;
24     start = new;

25     printf("Enter X and Y: ");
26   }

27   p = start;
28   while( p != NULL ) {
29     printf("(%lf %lf)\n", p->x, p->y);
30     p = p->next;
31   }
32 }
```

図 82: poly1.c

```
        printlist(p->next);
```

```
> gcc plist2.c
> a.out
Enter X and Y: 1.2 2.2
Enter X and Y: 9.2 4.7
Enter X and Y: 3.3 6.5
Enter X and Y: ^D             (Control-D)
(3.300000 6.500000)
(9.200000 4.700000)
(1.200000 2.200000)
>
```

図 83: 実行例

```
  }
}
```

いまリストを<p1,p2,p3> とすると,

```
    printlist(start=<p1,p2,p3>) の呼び出し
       |printlist p=<p1, p2, p3>
           |p1 を print
           |printlist(p.next=<p2, p3>) の呼び出し
               |printlist p=<p2, p3>
                   |p2 を print
                   |printlist(p.next=<p3>) の呼び出し
                       |printlist p=<p3>
                           |p3 を print
                           |printlist(p.next=<>)  NULL リストで呼び出し
                               |printlist p=<>
                                   |return
                           |return
                       |return
                   |return
               |return
           |return
        |return
```

プログラムは, 図 85のようになる.

```
 1  /* poly2.c  List structure for polygon vertexes */

 2  #include <stdio.h>
 3  #include <stdlib.h>

 4  typedef struct plist * PLIST_PTR;
 5  typedef struct plist {
 6    double x;
 7    double y;
 8    PLIST_PTR next;
 9    } PLIST;

10  void main(void)
11  {
12    PLIST_PTR new, start, p;
13    double xpos, ypos;

14    start = NULL;
15    printf("Enter X and Y: ");
16    while(scanf("%lf %lf", &xpos, &ypos) != EOF ){
17      new = (PLIST_PTR)malloc(sizeof(PLIST));
18      if(new == NULL){
19        printf("Memory allocation error!\n");
20        exit(-1);
21      };

22      new->x = xpos;
23      new->y = ypos;
24      new->next = start;
25      start = new;

26      printf("Enter X and Y: ");
27    }

28    p = start;
29    while( p != NULL ) {
30      printf("(%lf %lf)\n", p->x, p->y);
31      p = p->next;
32    }
33  }
```

図 84: poly2.c

```
1  /* List structure for polygon vertexes */

2  #include <stdio.h>
3  #include <stdlib.h>

4  typedef struct plist * PLIST_PTR;
5  typedef struct plist {
6    double x;
7    double y;
8    PLIST_PTR next;
9    } PLIST;

10 void printlist(PLIST_PTR);

11 void main(void)
12 {
13   PLIST_PTR new, start, p;
14   double xpos, ypos;

15   start = NULL;
16   printf("Enter X and Y: ");
17   while(scanf("%lf %lf", &xpos, &ypos) != EOF ){
18     new = (PLIST_PTR)malloc(sizeof(PLIST));
19     if(new == NULL){
20       printf("Memory allocation error!\n");
21       exit(-1);
22     };

23     new->x = xpos;
24     new->y = ypos;
25     new->next = start;
26     start = new;

27     printf("Enter X and Y: ");
28   }

29   printlist(start);
30 }

31 void printlist(PLIST_PTR p)
32 {
33   if(p != NULL) {
34     printf("(%lf %lf)\n", p->x, p->y);
35     printlist(p->next);
36   }
37 }
```

図 85: poly3.c

早見表

定数

1234	10 進定数
0137	先頭に 0 は，8 進定数
0x37FC	先頭に 0x は，16 進定数
9.5e-2　91.5	倍精度浮動少数点の定数
'm'	文字定数
L'x'	wide 文字定数
"hello"	文字列定数．ヌル文字で区切り．
L"hello"	wide 文字列定数

定数の接尾辞

接尾辞なしの 10 進数定数	int, long int, unsigned long int
接尾辞なしの 8/16 進数定数	int, unsigned int, long int, unsigned long int
f　F	float
l　L	long double
l　L	long int, unsigned long int
u　U	unsigned int, unsigned long int
ul　UL	unsigned long int

特殊文字，エスケープシーケンス

\a	ベル	\b	バックスペース
\f	改行	\n	復帰改行
\r	復帰	\t	水平タブ
\v	垂直タブ	\'	シングルクォート
\"	ダブルクォート	\?	疑問符
\\	バックスラッシュ	*ddd*	8 進数による指定
\x*dd*	16 進数による指定	\0	ヌル文字

文

`/* comment */`	コメントは/* と*/ で括る.
`jim = 5;`	代入文.
`;`	空文.
`{tmp=a; a=b; b=tmp;}`	複文. {と} とで括る.
`break;`	break 文. 一番内側の while, do, for, switch を終了.
`continue;`	continue 文. while, do, for ループを続ける (ループの末尾へ).

`do`
` c=getchar();`
`while(c == ' ');`
	do-while 文. while の条件が真の間繰り返す.

`for(i=0; i<MAX; i++)` ` a[i]=0;`	for 文. 最初に初期化を行ない, 次に文 (a[i]=0;) を実行し, 条件が真の間 (i<MAX), 更新 (i++) を行ないながら, 文の実行を繰り返す.
`goto error;` `error: printf("ABORT\n");`	goto 文. 無条件にラベルの文にジャンプする.
`if (a<0) a = -a;` `else printf("was plus\n");`	if 文. 条件が真ならば後続の文を実行する. else 部がある時は, 条件が偽の時実行.
`return x;`	return 文. 関数から戻る. 呼び出し側に値を戻す.

`switch(getchar()){`
` case 'X': exit(0);`
` case 'H': help();`
` break;`
`case 'A': case 'B':`
` a++; break;`
`default:`
` printf("try again\n");`
`}`
	式 (getchar()) を評価し, その値と一致する case 文にジャンプする. (exit(0) はプログラムを終了させる.)
	式の値が'H' の時に実行される部分. もし, ここに break がないと, 次の case の文も実行される.
	'A' または'B' の時はここが実行される.
	どの case ともマッチしない時は default の文が実行される.
	switch 文の終了.

`while(i<MAX) a[i++]=0`	条件が真の間, 文を実行する.

変数

`char alpha;`	1 バイト整数．文字.
`char msg[]="HELP";`	配列の初期化． (5 文字分の配列)
`char *ptr;`	文字型の値を指すポインタ変数
`const char w='*';`	定数．代入できない.
`double big;`	倍精度浮動少数点.
`enum color {red, yellow=2, blue};`	列挙型`enum color` の定義．`red` は 0，`yellow` は 2，`blue` は 3 の値を持つ.
`enum color good=blue;`	列挙型`enum color` の変数 good の宣言．初期値は `blue`.
`float raft[10][20];`	2 次元配列.
`int i,j,k;`	(符合付きの) 整数.
`long int jumbo;`	(符合付き) 整数． `int` よりも大きい.
`long double huge;`	最大の浮動少数点.
`short x,y;`	(符合付きの) 整数． `char` より大きく， `int` より小さい.

`struct emp{`
` char fname[15];`
` unsigned sex:1;`
`} person;`
	構造体`struct emp` の定義.
	メンバー fname の定義.
	ビットフィールドのメンバー
	この構造体型の変数 person の宣言.

`typedef char *string;`	string という名前の新しいデータ型の定義.

`union kludge{`
` char c;`
` float f;`
`} mixed={'x'};`
	共用体の宣言.
	二つのメンバーが同じメモリを占める.
	
	初期化

`unsigned limit=0x3FF;`	符合無し整数の宣言．初期化
`void end(void);`	値を戻さない関数 end の宣言.
`void *ptr;`	汎用ポインタ

記憶クラス

`auto`	自動変数．関数の中で定義され，呼び出された時に生成され，呼び出しが終了すると消滅する.
`extern`	外部 (別ファイル) で定義されていることを示す．プログラム実行中保持される.
`register`	レジスタ変数．速いアクセスの必要な変数に使う．アドレスは取れない.
`static`	静的変数．関数の実行終了後も値が保持される.

演算子と優先順序

[]	→	配列要素	a[1]
()	→	関数呼び出し	f(x)
.	→	構造体 (共用体) のメンバ	p1.fname
->	→	構造体 (共用体) へのポインタ	p->x
++	→	後置インクリメント演算子	i++
--	→	後置デクリメント演算子	i++
*	←	間接参照演算子	i--
&	←	アドレス演算子	&x
+ -	←	単項の＋－演算子	+1 -2
!	←	論理否定	!flag
~	←	ビット毎の否定	~b
++	←	前置インクリメント演算子	++i
--	←	前置デクリメント演算子	--i
sizeof	←	オブジェクトの大きさ	sizeof (int) sizeof x
(*type*)	←	キャスト演算子	(int)x
*	→	乗算	2*x
/	→	除算	x/2
%	→	剰余	i%2
+ -	→	和と差演算	1+2 2-1
>>	→	右シフト	b << 1
<<	→	左シフト	b >> 4
<	→	< 大小比較	x < 0
>	→	>	x > 0
<=	→	≤	x >= 0
>=	→	≥	x <= 0
==	→	等しい.	x == 0
!=	→	等しくない.	x != 0
&	→	ビット毎の AND	b&\x0F
^	→	ビット毎の XOR	b^\x0F
|	→	ビット毎の OR	b|\x0F
&&	→	論理積	(a>0)&&(b<0)
||	→	論理和	(a>0)||(b<0)
? :	←	条件演算子	(x>0) ? x:-x
= *= /= %= += -= ^= |= &= <<= >>=	←	代入演算子	x *= 2

入出力関数

`int fclose(FILE *stream)`	出力バッファをフラッシュし，ファイルをクローズ.
`int fgetc(FILE *stream)`	ストリームから文字を入力．エラーや end of file ならEOF を戻す.
`FILE *fopen(const char *fname, const char *mode)`	ファイルのオープン.
mode	
r	データ読み込み．ファイルが存在しない時はNULL を戻す.
w	データ書き込み．ファイルが存在する時は，その内容を捨てる．またファイルが存在しない時は作成される.
a	ファイルの終端からデータ追加．ファイルポインタは終端に設定される.
`fputc(int c, FILE *stream)`	ストリームへの文字の出力.
`putc(int c, FILE *stream)`	fputc と同じ．ただし，マクロで定義されていることがある.
`fgetc(FILE *stream)`	ストリームからの文字の入力.
`getc(FILE *stream)`	fgetc と同じ．ただし，マクロで定義されていることがある.
`getchar(void)`	標準入出力からの文字の入力.
`char *gets(char *str)`	標準入出力からの文字列の入力.
`putchar(int c)`	標準入出力への文字の出力.
`puts(const char *str)`	標準入出力への文字列の出力.

書式付き入出力関数

`const char *format;`	変換仕様は文字列
`int scanf(format, addr1, addr2,...)`	入力を変換仕様に従って変換．変換 / 代入に成功した個数を戻す. あるいは入力が end of file の時にはEOF を戻す．引数は全てポインタ.
`int printf(format, expr1, expr2,...)`	出力した文字数を戻す．エラーの時は負の値を戻す.
`int fscanf(stream, format, addr1, addr2,...)`	ファイル入力
`int fprintf(stream, format, expr1, expr2,...)`	ファイル出力
`int sscanf(string, format, addr1, addr2,...)`	バッファ (文字列) からの入力
`int sprintf(string,format, expr1, expr2,...)`	バッファへの出力

書式制御文字列

%<f><w><.p><m><c>	変換仕様の一般形.
<f>	フラッグ
-	左詰め
+	符合付き
␣ 空白	最初の文字が符合ならば符合を付ける.
#	0 を埋める.
<w>	最大幅
<.p>	精度
<m>	修飾子
h	short int
l	long int
l	double
L	long double
<c>	変換指定
d	符合付き 10 進整数. scanf では 16 進, 8 進も.
i	符合付き 10 進整数
u	符合無し 10 進整数
x,X	符合無し 16 進整数
o	符合無し 8 進整数
c	文字
s	文字列へのポインタ
e,E	指数表示
f	浮動少数点
g,G	指数表示と浮動少数点表示を適宜使用
p	ポインタ (アドレス)
n	整数へのポインタで, それまでに printf が出力した文字数を得る.

前処理命令 / マクロ命令

`#define TRUE 1`	識別子 TRUE を 1 で置き換える.
`#define NEG(x) (-(x))`	マクロ NEG(x) を (-(x)) に展開. x は引数.
`#undef DEBUG`	マクロ定義を無効にする.
`#if MODE == 1`	式が真ならば, その次の行から, #elif, #else, #endif のいずれかの行まで取り込む.
`#ifdef TESTME`	識別子が定義されているかどうか調べる. 後は#if と同じ. #if defined(TESTME) でもよい.
`#ifndef TESTME`	識別子が定義されていないかどうか調べる. 後は#if と同じ. #if !defined(TESTME) でもよい.
`#elif defined (FLAG)`	条件を調べ, その後の行を取り込むかどうか決める. #if, #ifdef, #ifndef のいずれかと, #endif の間で使う.
`#else`	条件を調べ, その後の行を取り込むかどうか決める. #if, #ifdef, #ifndef のいずれかの後で, 任意個の#elif の後に使う. それらの全てが偽になったときに#else 以下の行から#endif までが取り込まれる.
`#endif`	#if, #elif, #else 参照.
`#include "local.h"`	この行を指定されたファイルで置き換える.
`#include <local.h>`	この行を指定されたシステムヘッダーファイルで置き換える.
`#include FAVORITE`	識別子FAVORITE の値をファイル名とするファイルで置き換える.
`#line 100 "somewhere"`	行番号とファイル名の付け直し.
`#pragma`	システム (コンパイラ) 固有の処理を指定.
`#`	空な命令
`##`	この命令の直前と直後のトークンを結合する.
`__LINE__`	行番号
`__FILE__`	ファイル名
`__DATE__`	コンパイルを開始した日付
`__TIME__`	コンパイルを開始した時刻
`__STDC__`	ANSI C のコンパイラの時に 1