第１回プログラミング入門

（教科書１～3章）

1

本演習履修にあたって

教科書：「C言語によるプログラミング入門」
吉村賢治著、昭晃堂

参考書：「プログラミング言語C」
カーニハン、リッチー著、共立出版

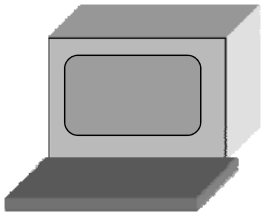
サポートページ：
http://www.ec.h.akita-pu.ac.jp/~programming/

2

プログラミング演習の目的

コンピュータを用いた問題解決ができるようになる。

そのために、プログラムの作成能力を身に付ける。



3

今回の目標

- 演習の遂行に必要なツールの使用方法を習得する。
- 課題の提出法を習得する。
- 現実の問題をプログラムにするまでの概要を理解する。

☆演習室から課題を提出する。

4

演習で利用する Linux上プログラミング環境（1）

- GNOME 端末
  コンピュータをキーボードを使って操作する
- Emacs テキストエディタ
  プログラムを記述し、保存する
- Subversion バージョン管理システム
  記述したファイルを他の人と共有したり、
  過去の内容を調べたりする
  （本演習では課題の提出に利用）

5

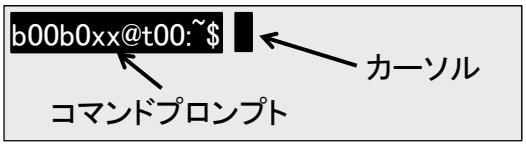
演習で利用する Linux上プログラミング環境（2）

以下のツールは次回の演習で説明する

- GCC（ GNU C コンパイラ ）
  C言語で記述されたプログラムを
  コンピュータが実行できる形式（機械語）
  に変換する
- Make
  GCCなどを自動的に起動する

6

端末とコマンド



- コマンドプロンプトが出ている時に、コマンド(命令)をキーボードを使って入力
- カーソル位置に文字が入力される
- 矢印キーなどを使って編集できる

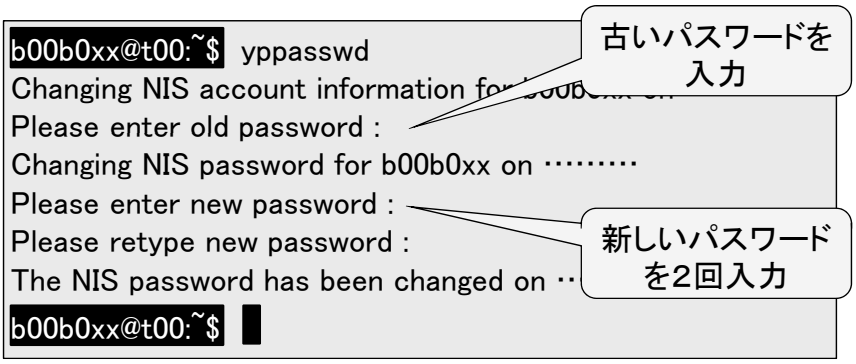
7

コマンドの実行

- 多くのコマンドはコマンドの引数を必要とする(スペースで区切って入力)
- コマンドを入力後、Enterキーで実行
- ウィンドウを開くコマンド(Emacsなど)を実行する際には、最後に「&」を付けること

8

パスワードの変更



注意:
パスワード入力時は何も表示されないので
(「***」も表示されない)、慎重に入力すること

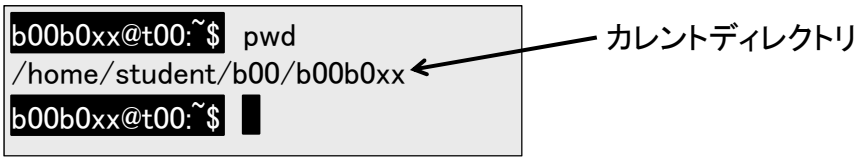
9

パスワードの付け方

- 英文字の大文字・小文字・記号・数字を必ず混ぜて使うこと
- 6文字以上とすること
- ユーザ名(学籍番号)と同一の文字列を含んではならない
- 氏名、生年月日、車のナンバー、電話番号、誕生日などを含んでいてはならない

10

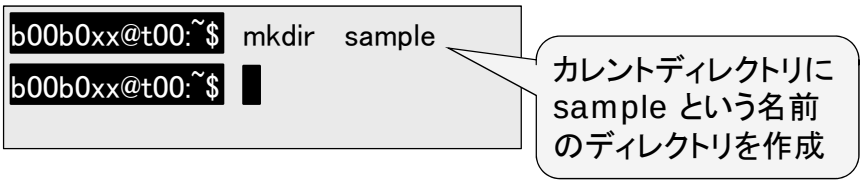
カレントディレクトリ



- カレントディレクトリ：今いるディレクトリ
- 特に指定しない限り、多くのコマンドは
  カレントディレクトリにあるファイルを操作
- pwd コマンドでカレントディレクトリを表示
- ウィンドウ毎に異なるので注意

11

ディレクトリの作成



- mkdir コマンドで新しいディレクトリを作成
- mkdir コマンドの引数に指定した名前の
  ディレクトリを、カレントディレクトリの中に
  作成する

12

ディレクトリ内容の表示

```
b00b0xx@t00:~$ ls
Desktop/        sample/
b00b0xx@t00:~$
```

カレントディレクトリに Desktop と sample という名前のディレクトリが存在

- ls コマンドでカレントディレクトリにあるファイルやディレクトリなどの一覧を表示
- ファイルやディレクトリの種類を色や記号で区別
- 作業前後に実行する癖を付けておくと良い

13

カレントディレクトリの変更

```
b00b0xx@t00:~$ cd sample
b00b0xx@t00:~/sample$ pwd
/home/student/b00/b00b0xx/sample
b00b0xx@t00:~/sample$
```

カレントディレクトリが確かに変更されている

- cd コマンドの引数に指定したディレクトリにカレントディレクトリを変更
- カレントディレクトリの変更に伴い、コマンドプロンプトの表示も変わる

14

cd コマンドの特別な使い方

```
b00b0xx@t00:~/sample/test$ pwd
/home/student/b00/b00b0xx/sample/test
b00b0xx@t00:~/sample/test$ cd ..
b00b0xx@t00:~/sample$ pwd
/home/student/b00/b00b0xx/sample
```

cd .. で「ひとつ上のディレクトリ」に移動

```
b00b0xx@t00:~/sample/test$ pwd
/home/student/b00/b00b0xx/sample/test
b00b0xx@t00:~/sample/test$ cd
b00b0xx@t00:~$ pwd
/home/student/b00/b00b0xx
```

引数を付けずに cd を実行すると、いつでもホームディレクトリへ移動

15

その他の有用なコマンド（一例）

- lv ： ファイルの内容を表示
- cp ：ファイルのコピー
- mv ：他のディレクトリへのファイルの移動、ファイル名の変更
- rm ：ファイルの消去
- rmdir ：ディレクトリの消去
- man ：コマンド使用法（マニュアル）の表示

16

プログラミングにおけるバージョン管理

- 複数の開発者による共同作業で
  プログラムを作成できる
- 過去のファイルの内容を調べることが
  できる（以前の内容に戻せる）

17

作業ディレクトリの作成

```
b00b0xx@t00:~$ svn checkout http://...../svn/b00b0xx/prog
b00b0xx@t00:~$ ls
Desktop/      prog/      sample/
b00b0xx@t00:~$ cd prog
b00b0xx@t00:~/prog$ ls
01/  03/  05/  07/  09/  11/  13/  S2/
02/  04/  06/  08/  10/  12/  S1/
b00b0xx@t00:~/prog$ cd 01
b00b0xx@t00:~/prog/01$ ls
comment.c
```

作業ディレクトリと、
その中身が、
カレントディレクトリに
自動的に作られる

- 本演習で使うリポジトリのアドレスは
  http://dav.ec.h.akita-pu.ac.jp/svn/ユーザ名/prog

18

毎日の準備

```
b00b0xx@t00:~$ ls
Desktop/      prog/      sample/
b00b0xx@t00:~$ cd prog
b00b0xx@t00:~/prog$ svn update
リビジョン 1 です。
b00b0xx@t00:~/prog$ cd 01
b00b0xx@t00:~/prog/01$ ls
comment.c
```

作業ディレクトリの中身が最新の状態に更新される

- その日の作業を始める前に
  作業ディレクトリで svn update を実行

19

Emacs の起動（既存ファイルの編集）

```
b00b0xx@t00:~/prog/01$ ls
comment.c
b00b0xx@t00:~/prog/01$ emacs comment.c &
b00b0xx@t00:~/prog/01$ ls
comment.c
```

comment.cの内容を編集できる

- コマンドの引数として、ファイル名を指定する（各種プログラミング支援機能が利用できるようになる）
- 最後に「&」を付けること

20

Emacs の起動（新規ファイルの作成）

```
b00b0xx@t00:~/prog/01$ ls
comment.c
b00b0xx@t00:~/prog/01$ emacs  comment2.c  &
b00b0xx@t00:~/prog/01$ ls
comment.c  comment2.c
```

カレントディレクトリに comment2.c が自動的に作成される

- コマンドの引数として、ファイル名を指定する（各種プログラミング支援機能が利用できるようになる）
- 最後に「&」を付けること

21

課題の提出

```
b00b0xx@t00:~/prog/01$ ls
comment.c  comment2.c
b00b0xx@t00:~/prog/01$ svn  add  comment2.c
A  comment2.c
```

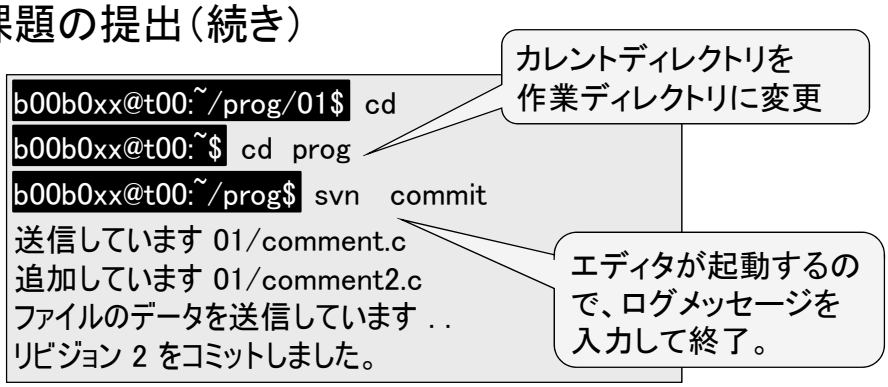
提出すべきファイルが正しいディレクトリにあるか必ず確認

提出するファイルであることを示すマークを付ける

- 新しく作ったファイルを提出したいときは、svn add コマンドでマークを付ける
- すでにマークが付いているときは、そのままで良い

22

課題の提出(続き)



- svn commit コマンドで、マークが付いていて変更があったファイルを全て提出
- ログメッセージには、何を解決したかをわかりやすく書く(作業報告)
- 失敗したら再度 svn update してから

23

提出の確認

- svn log -r '{締切日}'
  で、指定した締切日より前に、最後に提出した提出日時とログメッセージを確認
- svn log
  で、全てのログメッセージを確認

24

提出内容の確認

```
b00b0xx@t00:~/prog$ cd 01
b00b0xx@t00:~/prog/01$ svn cat -r '{2009-04-10}' comment.c
/*
  作成日:2009年4月09日
  作成者:本荘 てまり
  学籍番号:b00b0xx
...
```

- svn cat -r '{締切日}' ファイル名
  で、提出された時点でのファイルの内容
  (採点対象になる内容)を確認

25

ヒントの確認

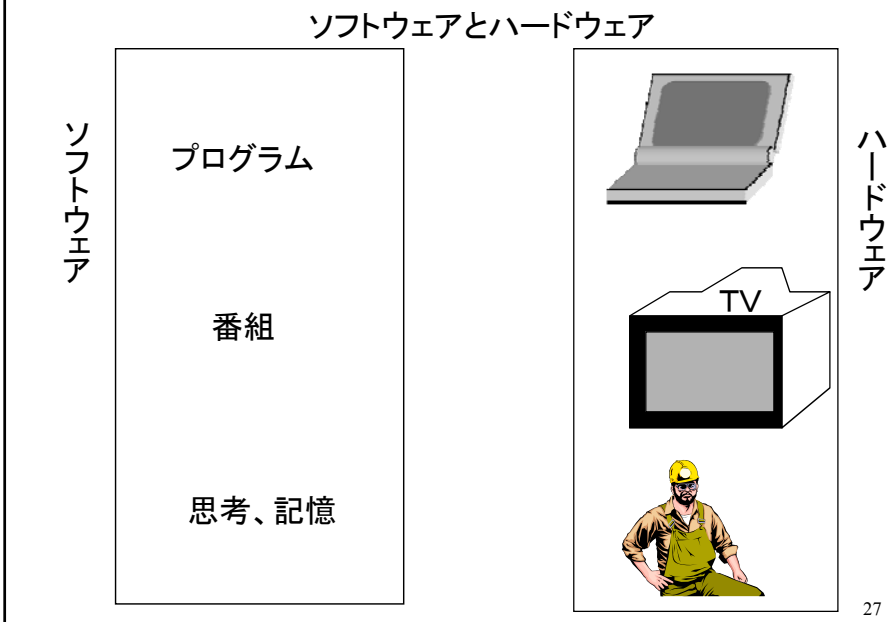
```
b00b0xx@t00:~/prog$ svn update
U  01/comment.c
A  01/README
リビジョン 3 に更新しました。
b00b0xx@t00:~/prog$ cd 01
b00b0xx@t00:~/prog/01$ lv README
◎ コメントを表す「/*」と「*/」は1バイト文字でなくてはいけません。
b00b0xx@t00:~/prog/01$ lv comment.c
/* TODO : 日付を最終更新日に直してください。*/
/*
...
```

svn update でリビジョンが更新されたときはヒントあり

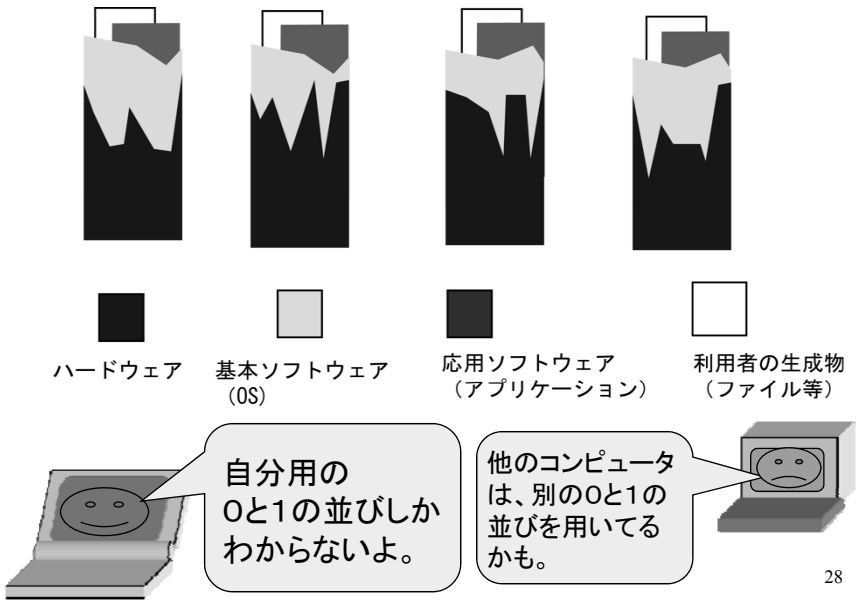
- READMEというファイルやソースファイル
  上に教員からのヒントが記述される

26

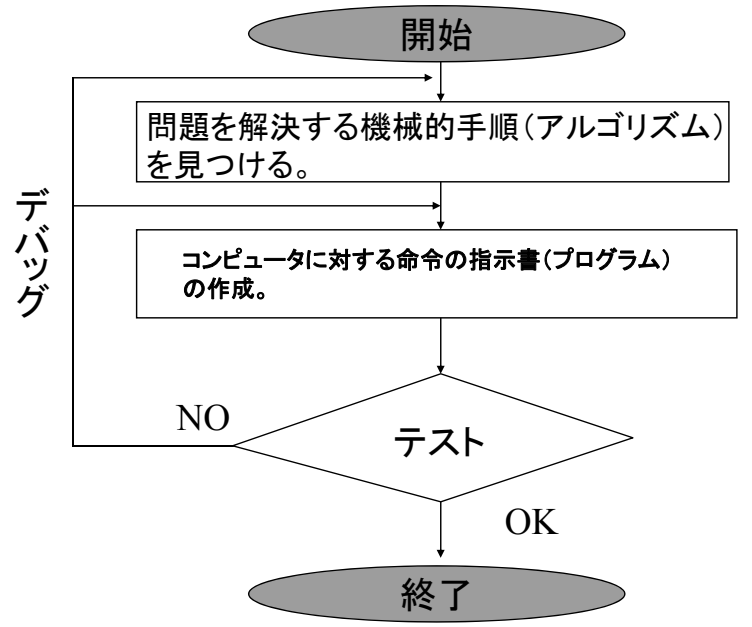
コンピュータの2つの側面



ハードウェアとソフトウェアの階層構造

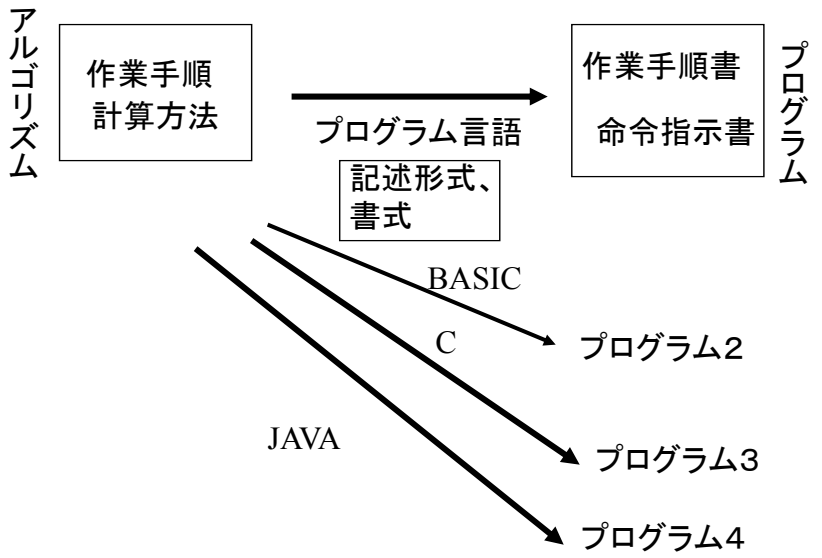


問題解決のためのソフトウェアの開発



29

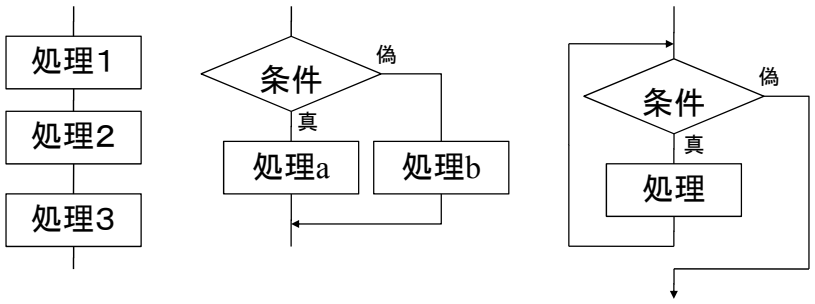
アルゴリズムとプログラム



30

アルゴリズムの基本要素

1. 決められた順番にいくつかの処理を行う （順次）
2. 条件判断により二つの処理のどちらかを行う （選択）
3. ある処理を繰り返し何度も行う （反復）

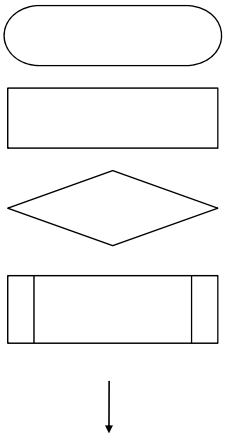


順次　　選択　　反復

31

フローチャートによる手順の記述

フローチャートの記号



端子（手順の開始・終了）

処理の基本単位

条件判断

他のフローチャートで
定義された手順

処理の流れ（通常は上から下へ）

32

処理の基本単位

- 1つの値（定数）を変数に代入
- 1つの変数と1つの定数の二項演算（＋，－，×，÷）の結果を変数に代入
- 2つの変数の二項演算の結果を変数に代入

$x \leftarrow 2.5$	$x \leftarrow x+1$	$x \leftarrow y+z$
変数 x に 2.5を代入	変数 x の値をそれまでより1増やす	変数 x に $y+z$ を計算した値を代入

33

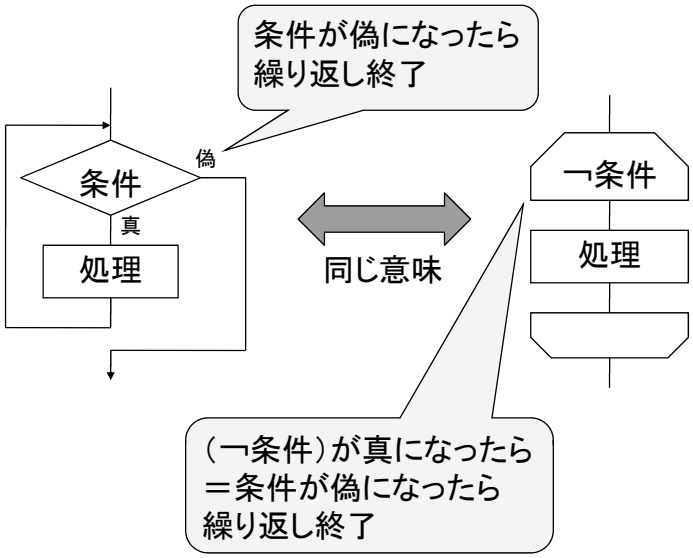
条件判断

- 1つの値（定数）と変数の一致・大小比較
- 2つの変数の値の一致・大小比較
- 上記の論理式（かつ「∧」、または「∨」、否定「¬」）による組み合わせ

$x<2.5$	$x \leqq y$	$0 \leqq x \wedge x<5$
変数 x の値が2.5未満ならば真	変数 y の値が x 以上ならば真	変数 x の値が0以上5未満（0～4）ならば真

34

反復処理の省略記法



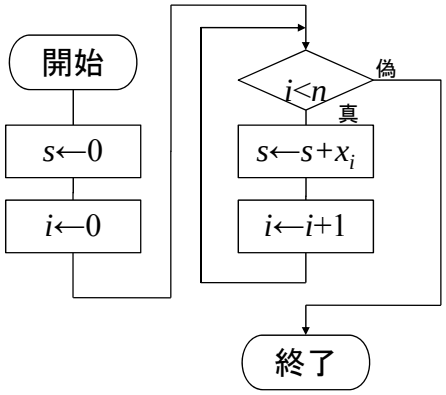
35

フローチャートの例

- n 個の要素からなる数列 $X=x_0,x_1,\ldots,x_{n-1}$ の要素の総和を求めるアルゴリズム

入力:
要素数n
数列要素 $x_0,x_1,\ldots,x_{n-1}$

出力:
総和 $s=\sum_{i=0}^{n-1} x_i$



36

正しいアルゴリズム

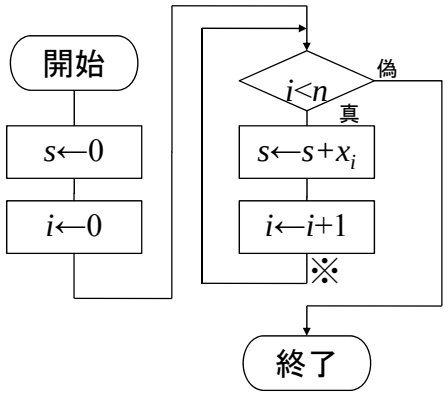
- アルゴリズムが正しいことを数学的に証明することができる

右図※の箇所で

$$s = \sum_{j=0}^{i-1} x_j$$

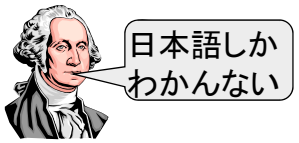
が常に真。

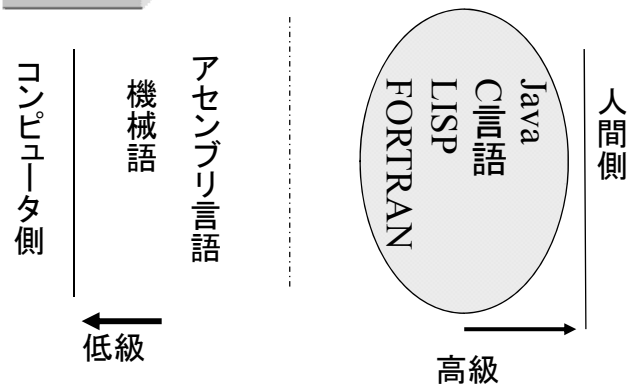
ループ不変条件：
数学的帰納法を使って
証明してみよう



37

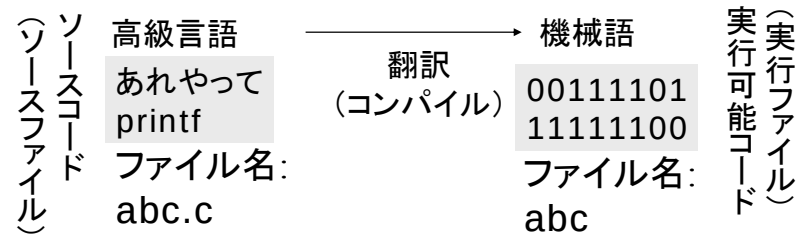
プログラミング言語の分類（高級言語と低級言語）





38

コンパイラ



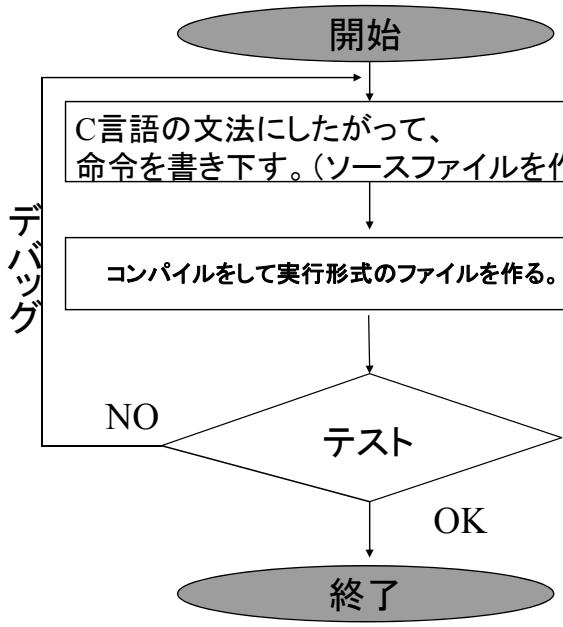
コンパイラ:

高級言語（例えばC言語）から、低級言語（機械語）へ翻訳（コンパイル）するソフトウェア。

注意：C言語では、ソースファイルは
*****.c
という名前にする。（拡張子が".c"のファイルにする。）

39

C言語でのプログラムの作り方



	使うもの	できるもの
C言語の文法にしたがって、命令を書き下す。（ソースファイルを作る。）	エディタ	ソース（abc.c）
コンパイルをして実行形式のファイルを作る。	コンパイラ	実行形式（abc）
テスト	シェル（OS）	いろいろ（プログラム次第）

40

良いソフトウェアの基準

- 速い

  同じことするなら速い方がいいでしょ。

- 強い

  どんな入力でもきちんと動作してほしいでしょ。

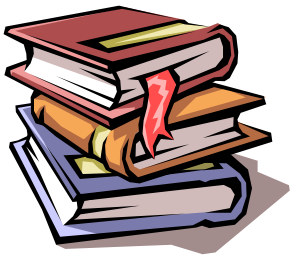
- 分かりやすい

  誰がみても理解しやすいほうがいいでしょ。

本演習では、独自のスタイル規則に沿って
プログラミングしてもらいます。(ガイダンス資料参照)

41

第2回C言語の基本的な規則



1

今回の目標

- C言語の基本的な規則を理解する。
- C言語のソースコードから実行可能なコードへの変換法を習得する。（コンパイル法の習得）
- 標準入出力を理解する。
- C言語での標準入出力制御方法を理解する。

☆標準入出力を用いたプログラムを作成する。

2

世界一短いCのプログラム

shortest.c

```
main(){}
```

このプログラムからわかること

[注意1]C言語のソースファイルは、mainという名前の関数が必要。

[注意2]括弧が重要（括弧の種類も含めて）

[注意3]いろんな部分を省略できる。

↓

実は、コンパイラ任せにしているだけ。
本演習では、省略してはいけない。
スタイル規則参照。

3

実行ファイルの作り方と実行

（実行ファイルの作り方その1、ガイダンス資料も参照のこと）

ソースファイルから実行ファイルを作ることを「コンパイル」といい、
コンパイルするためのプログラムを「コンパイラ」という。

本演習で用いるコンパイラ　gcc（GNU　C　Compiler）

コンパイラを手動で使う方法

```
gcc　ソースファイル名　　-o　実行ファイル名
```

```
$ gcc shortest.c　-o shortest
```

なお、「－o　実行ファイル名」を省略すると、
a.out　という名前の
実行ファイルが生成される。

プログラムを実行する方法

```
./実行ファイル名
```

```
$./shortest
```

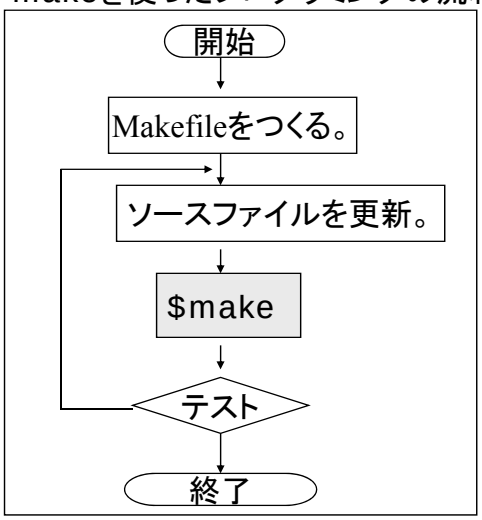
4

makeによる実行ファイルの作り方

（実行ファイルの作り方その2、ガイダンス参照も参照のこと）

make:コンパイラを自動で起動するコマンド

makeを使ったプログラミングの流れ



Makefile:コンパイルのやり方を記述するファイル。本演習ではほとんど同じファイルをずっと使えて便利。

この方法の利点。

・コンパイラへの指示が毎回同じである。
・デバッグ作業が楽にできる。
・間違って、ソースファイルを消す危険性が減る。

5

Makefile の記述

Makefieには、コンパイラへの指示がいろいろ記述できる。
本演習では、以下のようにかけば良い。

書式

```
CC  = gcc
all: 実行ファイル名（ソースファイル名から「.c」を除いたもの）
```

例

Makefile

```
CC = gcc
all:shortest
```

```
$ make
```

⇕ 同じ効果

```
$ gcc shortect.c  -o shortest
```

一回Makefileを書くだけで、デバッグごとの実行ファイルの作成が格段に楽になる。

（もう少し複雑なMakefileは第４回に説明する。）

6

本演習のスタイルによる最小のソースコード

（スタイル規則参照）

```
/*
        作成日: yyyy/mm/dd
        作成者:本荘　太郎
        学籍番号:B00B0xx
        ソースファイル:shortest.c
        実行ファイル:shortest
        説明:なにもしないプログラム
        入力:なし
        出力:なし
*/
int main()
{
  return  0;
}
```

練習1（挨拶表示プログラム）

```
/*
        作成日: yyyy/mm/dd
        作成者:本荘　太郎
        学籍番号:B00B0xx
        ソースファイル:hello.c
        実行ファイル:hello
        説明:あいさつを表示するプログラム
        入力:なし
        出力:標準出力に「hello,world」と出力する。
*/
#include <stdio.h>

int  main()
{
        printf("hello ,world ¥n");
        return  0;
}
```

とりあえず、
このように書く。

8

C言語のコメント

書式

```
/*     この間にコメントを書く  */
```

スラッシュ＋アスタリスク

アスタリスク＋スラッシュ
（順序に注意）

いろんなコメント

```
/*
課題 02-1
ename.c
*/

/*****************************/
/*          注目！！！！          */
/*          重要！！！！          */
/*****************************/
```

関数

C言語のプログラムは関数で構成される。

引数:　関数が受け取る入力（例えば実数値）
戻り値:　関数が出す出力（例えば実数値）

関数書式

一種の入力

```
戻り値の型    関数名（引数の型  引数）
{
   関数の本体
   return  戻り値;
}
```

一種の出力

詳しくは、9～11回で説明する。

10

main関数

プログラム実行時に（必ず）最初に実行される関数。

UNIX系のOSでは、main関数の戻り値型はint型にする。

main関数の戻り値型:
本演習では省略しない。

```
int  main()
{
    return  0;
}
```

OK

Unix系のOSでは、
main関数が0を出力することで
正常終了を意味する。

（スタイル規則参照）

11

プログラムの実行順序

書式を抽出。

```
int main()
{
        ＊＊＊＊＊＊
        ＊＊＊＊＊＊＊
        ＊＊＊＊
        ＊＊＊＊＊＊＊＊＊
        ＊＊＊＊＊＊
        return  0;

}
```

C言語のプログラムは、
main関数から始まり、
通常は、上から下に実行される。

12

インデント（字下げ）

インデント

人間がプログラムを読みやすくするための工夫。
中括弧の内部をすべて1タブ分あけてから書く。
(Emacs上では、Tabキーで揃えることができる。)

OK

```
#include <stdio.h>
int main()
{
    → printf(“プログラミング情報表示¥n");
    → printf(“作成者:本荘太郎¥n”);
    → printf(“内容:文字列を表示するための、”);
    → printf(“練習用コード¥n");
    → return 0;
}
```

1行には一つの命令文。
（長すぎるときは改行して見やすくする。）

13

悪いプログラム例1

NG

```
#include <stdio.h>
int main()
{
      printf(“hello,world¥n”)
      return  0;
}
```

「;」がないので間違い。
コンパイルできない

NG

```
#include <stdio.h>
int main()
{
      printf(“hello”);printf(“world¥n”);
      return  0;
}
```

1行にセミコロンが2個以上ある。
（スタイル規則違反）

14

悪いプログラム例2

```
#include <stdio.h>
int main()
{
        printf(“hello,world¥n”);
}
```

NG

return　文の省略
（スタイル規則違反）

```
#include <stdio.h>
int main()
{
        printf(“hello”);
        printf(“world¥n”);
return  0;
}
```

NG

インデント不備
（スタイル規則違反）

15

標準出力

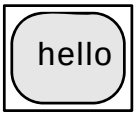
注意：標準出力は
関数の出力（戻り値）とは
無関係。

プログラムを普通に実行したとき：
標準出力＝ディスプレイ（端末）

```
$ ./実行ファイル名
```

```
$./hello
Hello,world!
$
```

モニタ

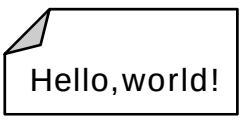


リダイレクション「＞」を使って実行したとき：
標準出力＝ファイル

```
$ ./実行ファイル名　＞　ファイル名
```

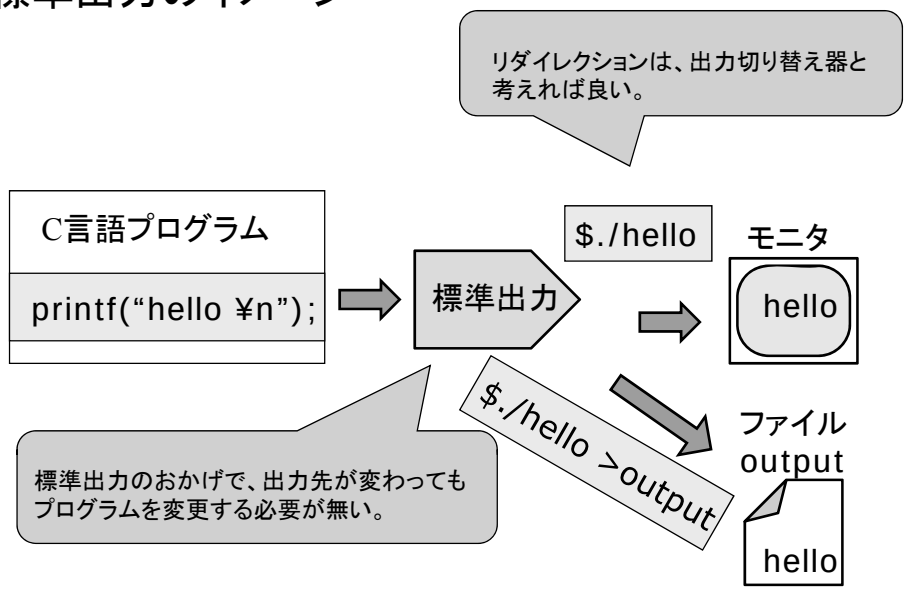
```
$./hello　　>　　output
$
```

ファイル output



16

標準出力のイメージ



17

エスケープ文字

ソース内で¥（バックスラッシュ）で始まる文字列（2文字）は、
コンピュータの内部では一つの記号を表す。
このような文字をエスケープ文字という。

エスケープ文字列集

¥n	改行
¥t	タブ
¥b	バックスペース
¥0	終端文字

¥¥	バックスラッシュ
¥’	シングルクォーテーション
¥“	ダブルクォーテーション

¥（バックスラッシュ）で始まる文字列は特別な意味を持つ。
と考えても良い。

なお、この資料では、「¥」でバックスラッシュを表わす。
UNIX上では、「 ＼ 」がバックスラッシュを表す。

18

1バイト文字の表現（ASCIIコード）

1バイト文字（半角文字）を数字で指定できる。

¥x＋16進数

‘A’＝¥x41

上位桁

下位桁	2	3	4	5	6	7
0		0	@	P	`	p
1	!	1	A	Q	a	q
2	“	2	B	R	b	r
3	#	3	C	S	c	s
4	$	4	D	T	d	t
5	%	5	E	U	e	u
6	&	6	F	V	f	v
7	‘	7	G	W	g	w
8	(	8	H	X	h	x
9	)	9	I	Y	i	y
A	*	:	J	Z	j	z
B	+	;	K	[	k	{
C	,	<	L	¥	l	|
D	-	=	M	]	m	}
E	.	>	N	^	n	~
F	/	?	O	_	o	

19

2バイト文字の扱い

2バイト文字（全角文字）を用いてもよいのは，
- コメント内
- printf 文の「”」と「”」で囲まれた内部

のみ。
それ以外では使うことはできない。

注意：
特に、**全角スペースを書かないように気を付けること。**
正しくコンパイルできない。

20

プリプロセッサへの指示

#で始まる行はプリプロセッサに指示を与える。

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

当面は、プログラム先頭に必ず記入すること。

OK

```
#include <stdio.h>
int main()
{
        printf("hello,world¥n");
        return  0;
}
```

NG

```
int main()
{
        printf("hello,world¥n");
        return  0;
}
```

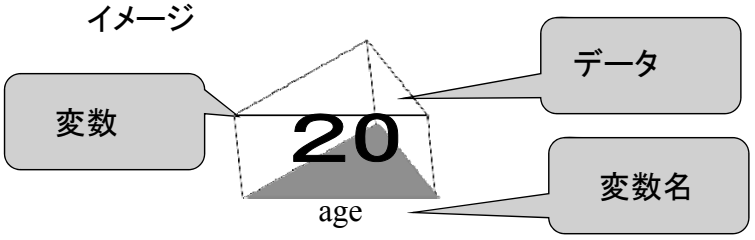
21

変数

変数：データを入れる入れ物

変数名：変数の名前
　　規則にのっとった文字列で命名する。

イメージ



22

数学とC言語の変数の違い

数学		C言語
入れられるデータ		
主に数で、 整数、実数の区別をしない。	⟺	主に数、文字で、 種類毎に区別する。 整数と実数も区別する。 （型という考え方）
変数名		
慣用的に決まっており、 x, y, t 等の１文字が多い。	⟺	長い名前を 自分で命名できる。

23

命名規則

[規則]　名前（変数名、関数名等）は英字、数字あるいは
_(アンダースコア）だけからなり先頭は数字以外の文字である。

（スタイル規則参照）

変数名の例

age	x_coordinate
i	y_coordinate
j	x1
k	y1

- なるべく意味のある文字列にする事。
- main内で宣言する変数は英**小文字**、数字、_（アンダースコア）だけを用い英大文字は用いない事。

24

予約語（キーワード）

auto	break	case	char
continue	default	do	double
else	for	goto	if
int	long	register	return
short	sizeof	static	struct
switch	typedef	union	unsigned
void	while		

C言語の予約語は以上である。

なお、printfとかscanfとかは、ライブラリ中で定義されている。

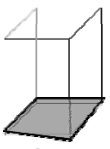
注意：変数名や関数名に予約語を用いてはいけない。
（予約語以外で、命名する。）

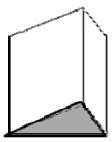
25

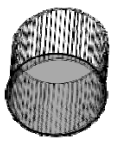
C言語の代表的なデータ型

データは、種類ごとに異なる扱いをしなければならない。
種類は、型として区別される。

char	1バイトの整数型（1つの文字を表す型）
int	整数型
float	単精度浮動小数点型
double	倍精度浮動小数点型


char


int


double

26

C言語のデータ型が扱える範囲

型	バイト長	表現範囲
char	1	0~255
int	4	-214748648~21483647
short int	2	-32768~32767
unsigned int	4	0~4294967295
float	4	$\pm 1.0*10^{-37}$ ~ $\pm 1.0*10^{38}$
double	8	$\pm$ $1.0*10^{-307}$ ~ $\pm 1.0*10^{308}$

27

数学の概念とC言語の型

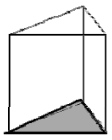
自然数 ⟷ unsigned int

整数 ⟷ int

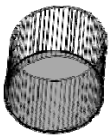
実数 ⟷ float / double

1文字 ⟷ char

文字列 ⟷ char *



int

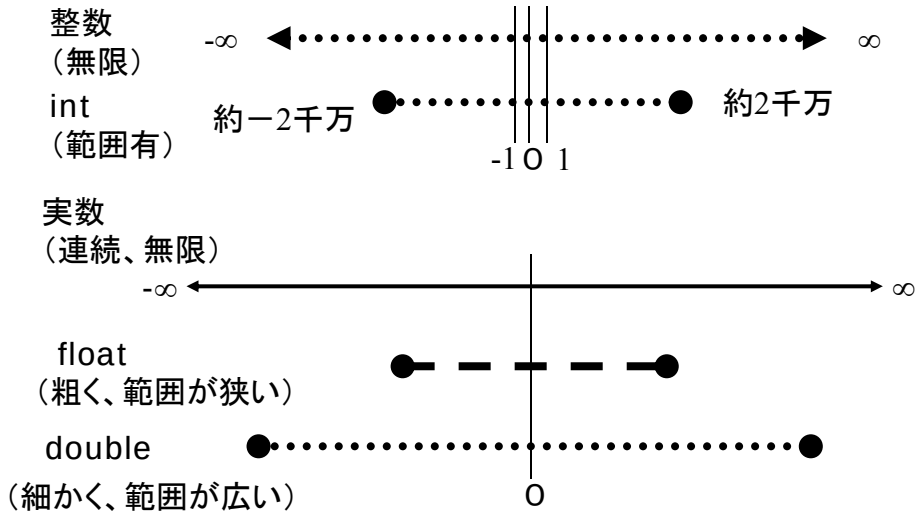


double

本演習で主に用いる型。
離散量→int型
連続量→double型

28

数学とC言語の型の違い



29

本演習で用いる変数の型

（スタイル規則参照）

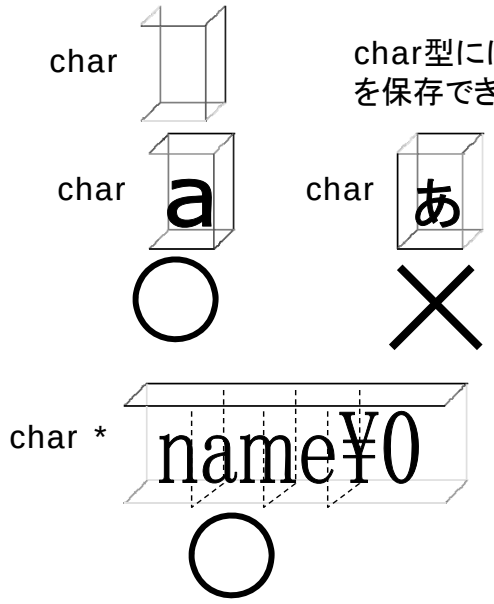
整数（離散量）　int　年齢、日数、等

実数（連続量）　double　体重、温度、等

明確に区別すること。

30

文字と文字列



31

変数宣言

Cでは使用する変数をすべて宣言しなければならない。

変数宣言書式

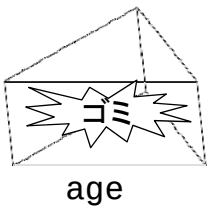
```
データ型1　　変数名1;
データ型2　　変数名2;
```

変数宣言例1

```
int　age;　　/*　　年齢　　*/
```

変数宣言は、プログラム内で用いるデータ
を入れる入れ物（変数）を用意して、
その入れ物に名前をつけると考えればよい。

宣言の際には、必ずコメントを付けること。
（スタイル規則参照）



age

32

変数宣言例2

```
double   x_pos;   /*   x座標      */
double   y_pos;   /*   y座標      */
```

変数宣言例3

```
int   age;          /*    年齢      */
double   x_pos;   /*   x座標      */
double   y_pos;   /*   y座標      */
```

33

変数宣言の場所

変数宣言は、関数の最初でまとめて行う。

典型的なmain関数

OK

```
int    main()
{
       /*変数宣言*/
       int     age;                /*    年齢    */
       double      x_pos;      /*    x座標   */
       double      y_pos;      /*    y座標   */

       ......
}
```

```
int    main()
{
       /*変数宣言1*/
       int  age;    /*   年齢     */

       /*変数宣言以外の処理*/
       f1=0;
       ・・・・・

       /*変数宣言2*/
       double  x_pos;    /* x座標     */
}
```

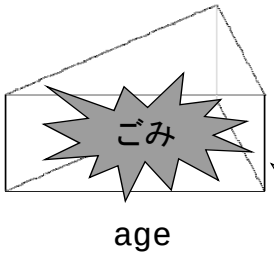
NG

環境（コンパイラ）によっては、エラーにしない場合がある。
本演習ではできるだけ多くの環境で動作するプログラム作成を目指す。
（コンパイルエラーが無いプログラムが正しいとは限らない。）

35

宣言直後の変数の中身

変数は宣言しただけでは、変数内のデータは不定です。
つまり、プログラムの実行時によって異なります。



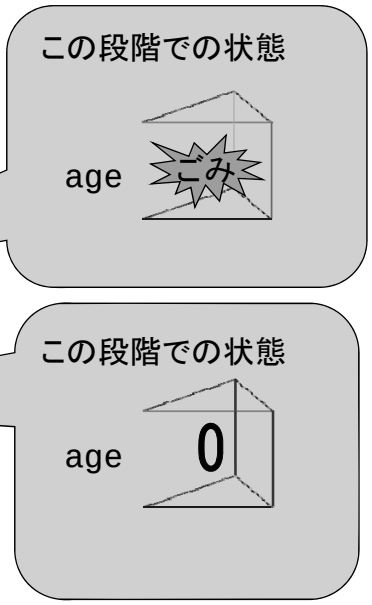
変数の中身を常に把握しておくことは、とても重要。
アルゴリズムに従って、適切な値を代入してから変数を用いることが多い。
このような代入を変数の初期化という。

36

変数の初期化

典型的なmain関数

```
int main()
{
        /*変数宣言*/
        int age;    /*年齢*/

        /*変数の初期化*/
        age=0;

        /*   ....*/
        .......
```



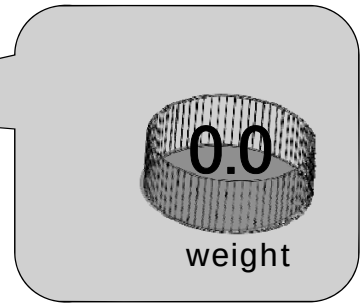
「＝」は、ソースコード内で、
変数に値を代入する方法（代入演算子）。
詳しくは次回。

37

変数の初期化と型

```
int main()
{
        /*変数宣言*/
        double weight;  /*体重*/

        /* 変数の初期化*/
        weight=0.0;

        /*   ....*/
        .......
```

定数にも型があるので、
注意すること。
詳しくは、次回。
（スタイル規則参照）



38

変数の中身の表示

printf文を用いると、標準出力に変数の中身を出力できる。

- 「"」と「"」の 間の文字列の中に**特別な文字列**を記述
- カンマの後に変数名

変換仕様 ：「％変換文字」

printfの典型的な使い方

```
printf("・・・・％変換文字・・・・", 変数名);
```

39

printf文と変換仕様

```
printf("You are %d years old¥n", age);
```

文字列を標準出力（ディスプレイ）に出力するライブラリ関数。

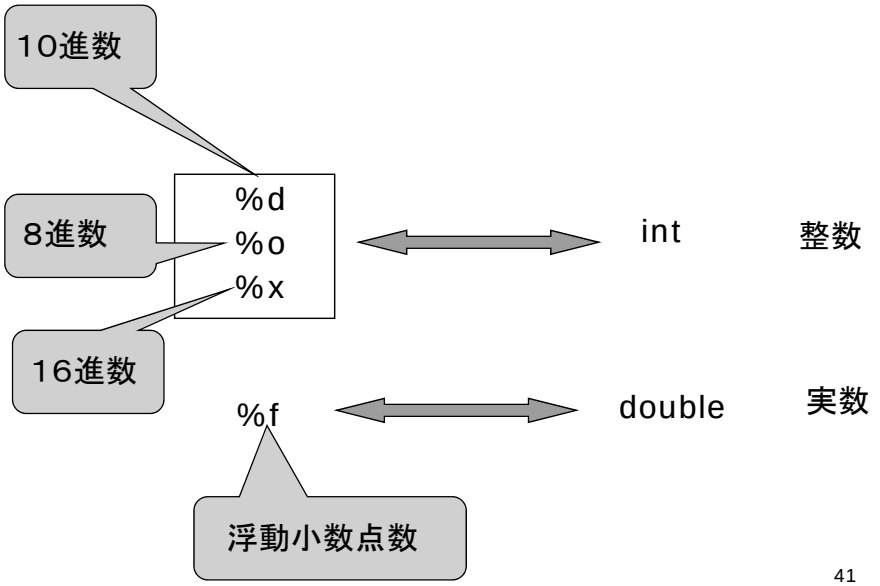
変換仕様　printf文の文字列内の「**％変換文字**」
後ろの変数に関する出力指示を表わす。

（int用）
%d　10進数の整数として表示
%6d　10進数として印字、少なくとも6文字幅で表示
%o　8進数の整数として表示
%x　16進数の整数として表示

（double用）
%f　小数（double）として表示
%6.2f　表示幅として6文字分とり、小数点以下2桁まで表示
%.2f　小数点以下2桁で表示

40

printf文における変換仕様と型の対応



41

printf文における複数の変換仕様

```
int    a1;
int    a2;
int    a3;
a1=1;
a2=2;
a3=3;
printf("first %d  second %d   third %d¥n ",a1,a2,a3);
```

```
int    a;
a=25;
printf("10進数%d 8進数 %o    16進数%x¥n ",a,a,a);
```

同じ変数を
何度も表示

42

printf文に関するよくある間違い

NG

```
int    age;
double      weight;

age=19;
weight=63.5;

printf(“My age is %d ¥n"age);
printf("The weight is %d¥n", weight);
```

カンマ忘れ

（変換仕様と変数の）
型の不一致

43

練習2（変数表示実験）

```
/*  変数の中身表示実験    print_var.c コメント省略*/
#include<stdio.h>
int  main()
{
        int      a;
        printf("代入前 a=%d¥n",a);

        a=0;
        printf("代入後 a=%d¥n",a);

        a=3;
        printf("正しい変換仕様a=%d¥n",a);
        printf("変換仕様間違いa=%f¥n",a);

        return  0;
}
```

44

標準入力から変数への値の読み込み

scanf文を用いて、標準入力（キーボード）から
変数に値を代入できる。

- 「”」と「”」の間に**特別な文字列だけ**記述
- カンマの後に「**＆変数名**」

変換仕様　:「％変換文字」

scanfの典型的な使い方

```
scanf("％変換文字", ＆変数名);
```

45

scanf文

```
scanf(“%d ”, &age);
```

標準入力（キーボード）から変数に値を読み込むライブラリ関数

変換仕様　scanf文の文字列内の「**％変換文字**」
scanfの「”」と「”」の間には**変換仕様しか書かないこと。**

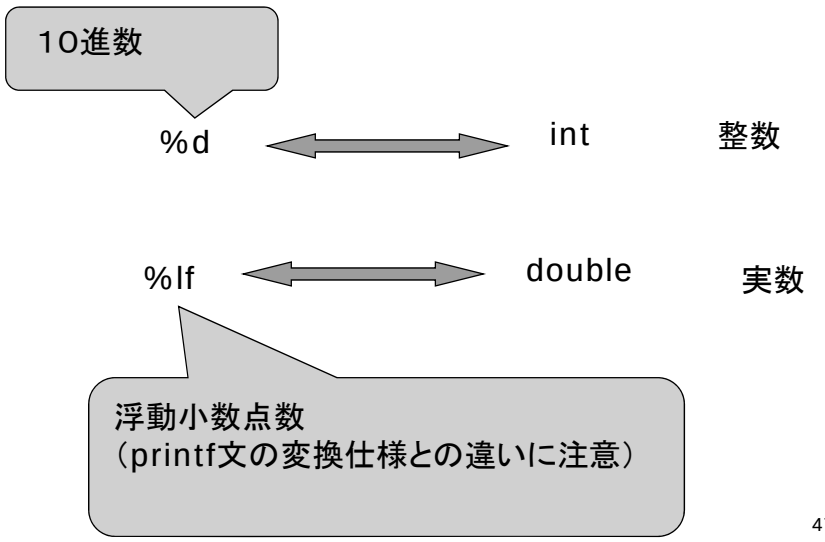
＆変数　　scanf文の変数名には&をつけること。
&は変数のアドレスを表わす。
詳しくは、13回ポインタで説明する。

%d　整数を入力
%lf　小数を入力（double型）

printfの変換文字との違いに注意！

46

scanf文における変換仕様と型の対応



47

scanf文における複数の変換文字

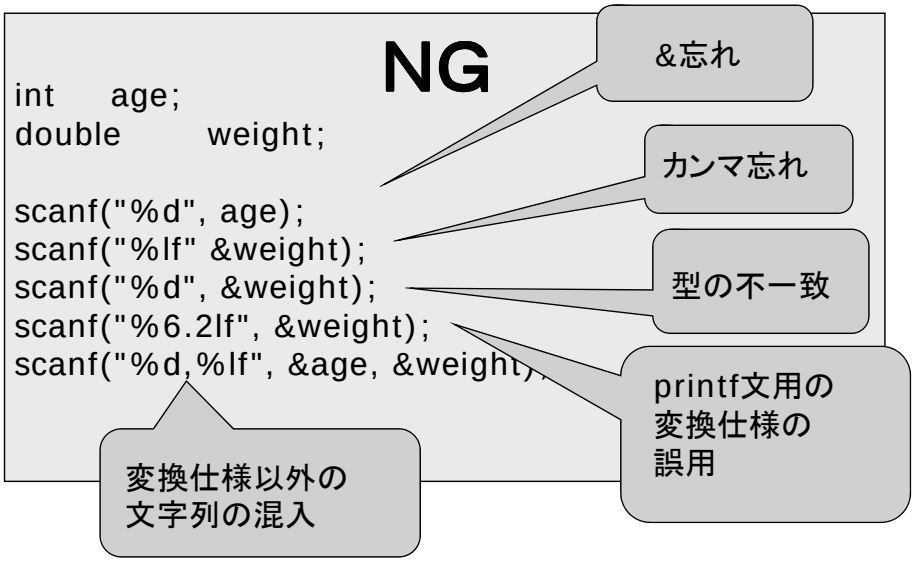
```
char   initial;
int    age;
double  weight;

scanf("%d%lf", &age, &weight);
```

同じ効果

```
/*同じ宣言*/
scanf("%d", &age);
scanf("%lf", &weight);
```

48

scanf文に関するよくある間違い



49

練習3（標準入出力実験）

```
/*標準入出力実験  test_stdio.c */
#include<stdio.h>
int main()
{
        int a;
        int b;
        int c;

        scanf("%d%d%d",&a,&b,&c);
        printf("a=%d  b=%d  c=%d ¥n",a,b,c);

        return 0;
}
```

50

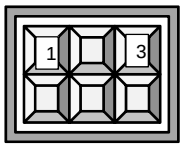
標準入力

注意：標準入力は
関数の入力（引数）とは
無関係。

プログラムを普通に実行したとき：
標準入力＝キーボード（端末）

```
$ ./実行ファイル名
```

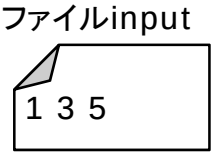
```
$./test_stdio
1 3 5
a=1 b=3 c=5
$
```



リダイレクション「＜」を使って実行したとき：
標準入力＝ファイル

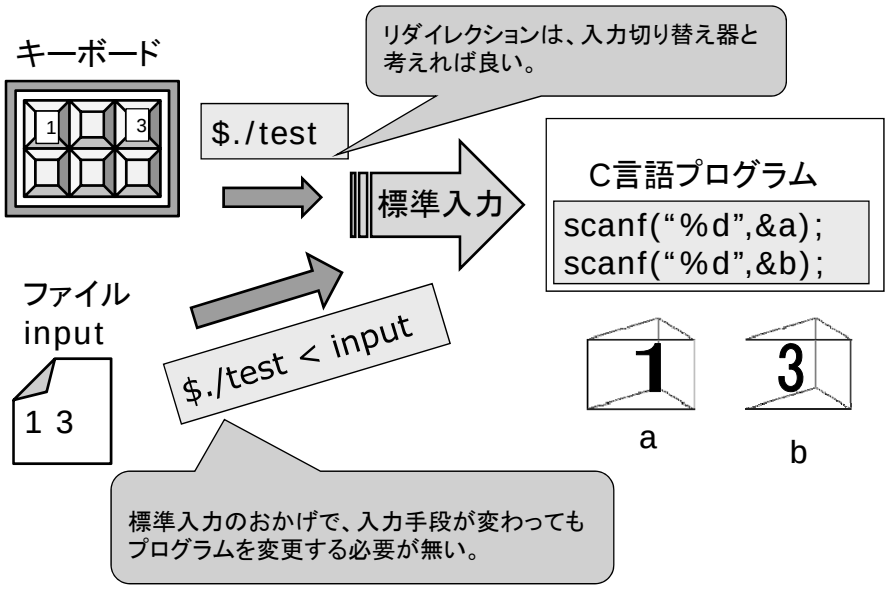
```
$ ./実行ファイル名　＜　入力ファイル名
```

```
$./test_stdio < test_stdio.in
a=1 b=3 c=5
$
```



51

標準入力のイメージ



52

標準入出力

UNIXのコマンドは、通常、標準入力と標準出力を用いる。

標準入力：通常はキーボードだが、
リダイレクション’<’を用いて
ファイルに変更可能。

標準出力：通常は画面だが、
リダイレクション’>’を用いて
ファイルに変更可能。

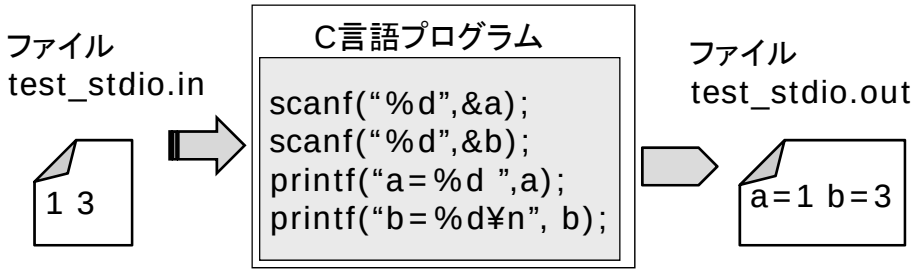
53

標準入出力の同時変更

リダイレクションを組み合わせればいい。

$./実行ファイル名　< 入力ファイル　>　出力ファイル

```
$./test_stdio  <test_stdio.in  > test_stdio.out
$lv test_stdio.out
a=1 b=3
$
```



54

標準入出力での年齢入出力プログラム

```
/*
        作成日:yyyy/mm/dd
        作成者:本荘　太郎
        学籍番号:B00B0xx
        ソースファイル:echoage.c
        実行ファイル:echoage
        説明:入力された年齢を表示するプログラム
        入力:標準入力から年齢(0以上の整数値)を受け取る。
        出力:標準出力に入力された年齢を出力する。
*/

/*      プログラム本体は次のページ        */
```

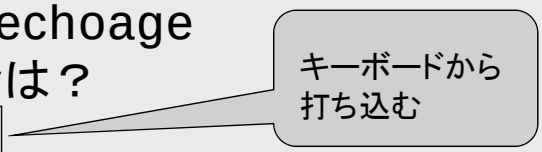
55

```
/*      前ページのプログラムの続き        */
#include <stdio.h>
int  main()
{
        /*      変数宣言                */
        int     age;        /*入力された年齢*/

        /*      年齢の入力     */
        printf("年齢は？¥n");
        scanf("%d", &age);

        /*      入力された値をそのまま出力する     */
        printf("年齢は %d  歳です。¥n", age);
        return  0;
}
```

56

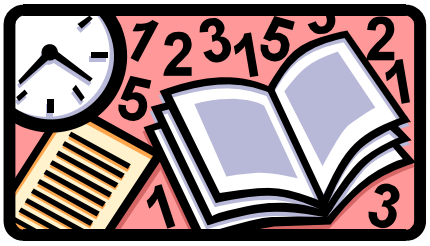
実行結果

```
$ make
gcc  echoage  -o echoage
$ ./echoage
年齢は？
20
年齢は 20  歳です。
$
```



57

第3回簡単な計算・プリプロセッサ



1

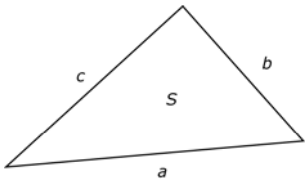
今回の目標

- 定数とその型を理解する。
- 演算子とその効果を理解する。
- 簡単なライブラリ関数の使用法を理解する。

☆ヘロンの公式を用いて
3角形の面積を求めるプログラムを
作成する。

2

ヘロンの公式



ヘロンの公式:
3辺の長さがわかっているときに、
3角形の面積を求める方法。

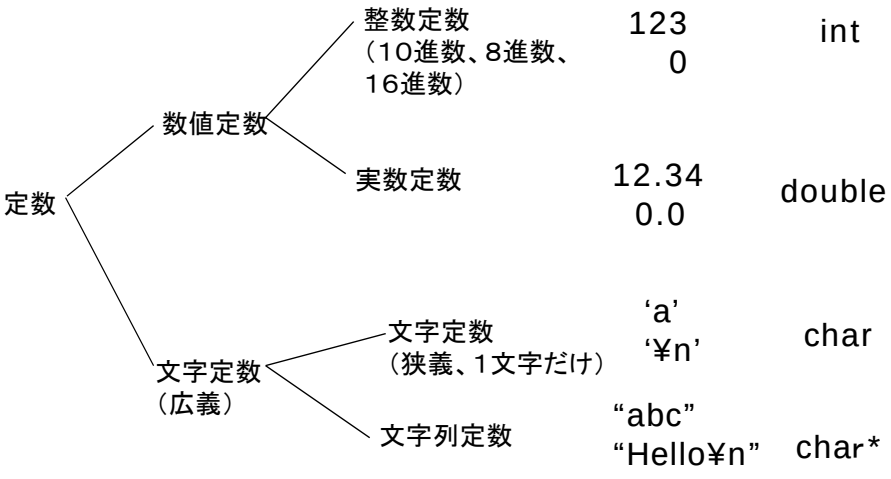
$$d = \frac{a + b + c}{2}$$

$$S = \sqrt{d\,(d - a)(d - b)(d - c)}$$

3

定数の分類と型

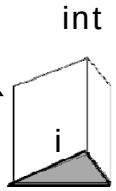
定数:プログラム中で、常に特定の値をもつ。



4

プログラム（ソース）内での整数定数の表現

0以外で始まる数字だけの列	10進数	123
0で始まる数字だけの列	8進数	0173
	（10進数	123）
0xで始まり、残りが数字だけの列	16進数	0x7B
	（10進数	123）



次の3つは、コンピュータ内ではまったく同じ処理を行う。

```
int     i;
i=123;
```

```
int     i;
i=0173;
```

```
int     i;
i=0x7B;
```

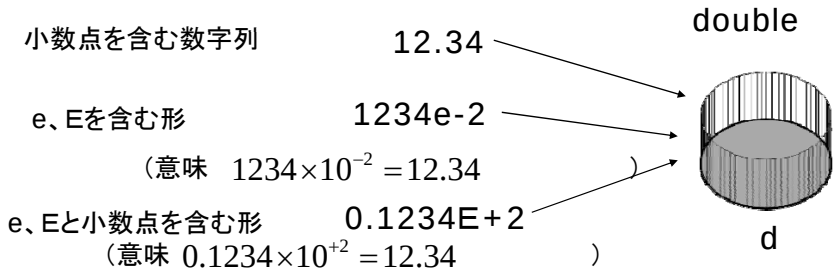
5

練習1（整数定数実験）

```
/*     整数定数実験  int_const.c  コメント省略*/
#include <stdio.h>
int  main()
{
       int     i;

       i=123;
       printf("%d¥n",i);
       i=0173;
       printf("%d¥n",i);
       i=0x7B;
       printf("%d¥n",i);

       return 0;
}
```

6

プログラム（ソース）内での実数定数の表現



次の3つは、コンピュータ内ではまったく同じ処理を行う。

```
double d;
d=12.34;
```

```
double d;
d=1234e-2;
```

```
double d;
d=0.1234E+2;
```

7

練習2（double定数実験）

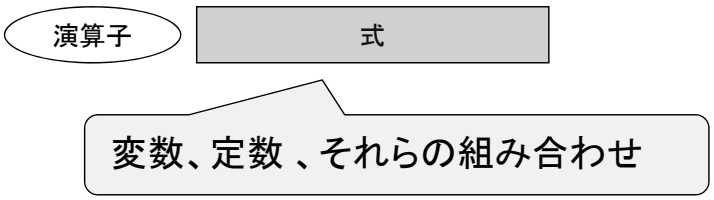
```
/*double定数実験  double_const.c  コメント省略    */
#include <stdio.h>
int  main()
{
      double      d;

      d=12.34;
      printf("%6.2f¥n",d);
      d=1234e-2;
      printf("%6.2f¥n",d);
      d=0.1234E+2;
      printf("%6.2f¥n",d);

      return 0;
}
```

8

演算子

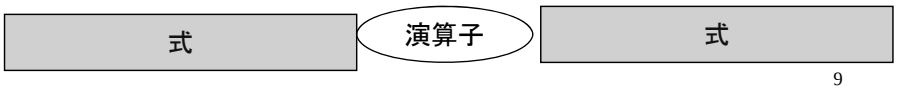
C言語では、演算子と式（変数、定数等）を組み合わせて、
プログラムが記述される。

単項演算子の書き方（前置型）



単項演算子の書き方（後置型）

二項演算子の書き方



9

算術演算子（C言語での算術計算）

	記号	例	意味
単項演算子	-	-a	aの値と-1の積
2項演算子（四則演算とモジュロ演算）	+	a+b	aの値とbの値の和
	-	a-b	〃 差
	*	a*b	〃 積
	/	a/b	〃 商
	%	a%b	aの値をbの値で割ったときの余り

10

数学との表記の違い（1）

	数学	C言語
掛け算	$a \times b$	`edge1*edge2`
	$a \bullet b$	
	ab（記号を省略）	`edge1edge2`

間違い
（演算子省略不可）
この記述だと、長い変数名だと思われる。

11

数学との表記の違い（2）

	数学	C言語
指数（べき乗）	a^2	`edge1*edge1`
	$a \wedge 2$	`pow(edge1,2.0)`

数学ライブラリの関数 pow を利用して
実数のべき乗を計算できる

12

結合規則と細則

[規則] 整数どうしの割り算では、商の小数部分は**切り捨てられる**。

$$\frac{\text{整数}}{\text{整数}} \rightarrow \text{整数}$$

int/int → int

```
int  i;
int  j;
double  x;
i  = 7;
j  = 2;
x =  i / j;
```

上のようなプログラムでは、
x の値は **3.0**である。

```
double      taiseki;
double      takasa;
double      teimen;

taiseki = 1/3 * takasa * teimen;
```

上のようなプログラムでは、
takasa と teimen の値に関わらず、
taiseki の値は **0.0** である。

13

[規則]演算子%は、double型には適用できない。

```
a=b%c;
```

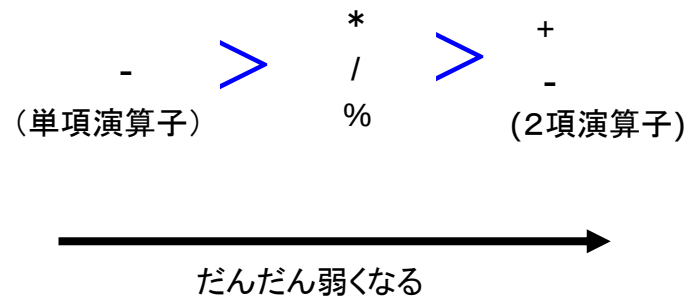
余りを求める演算。

このような例では、a,b,cすべてが整数（int型）でなくてはならない。

例
```
x=7/2;
y=7%2;
```
7÷2は、商が 3 で余りが 1 である。
x　　y

14

[規則]
演算子の結合力は以下のとおり。同じ結合力のときは、
左から計算される。また、括弧で計算順序を指定できる。



15

結合力の例

意味

```
x = a / b *c;
```
→ x = (a/b) *c ○

x = a/(b *c)とは違う。

```
y = -a - b*-c;
```
→ y = (-a) -(b*(-c)) ○

結合力が不安であれば、
括弧をつかって明示した方がいい。

```
x = (a/b) *c;
```

```
y = (-a) -(b*(-c));
```

16

実験3

```
/*    結合力実験  priority.c   コメント省略    */
#include <stdio.h>
int  main()
{     int    a;
      int    b;
      int    c;
      int    d;
      int    x;
      int    y;
      int    z;
      a=5;
      b=4;
      c=3;
      d=2;
      x=0;
      y=0;
      z=0;
      /*つづく*/
```

```
      /*つづき*/
      x=-a+b/c*d;

      y=-(a+b/c*d);

      z=-((a+b)/c*d);


      printf("x= %3d ,  y=%3d,  z=%3d ¥n",x,y,z);

      return 0;
}
```

18

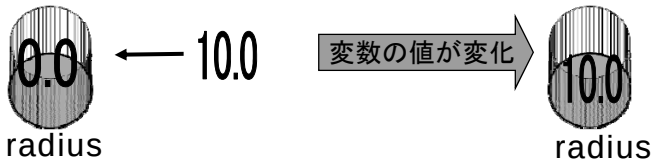
代入演算子

C言語では「＝」は代入演算子（2項演算子）である。
（数学の「＝」とは違う意味）

書式 変数＝式

- 左辺は必ず変数
- 右辺は式（定数や算術式等）

```
radius=10.0;
```



19

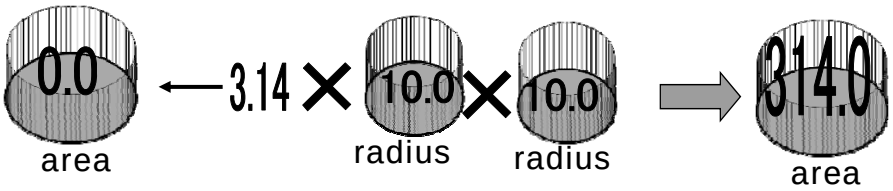
式の計算と代入代入演算子

C言語では「＝」は代入演算子（2項演算子）である。
（数学の「＝」とは違う意味）

書式 変数＝式

- 代入の右辺は式（定数や算術式等）

```
area=3.14*radius*radius;
```



20

型の表現能力

型の表現能力

低い ← char　int　double → 高い

高い＝低い → 問題なし

```
double = int;
```

低い＝ 高い → 切り捨てが起こる。

```
int = double;
```

本演習では、
代入演算子（＝）において
左辺の型と右辺の型は必ず同じにすること。
（スタイル規則参照）

どうしても、違う型になるときは、
キャスト演算子を用いること。

21

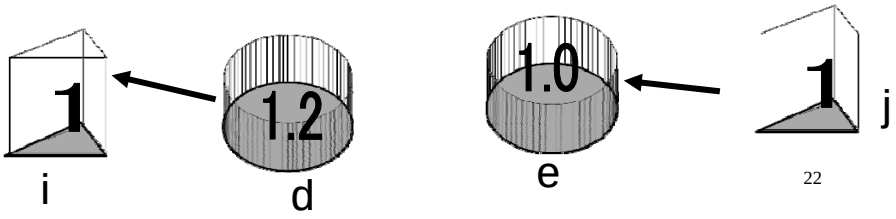
キャスト演算子（型の変換法）

キャスト演算子は型を変換する。
（スタイル規則参照）

書式 （データ型）式

```
int      i;
double   d;
d=1.2;
i = (int) d;
```

```
int   j;
double   e;
j=1;
e = (double) j;
```



22

異なる型同士の演算

[規則]2項演算子の被演算項の型が異なる場合には、
表現能力の低い方を高い方に変換してから演算が行われ、
演算結果は表現能力の高い方になる。

```
int    a;
double b;
double c;
c=a*b;
```

→

```
int    a;
double b;
double c;
c= ((double) a) *b;
```

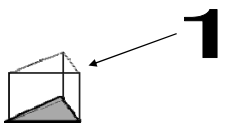
本演習では、
このような自動型変換は用いない事。
（スタイル規則参照）

23

インクリメント演算子とデクリメント演算子

C言語では、1つづつの増減用に、
簡単な形が用意されている。

インクリメント演算子 ++

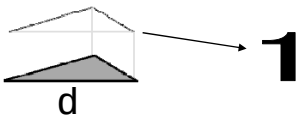


別の書き方。

```
c++;
```

```
c=c+1;
```

デクリメント演算子 --



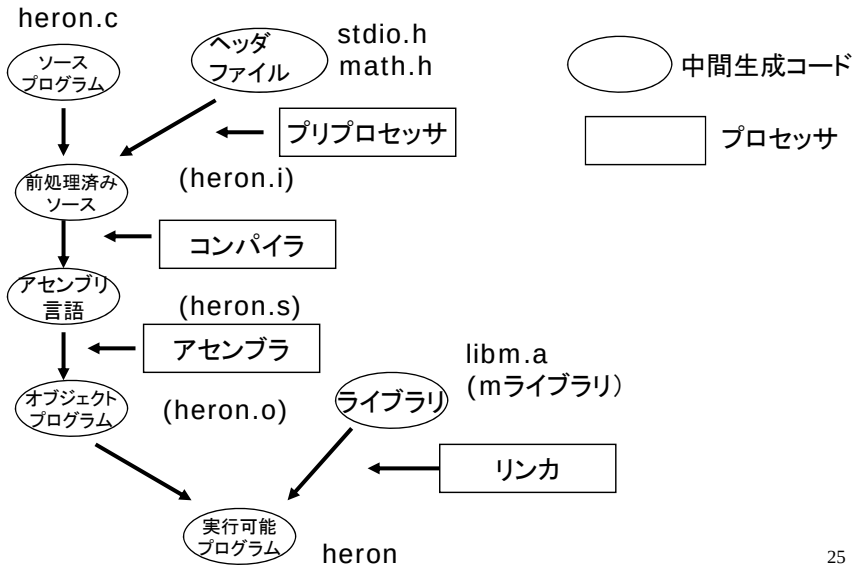
別の書き方。

```
d--;
```

```
d=d-1;
```

24

gccコマンドの詳細



25

プリプロセッサへの指示

[規則]プリプロセッサへの指示行は、
必ず # ではじまる。

例

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

#define  MAX  1000
```

注意:プリプロセッサ行は
行末にセミコロン「;」をつけない。

26

#define

```
#define　文字列1 文字列2
```

ソースのなかの文字列1を文字列2に書き換える（置換する。）

この定義をマクロ定義と呼び、
文字列1をマクロ名、
文字列2をマクロ展開という。

通常の変数と区別するため、本演習ではマクロ名に英小文字を用いないこと。（スタイル規則参照）

例

```
#define  M_PI   3.14159265358979323846   /*  pi  */
```

math.hの中でこう定義されている。

27

#difine例

```
#define    EPS  (1.0e-5)
int main()
{
      .
      .
      yukou=data+EPS;
```

同じ効果

```
int main()
{
      .
      .
      yukou=data+(1.0e-5);
```

意味の分かりずらい
数値だけの記述
（マジックナンバー）は、
できるだけ避けること。

28

#include

他のファイルをソースファイル内に読み込む。
読み込まれるファイルをヘッダファイルという。
ヘッダファイルの拡張子は、
.h
である。

書式

#include　<ファイル名＞ → システムが用意している
ヘッダファイルを取り込む

#include　"ファイル名" → 自分で作ったヘッダファイル
などを取り込む

29

代表的なヘッダファイル

stdio.h:　標準入出力用
（ printf() , scanf()等が使えるようになる。）

string.h:文字列処理用
（strcpy(), strcmp()等が使えるようになる。）

stdlib.h:数値変換、記憶割り当て用
（atof(), malloc()等が使えるようになる。）

math.h:数学関数用
(sin(), cos(), sqrt()等の数学ライブラリ関数が
使えるようになる。
mライブラリと一緒に使う。）

（ヘッダファイルで宣言している関数を用いるには、
コンパイルオプションが必要なものもある。
コンパイルの仕方の概略をコメントしておくとよい。）

30

ライブラリ関数の使い方

書式

関数名（式）

単独で使う場合

関数名（式）;

値を変数に代入する場合

変数＝関数名（式）;

ライブラリ関数：
誰かがあらかじめ作っておいてくれたプログラムの部品。
通常ヘッダファイルと一緒に用いる。
コンパイルオプションが必要なものもある。

詳しくは、第9～11回の関数Ⅰ、Ⅱ、Ⅲで扱う。

31

ライブラリ関数使用例

単独で記述する場合

```
printf("辺1:¥n");
```

（）内の文字列を標準出力に出力するライブラリ関数

他の演算子と組み合わせる場合

```
diag=sqrt(2.0)*edge*2.0;
```

同じ効果

sqrt：平方根を求めるライブラリ関数

```
a=2.0;
diag=sqrt(a)*edge*2.0;
```

32

Makefile の記述追加

いままでのMakefile

```
CC　= gcc
all: 実行ファイル名（ソースファイルから.cを除いたもの）
```

これだと、コンパイルオプションがないので、数学関数ライブラリ（mライブラリ）を用いるソースをコンパイルできない。

数学ライブラリを用いるときのMakefileは以下のように記述する。

```
CC　= gcc
LDFLAGS＝－lm
all: 実行ファイル名（ソースファイルから.cを除いたもの）
```

（ガイダンス資料参照）

33

Makefile 例

```
CC　= gcc
LDFLAGS＝－lm
all: 実行ファイル名（ソースファイルから.cを除いたもの）
```

Makefile

```
CC=gcc
LDFLAGS=-lm
all:heron
```

34

3角形の面積を求めるプログラム(p.56参照)

```
/*
        作成日:yyyy/mm/dd
        作成者:本荘太郎
        学籍番号:B00B0xx
        ソースファイル:heron.c
        実行ファイル:heron
        説明:ヘロンの公式を用いて3角形の面積を求めるプログラム
                数学関数を用いるので、
                -lmのコンパイルオプションが必要。
        入力:標準入力から3辺の長さを入力する。
                各入力は、正の実数とし、
                どの順序に入力されてもよい。
        出力:与えられた3辺の長さを持つ三角形の面積を
                標準出力に出力する。面積は正の実数。
*/
/*      プログラム本体は次のページ以降  */
```

```
#include <stdio.h>
#include <math.h>
int  main()
{
        /*      変数宣言                */
        double      edge1;      /*辺1の長さ*/
        double      edge2;      /*辺2の長さ*/
        double      edge3;      /*辺3の長さ*/

        double      heron_d;    /* ヘロンの公式用
                                    の一時保存用の変数*/
        double      area;       /*3角形の面積*/

/*      次ページへ続く      */
```

36

```
        /*      入力処理        */
        printf("辺1の長さ:¥n");
        scanf("%lf",&edge1);
        printf("辺2の長さ:¥n");
        scanf("%lf",&edge2);
        printf("辺3の長さ:¥n");
        scanf("%lf",&edge3);

/*      次ページへ続く          */
```

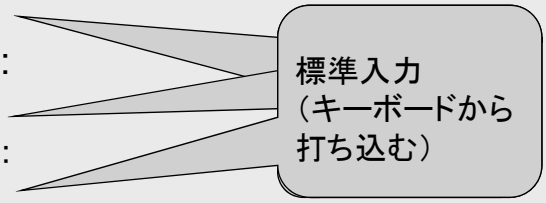
37

```
        /*      計算処理        */
        heron_d=(edge1+edge2+edge3)/2.0;

        area=sqrt(heron_d*(heron_d-edge1)*
                (heron_d-edge2)*(heron_d-edge3));

        /*      出力処理        */
        printf("3角形の面積は  %6.2f  です。¥n",area);

        return 0;
}
```

38

コンパイルと実行

```
$gcc -lm heron.c  -o  heron
$ ./heron
辺1の長さ:
3.0
辺2の長さ:
4.0
辺3の長さ:
5.0
3角形の面積は       6.00です。
$
```



39

第4回　配列

1

今回の目標

- マクロ定義の効果を理解する。
- １次元配列を理解する。
- ２次元配列を理解する。

☆２×２の行列の行列式を計算する
プログラムを作成する

2

行列と行列式

$$\boldsymbol{X} = \begin{pmatrix} x_{00} & x_{01} \\ x_{10} & x_{11} \end{pmatrix}$$ のとき、

$\boldsymbol{X}$ の行列式は

$$\left|\boldsymbol{X}\right| = \begin{vmatrix} x_{00} & x_{01} \\ x_{10} & x_{11} \end{vmatrix} = x_{00}x_{11} - x_{01}x_{10}$$

3

マクロ定義

#define　文字列1 文字列2

ソースのなかの文字列1を文字列2に書き換える（置換する。）

この定義をマクロ定義と呼び、
文字列1をマクロ名、
文字列2をマクロ展開という。

通常の変数と区別するため、本演習ではマクロ名に英小文字を用いないこと。（スタイル規則参照）

例　#define　DATANUM　1000

重要な定数に名前をつける効果がある。

4

#difine 例

```
#define     MEMBER 50
int  main()
{
      double kokugo[MEMBER];
      double suugaku[MEMBER];
      double eigo[MEMBER];
      double rika[MEMBER];
```

同じ効果

```
int  main()
{
      double kokugo[50];
      double suugaku[50];
      double eigo[50];
      double rika[50];
```

プログラムにおける利用データ数の変更が容易になる。

配列の最大要素数は、重要なのでマクロ定義で名前（マクロ名）を付けること。
（スタイル規則参照）

5

配列

同じ型の変数を複数集めたもの。
個々の要素（配列要素）は、普通の変数として扱える。

宣言：

```
要素のデータ型  配列名[要素数];
```

配列を宣言する際は、マクロを用いること。

例

```
#define     SIZE    4

double x[SIZE];
```

```
double x[100];
```

注意：
1. 変数は（要素数）の個数だけ作られる。
2. 宣言で用いる要素数は定数でなくてはならない（変数で指定することは、できない）

6

使い方：

配列名[添字]

で普通の変数のようにつかえる。配列要素は、整数型の値（定数、変数、式）で指定される。要素を指定する整数値を添字（インデックス）と呼ぶこともある。
ただし、添字の値は、「0から（要素数－1）」でなければならない。

例

```
max=x[0];
```

添字の値が不正になるようなプログラムでも、コンパイルはできてしまう。十分注意する事。

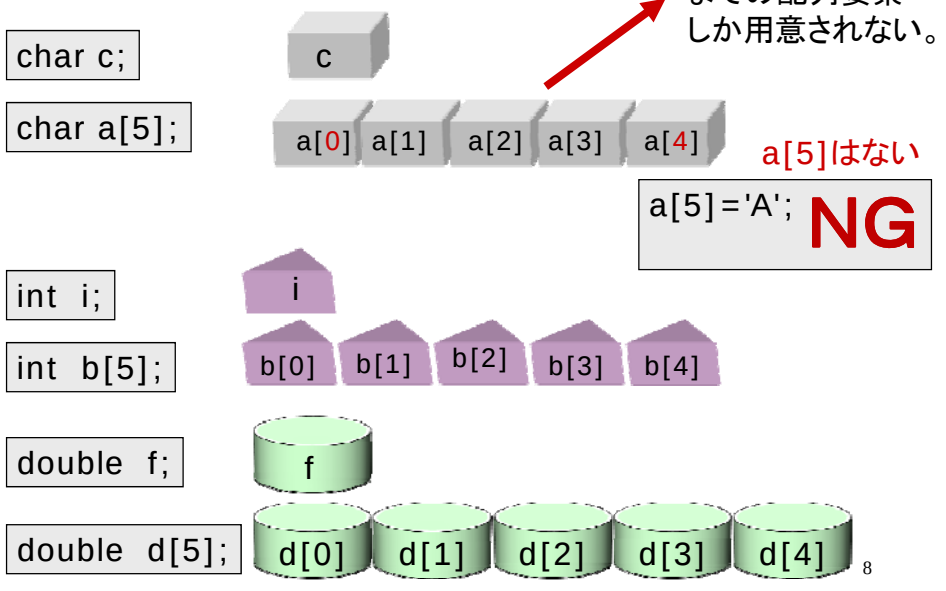
```
int i;
i=3;
min=x[i];
```

添え字にはint型の変数も利用可能である。この場合、変数に蓄えられている値に注意する事。

注意：添字にはint型の式を使い、
添字の値が取りえる範囲に気を付ける事。

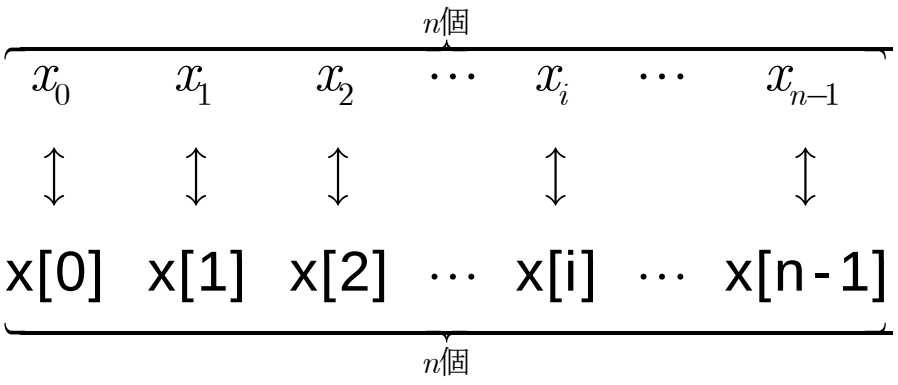
7

イメージ



8

数学の変数とC言語の配列



9

練習1

```
/*  test_array.c   配列実験  コメント省略 */
#include <stdio.h>
#define  SIZE  4
int  main()
{
        double   a[SIZE];
        double   b[SIZE];
        double   c[SIZE];

        /* 配列要素への代入*/
        a[0]=1.0;
        a[1]=1.1;
        a[2]=1.2;
        a[3]=1.3;

        b[0]=2.0;
        b[1]=2.1;
        b[2]=2.2;
        b[3]=2.3;
```

10

```
/*      次に続く        */
```

```
        c[0]=3.0;
        c[1]=3.1;
        c[2]=3.2;
        c[3]=3.3;

        /*間違った代入*/
        b[4]=2.4;     /*間違い*/
        b[5]=2.5;     /*間違い*/

        /*配列内容表示*/
        printf("a[0] =%f ¥n",a[0]);
        printf("a[1] =%f ¥n",a[1]);
        printf("a[2] =%f ¥n",a[2]);
        printf("a[3] =%f ¥n",a[3]);

        printf("b[0] = %f ¥n",b[0]);
        printf("b[1] = %f ¥n",b[1]);
        printf("b[2] = %f ¥n",b[2]);
        printf("b[3] = %f ¥n",b[3]);
```

11

```
/*      次に続く      */
```

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

        /*間違った配列要素の参照*/
        printf("c[4] =  %f ¥n", c[4]);
        printf("c[5] =  %f ¥n", c[5]);
        printf("c[6] =  %f ¥n", c[6]);

        return  0;
}
```

12

2次元配列

宣言:

要素のデータ型　配列名[行の要素数][列の要素数];

例

```
#define     TATE   5
#define     YOKO   3

double      m[TATE][YOKO];
```

使いかた:

配列名[添字1][添字2]

で普通の変数のように使える。
また、添字には(整数型の)
変数や式も使える。

13

2次元配列のイメージ

2次元配列の宣言:
double　m[TATE][YOKO];

				*j*列		
	m[0][0]	m[0][1]	・・・	m[0][*j*]	・・・	
	m[1][0]	m[1][1]	・・・	m[1][*j*]	・・・	
	・ ・ ・	・ ・ ・		・ ・ ・		
*i*行	m[*i*][0]	m[*i*][1]	・・・	m[*i*][*j*]	・・・	
	・ ・ ・	・ ・ ・		・ ・ ・		
						m[TATE-1][YOKO-1]

14

2次元配列でよくある間違い

✗ 間違い1 数学の座標のように、
カンマで区切るのは間違い。

```
m[i, j]
```

NG

✗ 間違い2
配列の添字を入れ替えてしまうは間違い。

`m[i][j]` のつもりで、 `m[j][i]` と書く。

NG

15

練習2

```
/*  tenchi.c    2次元配列実験(転置行列)
      コメント省略
*/
#include <stdio.h>
#define  GYO  2  /* 入力される行列の行の数 */
#define  RETU 2  /* 入力される行列の列の数 */
int  main()
{
      /* 配列の宣言 */
      doulbe x[GYO][RETU];  /*入力された行列*/
      double tx[RETU][GYO]; /*転置行列*/

      /*次に続く*/
```

16

```
	/*  行列の要素の入力 */
	printf("x[0][0]?");
	scanf("%lf", &x[0][0]);
	printf("x[0][1]?");
	scanf("%lf", &x[0][1]);
	printf("x[1][0]?");
	scanf("%lf", &x[1][0]);
	printf("x[1][1]?");
	scanf("%lf", &x[1][1]);

	/* 転置行列の計算 */
	tx[0][0] = x[0][0];
	tx[0][1] = x[1][0];
	tx[1][0] = x[0][1];
	tx[1][1] = x[1][1];

/*次に続く*/
```

17

```
	/* 入力された行列と、その転置行列の表示*/
	printf("x¥n");
	printf("%6.2f %6.2f ¥n", x[0][0], x[0][1]);
	printf("%6.2f %6.2f ¥n", x[1][0], x[1][1]);

	printf("xの転置行列¥n");
	printf("%6.2f %6.2f ¥n", tx[0][0], tx[0][1]);
	printf("%6.2f %6.2f ¥n", tx[1][0], tx[1][1]);

	return 0;
}
```

18

多次元配列

宣言:

データ型 配列名[次元1の要素数][次元2の要素数][次元3の要素数]・・・;

例

```
#define     TATE    5
#define     YOKO    4
#define     OKU     3

double      cube[TATE][YOKO][OKU];
```

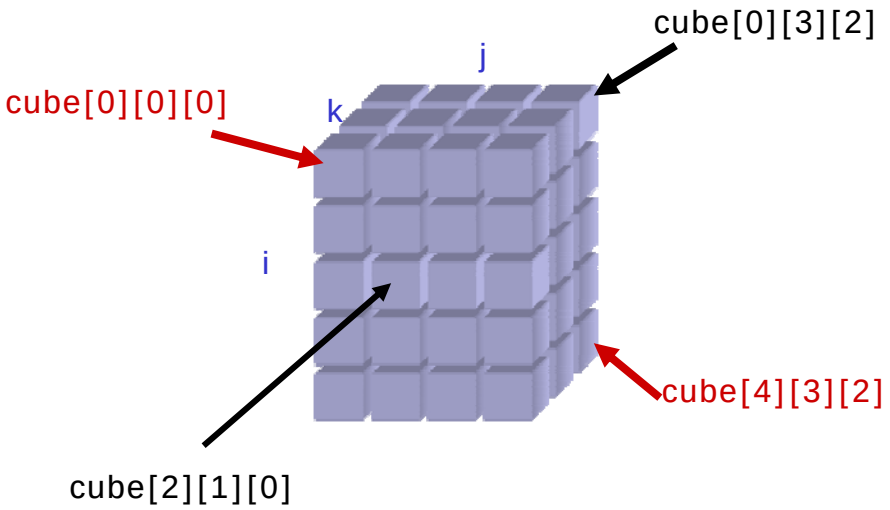
使いかた:

配列名[添字1][添字2][添字3]・・・

で普通の変数のようにつかえる。
また、添え字には(整数型の)
変数や式も使える。

19

3次元配列のイメージ



20

行列式の値を計算するプログラム

```
/*
        作成日: yyyy/mm/dd
        作成者:本荘太郎
        学籍番号:B00B0xx
        ソースファイル:determinant.c
        実行ファイル:determinant
        説明:2×2行列の行列式を計算するプログラム。
        入力:標準入力から行列の4つの成分を入力する。
                行列の成分は全て任意の実数値とする。
                同じ行の要素が連続して入力されるとする。
        出力:標準出力に入力された行列の
                行列式の値を出力する。
                行列式の値は実数値である。
*/
/* 次のページに続く */
```

21

```
/* 続き */

#include <stdio.h>

/*マクロ定義*/
#define   SIZE   2 /* 行列の次元 */

int  main()
{
        /* 変数、配列の宣言 */
        double  matrix[SIZE][SIZE];
                                /* 入力された行列 */
        double  det; /* 行列matrixの行列式の値 */

/* 次のページに続く */
```

```
/* 続き */
      /* 行列の各成分の入力*/
      printf("matrix[0][0]?");
      scanf("%lf", &matrix[0][0]);
      printf("matrix[0][1]?");
      scanf("%lf", &matrix[0][1]);
      printf("matrix[1][0]?");
      scanf("%lf", &matrix[1][0]);
      printf("matrix[1][1]?");
      scanf("%lf", &matrix[1][1]);

/* 次のページに続く */
```

23

```
/* 続き */

      /*行列式の計算*/
      det = matrix[0][0]*matrix[1][1]
            - matrix[0][1]*matrix[1][0];

      printf("行列 matrix :¥n");
      printf("%6.2f %6.2f ¥n",
            matrix[0][0], matrix[0][1]);
      printf("%6.2f %6.2f ¥n",
            matrix[1][0], matrix[1][1]);
      printf("の行列式は、¥n");
      printf("%6.2f です。¥n", det);

      return  0;
}
```

24

実行例

```
$./determinant
matrix[0][0]?1.0
matrix[0][1]?2.0
matrix[1][0]?3.0
matrix[1][1]?4.0
行列 matrix :
   1.00  2.00
   3.00  4.00
の行列式は
 -2.00です。
$
```

第5回条件による分岐

1

今回の目標

- 式、文（単文、ブロック）を理解する。
- 条件分岐の仕組みを理解する。
- 関係演算子、論理演算子の効果を理解する。

☆2次方程式の解を求めるプログラムを作成する。

2

2次方程式の解法

2次方程式 $ax^2 + bx + c = 0$ は、

判別式(discriminant) $D = b^2 - 4ac \geq 0$ のとき、

実数解

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

を持つ。

判別式 $D = b^2 - 4ac < 0$ のとき、

実数解を持たない。

3

式と単文

C言語では、

式：定数、変数、関数呼び出し
と、それらを演算子で結合したもの。

式の例

```
3.14
age=20
radius * radius
area =  3.14 * radius * radius
```

単文：式＋';'

単文の例

```
3.14;
age=20;
radius * radius;
area =  3.14 * radius * radius;
```

4

文と複文

文：単文、複文、・・・

単文

```
******;
```

複文
（ブロック）

```
{
    ****;
    *********;
}
```

文をならべて、
中括弧で囲んだもの。

C言語のプログラムは、
このような文（単文、複文、・・・）から構成される。

5

複文とインデント

```
{
    ****;
    *********;
}
```

中括弧は、
それだけを書く事。

中の文は、
一段字下げして
左端をそろえる事。

中括弧とじは、
対応する中括弧と
同じ列に書く事。

（スタイル規則参照）

6

if文

C言語で、条件式によって、
文を選択して実行する文

書式

```
if(条件式)
    {
        選択実行部分1
    }
```

条件式が **真** なら選択実行部分1を **実行する**。

条件式が **偽** なら選択実行部分1を **実行しない**。

7

式と真偽

C言語には真と偽を表す専用の型はなく、
int型の値で代用する。

真：1（0以外）

偽：0

if文の条件式には、
真偽を表す整数型の式（論理式）
を書く。
（スタイル規則参照）

必ず、中括弧を書く。
（スタイル規則参照）

この例では、
この部分は実行
されます。

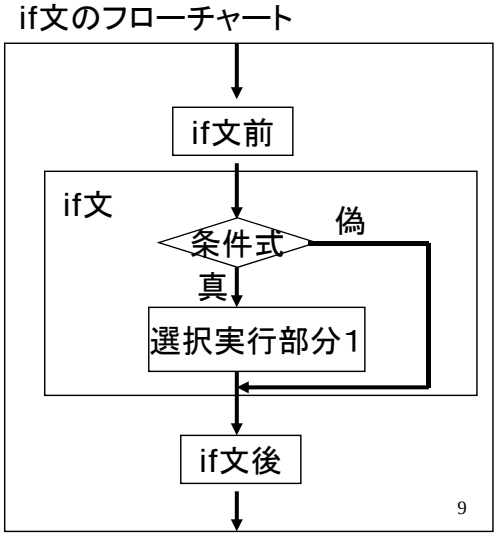
```
int   bool;

bool=1;
if(bool)
    {
        …
    }
```

8

if文の動作1（フローチャート）

書式

```
if(条件式)
    {
        選択実行部分1
    }
```

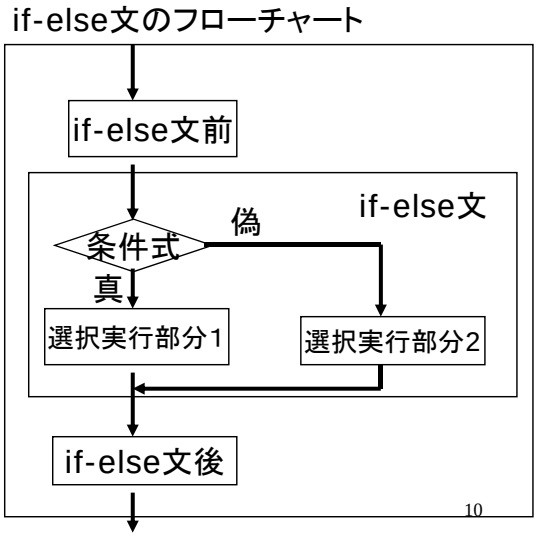


if-else文

条件によって2つの文のどちらかを選択して実行する。

書式

```
if(条件式)
    {
        選択実行部分1
    }
else
    {
        選択実行部分2
    }
```



練習1

```
/*if_test.c      コメント省略  */
#include<stdio.h>
int  main()
{
       int  a;
       printf("実験開始 ¥n");
       if(1)
          {
             printf("常に表示される。¥n");
          }

       if(0)
          {
             printf("常に表示されない。¥n");
          }
/*次のページに続く*/
```

11

```
/*前ページの続き  */
       printf("1(真)または0(偽)を入力して下さい。¥n");
       scanf("%d", &a);

       if(a)
          {
             printf("真です。aの値は0以外です。¥n");
          }
       else
          {
             printf("偽です。aの値は0です。¥n");
          }

       printf("実験終了¥n");
       return  0;
}
```

12

関係演算子

a = = b	a がbと等しい時に真
a ! = b	a がbと等しくない時に真
a < b	a がbより真に小さいとき真
a > b	a がbより真に大きいとき真
a < = b	a がb以下のとき真
a > = b	a がb以上のとき真

関係演算子を使った式は、真偽値を表すint型の値を返す。
本演習では、関係演算子を使った式は論理式として扱い、
算術式とは明確に区別すること。

13

間違いやすい関係演算子（1）

間違い　a =< b　　NG
　　a => b

正しい　a < = b　a がb以下のとき真　　OK
　　a > = b　a がb以上のとき真

他の間違い
a < b < c　NG
a = b ＞ c

関係演算子は2項演算子です。
関係演算子は組み合わせて
使ってはいけません。

これらは、コンパイルエラー
にならない。

14

間違いやすい関係演算子（2）

関係演算子「＝＝」と代入演算子「＝」は間違えやすいので、
気をつける事。

「＝」は代入演算子。
間違いだが
コンパイルエラーにならない

```
if(a = b)
   {
      printf("同じ数字です。¥n");
   }
```

NG

こう書くと、bの値が0以外のときに実行されます。

if文の条件式には論理式（関係演算子を使った式）を書くこと。

```
if(a==b)
   {
      printf("同じ数字です。¥n");
   }
```

OK

15

関係演算子と型

関係演算子は2項演算子です。
両辺の型を一致させること。
（スタイル規則参照）

比較の左辺は実数なのに、
右辺は整数

```
double  a;
if(a <= 0)
   {
      …
```

NG

```
double  a;
if(a <= 0.0)
   {
      …
```

OK

16

練習2

```
/*relation.c      関係演算子実験( コメント省略 ) */
#include<stdio.h>
int  main()
{
        int a;
        int b;
        printf("2つの整数を入力して下さい¥n");
        scanf("%d", &a);
        scanf("%d", &b);
        if(a==b)
           {
               printf("同じ数字です。¥n");
           }
        else
           {
               printf("異なる数字です。¥n");
           }
        return  0;
}
```

17

論理演算子(1)

演算子	演算の意味	演算結果
!A	A の否定 (NOT A)	A が真のとき !A は偽。 A が偽のとき !A は真。
A && B	AかつB (A AND B)	A と B が共に真のとき A&&B は真。 それ以外のときは偽。
A || B	AまたはB (A OR B)	A と B が共に偽のとき A||B は偽。 それ以外のときは真。

論理演算子の被演算項(AやB)
は論理式だけを記述する。
よって、AやBは真偽値を表す。

18

論理演算子（2）

式1 ＆＆ 式2 ＆＆ ・・・＆＆式n

式1から式nまですべてが真なら真。
それ以外の場合は偽。

式1 ｜｜ 式2 ｜｜ ・・・｜｜式n

式1から式nまですべてが偽なら偽。
それ以外の場合は真。

AND と OR が混在するような複雑な論理式を用いるときには、括弧をうまく用いて表現する。

19

3項関係の正しい書き方

間違い

×

a ＜ b ＜ c

a ＝ b ＞ c

数学での書き方は、
C言語ではできない。
（数学とは異なる意味で
実行される。）

NG

正しい

○

(a<b) && (b< c)　aがbより真に小さく、かつ
bがcより真に小さいとき 真。
それ以外のときは 偽。

(a＝＝b) && (b＞ c)　aとbが等しく、かつ
bがcより真に大きいとき 真。
それ以外のときは 偽。

OK

20

論理値と型

論理値はint型で扱うこと。（スタイル規則参照）
したがって、論理演算子の被演算項はすべてint型にする。

aはdouble型なので、
if文の条件には書いてはいけない。

```
double  a;
if(a)
   {
      ……
```

× NG

関係演算子を使って
比較を行う。

```
double  a;
if(a != 0.0)
   {
      ……
```

○ OK

21

練習3

```
/* logic.c      論理演算子実験（ コメント省略 ） */
#include<stdio.h>
int  main()
{
      int a;
      int b;
      int c;
      printf("3つの整数を入力して下さい¥n");
      printf("a=");
      scanf("%d", &a);
      printf("b=");
      scanf("%d", &b);
      printf("c=");
      scanf("%d", &c);
/*  次のページに続く  */
```

22

```
/*続き*/
        if((a<b) && (a<c))
            {
                printf("aが最小です。¥n");
            }

        if((b<a) && (b<c))
           {
                printf("bが最小です。¥n");
           }

        if((c<a) && (c<b))
           {
                printf("cが最小です。¥n");
           }

        return 0;
}
```

23

多分岐（else-ifによる）

書式

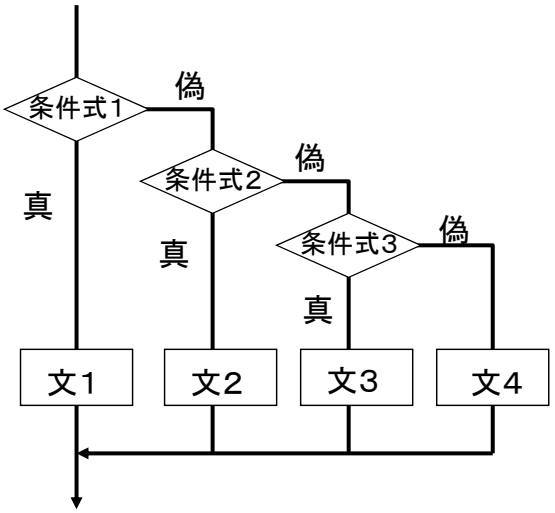
```
if(条件式1)
    {
        選択実行部分1
    }
else   if(条件式2)
    {
        選択実行部分2
    }
.
.
.
else  if(条件式n)
   {
        選択実行部分n
   }
else
  {
        選択実行部分(n+1)
  }
```

式は上から評価されて、
真になった条件式に対応する
選択実行部分が実行される。

すべての条件式が偽なら、
最後のelseの選択実行部分が
実行される。

24

多分岐のフローチャート




```
if(条件式1 )
    {
        文1
    }
else  if(条件式2)
    {
        文2
     }
else  if (条件式3)
    {
        文3
     }
else
    {
        文4
     }
```

25

多重分岐

選択実行部分中にも、if文を書く事ができる。

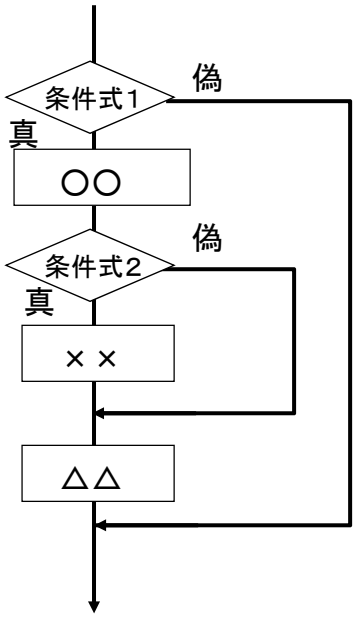
```
if(条件式)
    {
       選択実行部分1
    }
```

ここに、
またif文を書ける。

```
    if(a%2==1)
       {
          printf("aは奇数です。¥n");
          if(a>0)
             {
                printf("aは正の奇数です。¥n");
             }
       }
```

26

多重分岐のフローチャート




```
if(条件式1)
   {
      ○○
      if(条件式2)
         {
            ××
         }
      △△
   }
```

27

2次方程式を解くプログラム（p.67）

```
/*
        作成日:yyyy/mm/dd
        作成者:本荘太郎
        学籍番号:B00B0xx
        ソースファイル:quad_equation.c
        実行ファイル:quad_equation
        説明:
                2次方程式a(x*x)+bx+c=0の解を求めるプログラム。
                数学関数を用いるので、-lmのコンパイルオプションが必要。
        入力:
                標準入力から3つの係数a,b,cを入力する。
                aには0以外の任意の実数値を入力する。
                b,cには任意の実数値を入力する。
                a,b,cの順序に入力する。
        出力:
                標準出力に2つの解（実数値）を出力する。
*/
/*      次のページに続く        */
```

```
/* 続き */

#include <stdio.h>
#include <math.h>
int  main()
{
        /*      変数宣言                */
        double a;       /*2次の係数*/
        double b;       /*1次の係数*/
        doulbe c;       /*定数項*/

        double dis;     /*判別式(discriminant)の値*/
        double root_dis; /* 判別式の値の平方根*/

        double sol1;    /*小さい方の解*/
        double sol2;    /*大きい方の解*/

/*続く*/
```

29

```
/* 続き */
        /* 係数、定数項の入力*/
        printf("2次の項の係数を入力して下さい。a=？¥n");
        scanf("%lf", &a);
        printf("1次の項の係数を入力して下さい。b=？¥n");
        scanf("%lf", &b);
        printf("定数項を入力して下さい。c=？¥n");
        scanf("%lf", &c);

        /*入力値チェック*/
        if(a == 0.0)
          {
                /*a=0.0のときは、2次方程式でないので終了*/
                printf("2次の係数aは0.0以外にして下さい。¥n");
                return  -1;
          }
        /* これ以降では a は0.0以外*/

/*続く  */
```

30

```
/* 続き */
        dis = b*b - 4.0*a*c;        /* 判別式の計算 */

        if(dis>=0.0)  /* 実数解が存在する場合 */
           {
                /* disは正なので平方根を計算できる */
                root_dis = sqrt(dis);
                sol1=((-b)-root_dis)/(2.0*a);
                sol2=((-b)+root_dis)/(2.0*a);
                printf("(%6.2f)(x*x)+(%6.2f)x+(%6.2f)=0.0¥n"
                        , a, b, c);
                printf("の解は、%6.2fと%6.2fです。¥n", sol1, sol2);
           }
        else  /* 実数解が存在しない場合 */
           {
                printf("(%6.2f)(x*x)+(%6.2f)x+(%6.2f)=0.0¥n"
                        , a, b, c);
                printf("を満たす実数解はありません。¥n");
            }

        return 0;
}
```

31

実行例1

```
$ ./quad_equation
2次の項の係数を入力して下さい。a=？
1.0
1次の項の係数を入力して下さい。b=？
-3.0
定数項を入力して下さい。c=？
2.0
(  1.00)(x*x)+( -3.00)x+(  2.00)=0.0
の解は、   1.00 と   2.00です。
$
```

32

実行例2

```
$./quad_equation
2次の項の係数を入力して下さい。a=？
1.0
1次の項の係数を入力して下さい。b=？
1.0
定数項を入力して下さい。c=？
1.0
(  1.00)(x*x)+(  1.00)x+(  1.00)=0.0
を満たす実数解はありません。
$
```

33

第6回　繰り返しI
（条件による繰り返し）

1

今回の目標

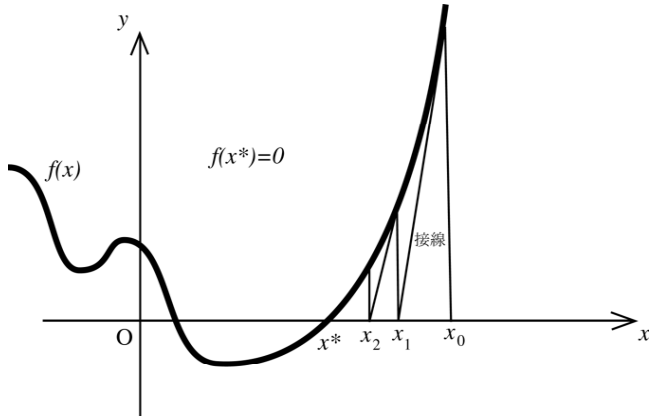
- アルゴリズムの基本となる制御構造（順次、分岐、反復構造）を理解する。
- 繰り返し（反復構造、ループ）を理解する。
- ループからの様々な終了の方法を理解する。
- 繰り返しを用いたアルゴリズムに慣れる。

☆ニュートン法を用いた平方根の計算プログラムを作成する。

2

ニュートン法

$f(x)=0$ の解 x^* を数値計算で求める方法

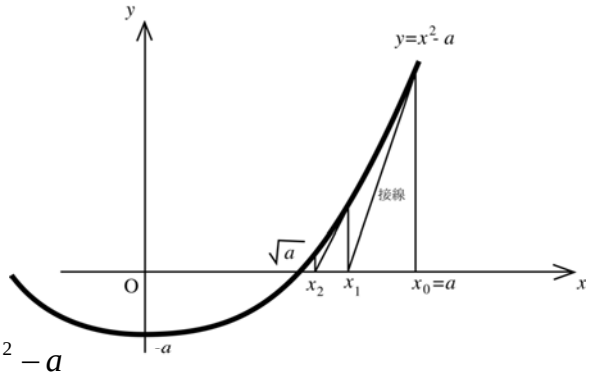


$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} \qquad x_0, x_1, x_2 \ldots, x_\infty \rightarrow x^*$$

3

ニュートン法による平方根の計算

$\sqrt{a}$ は $f(x)=x^2-a=0$ の解なので、



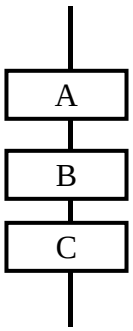
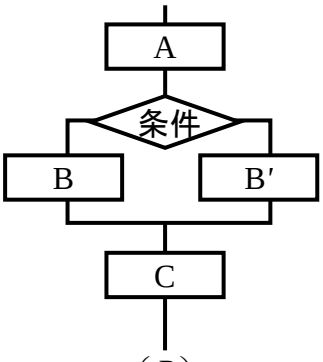
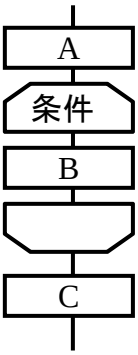
$$x_{i+1} = x_i - \frac{x_i^2 - a}{2x_i}$$

$$= \left(x_i + \frac{a}{x_i} \right) \times \frac{1}{2}$$

$$a = x_0, x_1, x_2 \ldots, x_\infty \rightarrow \sqrt{a}$$

4

プログラムの制御構造

順次構造 （連接構造）	選択構造 （分岐構造）	反復構造 （繰り返し構造）
		
$A \to B \to C$	$A \to \begin{pmatrix} B \\ B' \end{pmatrix} \to C$	$A \to \underbrace{B \to B \to \cdots B} \to C$
の順序で実行される	の順序で実行される	の順序で実行される （反復回数は条件で制御）

基本的な制御構造は、この3種類だけ

5

繰り返し

- 条件（論理式）が真の間繰り返す。

  主に、while文を用いると良い。
  （今回、繰り返しⅠで扱う。）

- 回数を指定して繰り返す。

  主に、for文を用いると良い。
  （次回の繰り返しⅡで詳しく扱う。）

C言語では、主にこの2つの文を用いて
繰り返しを記述する。

6

while文

条件式（論理式）が真である間、命令を繰り返し実行する。

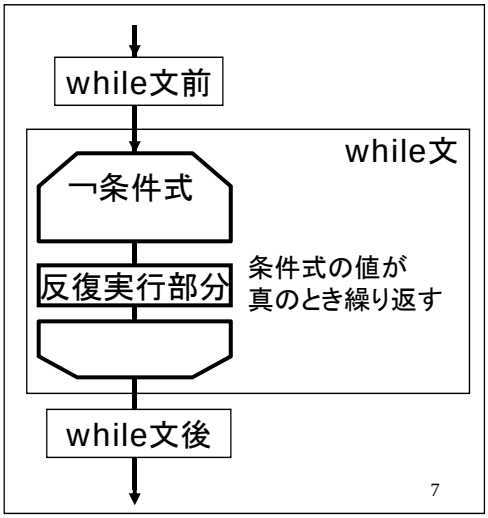
書式

```
while(条件式)
    {
        反復実行部分
    }
```

条件式：
反復を**続ける**条件
を表す論理式
（偽になったら反復終了）

while文は、
反復条件で繰り返し
を制御する。

while文のフローチャート

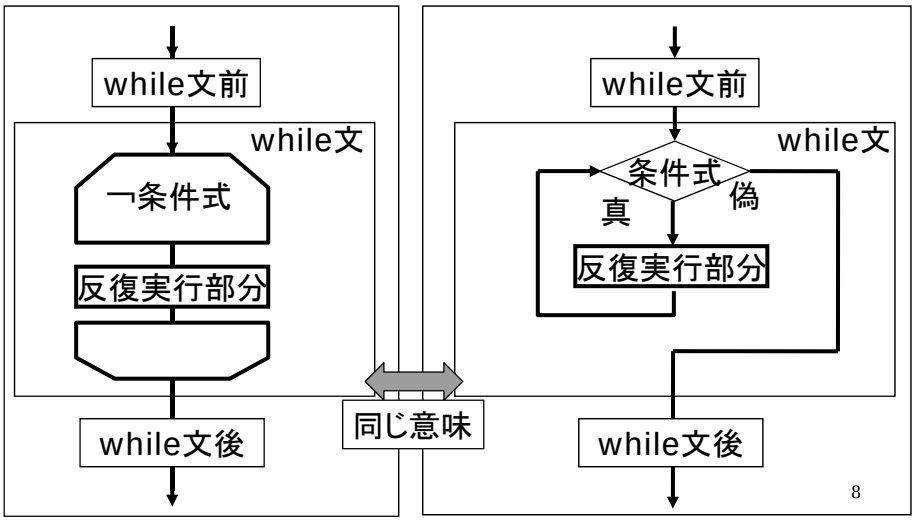


7

whileループのフローチャート

繰り返し文をループと呼ぶ事もある。

while文のフローチャート　　省略しない書き方



8

練習1

```
/*while 文実験1  while_test.c      コメント省略  */
#include<stdio.h>
int  main()
{
      int   a;
      a=1;  /* 最初の一回を必ず実行するため真値を代入  */
      printf("実験開始 ¥n");
      while(a)
         {
            printf("aの値は0以外（真）です。¥n");
            printf("1（真）か0（偽）を入力して下さい。¥n");
            scanf("%d", &a);
         }
      printf("aの値が0（偽）になりました。¥n”);
      printf("実験終了¥n");
      return  0;
}
```

繰り返しを回数で決めるには（while文）

ループカウンタを用いるとよい。

1. ループカウンタ（整数型の変数）を用意する（宣言する）。
2. while文直前でループカウンタを0に初期化する。
3. while文の論理式を
   ループカウンタ ＜ 望む回数
   とする。
4. while文の反復実行部分最後（右中括弧の直前）で、
   ループカウンタをインクリメントする（1増やす）。

10

指定した回数だけ繰り返し（while文）

典型的な書き方

```
#define  LOOPMAX  100
        .
        .
int main()
{
        int   i;    /*ループカウンタ*/
        .
        .
        i=0;    /* ループカウンタの初期化*/
        while( i < LOOPMAX)   /*条件判断*/
          {
                .
                .
                i++;  /*ループカウンタのインクリメント*/
          }
```

11

無限ループとその停止

終わらないプログラムの代表として、無限ループがある。
いろいろなプログラムを作る上で、
無限ループを知っていなければならない。

無限ループの書き方

```
while(1)
    {

    }
```

プログラムが終わらないときには、
プログラムを実行しているkterm上で、
コントロールキーを押しながら、cキーを
押す。（C－c）

それでも
終わらないときは、
教員に相談

12

練習2

```
/* infty_loop.c　無限ループ実験（コメント省略）　*/
#include<stdio.h>
int main()
{
        printf("無限ループ実験開始 ¥n");
        while(1)
          {
                printf("*¥n");
                printf("**¥n");
                printf("***¥n");
                printf("****¥n");
                printf("*****¥n");
                printf("******¥n");
          }
         printf("常に実行されない。¥n");
         return  0;
}
```

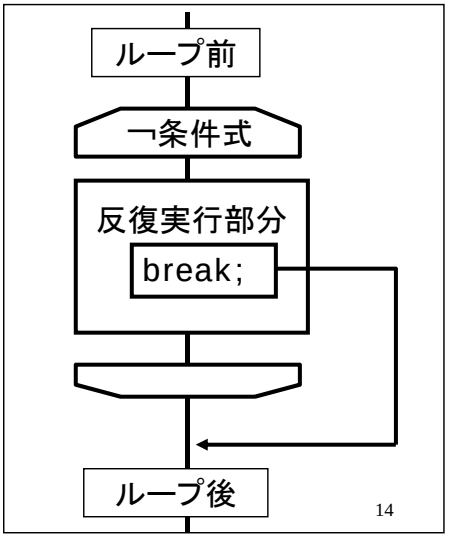
13

break文

反復実行部分内でbreakに出会うと繰り返しが終了する。
（次の実行は、ループを閉じる右中括弧直後から）

書式

```
while(条件式)
   {
      反復実行部分内のどこか
      （break;）
   }
```



14

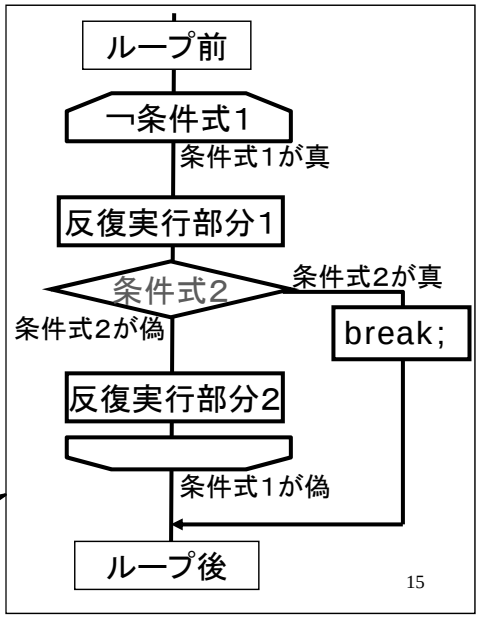
break文の典型的な使い方

典型的な使い方

```
while(条件式1)
    {
        反復実行部分1
        if(条件式2)
            {
                break;
            }
        反復実行部分2
    }
```

2つの条件式でループが終了するので注意して使うこと。

式2は、終了条件を意味する。



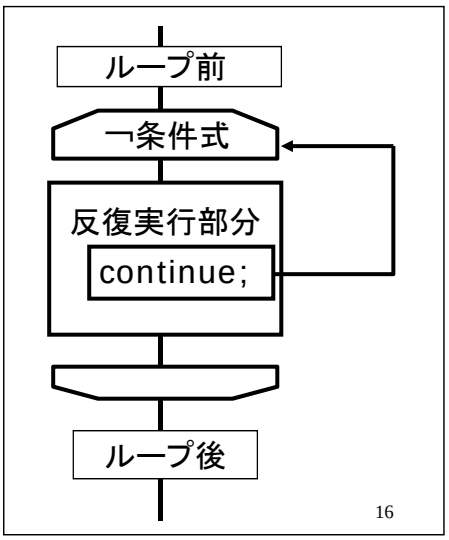
15

continue文

反復実行部分内でcontinue文に出会うと
条件式の判断の箇所まで戻る（次の繰り返し処理を実行する）。

書式

```
while(条件式)
    {
        反復実行部分内のどこか
        （continue;）
    }
```



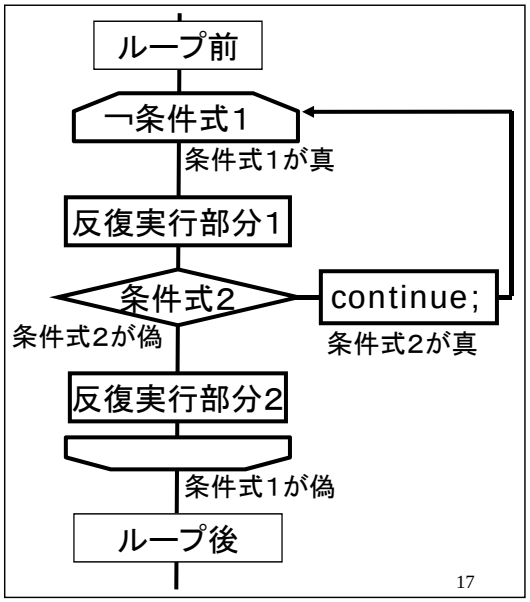
16

continue文の典型的な使い方

典型的な使い方

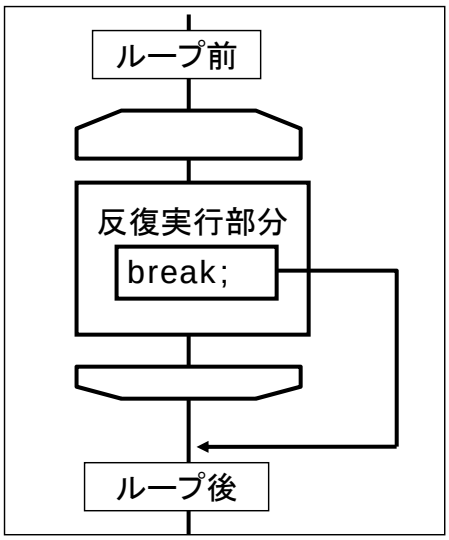
```
while(条件式1)
    {
        反復実行部分1
        if(条件式2)
            {
                continue;
            }
        反復実行部分2
    }
```

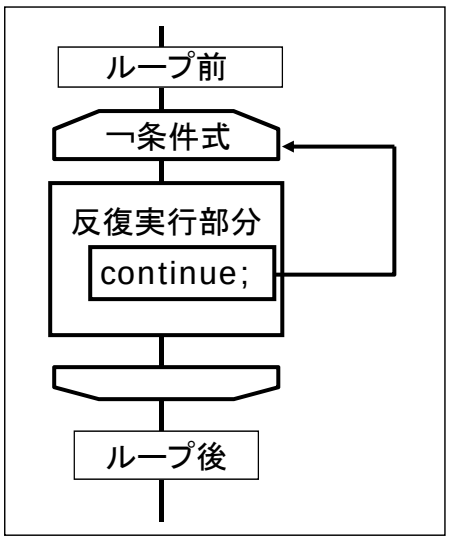
ループ中に反復実行文2を実行するか選択できる。



17

break文とcontinue文

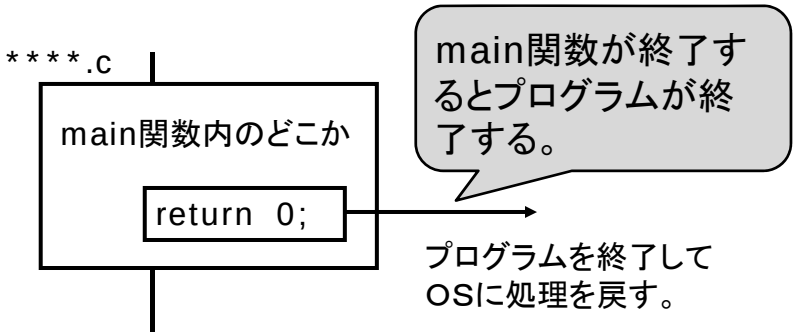




18

return文

return文は関数を終了させる。
main関数内のreturn文はプログラムが終了させる。
（詳しくは、第9～11回の関数Ⅰ、Ⅱ、Ⅲで説明。）



最後でなくても、プログラムを終了させられる。
（エラーチェック等で使うと良い。）

19

平方根を求めるプログラム

```
/*
        作成日:yyyy/mm/dd
        作成者:本荘太郎
        学籍番号:B00B0xx
        ソースファイル:mysqrt.c
        実行ファイル:mysqrt
        説明:ニュートン法を用いて平方根を求めるプログラム。
              求められた平方根の値の二乗と、入力された値の差の絶対値が
              EPS(1.0e-5)より小さくなるまで繰り返しを行う。
              （繰り返し回数がLOOPMAX(1000)回に達しときにも終了する。）
              数学関数を用いるので、-lmのコンパイルオプションが必要。
        入力:標準入力から1つの実数値を入力する。（正、0、負いずれも可）
        出力:入力された実数値の平方根が実数の範囲で存在するとき、
             標準出力にその平方根の近似値を出力する。
             計算の途中経過についても標準出力に出力する。
             入力の平方根が実数でないとき、
             標準出力にエラーメッセージを出力する。
*/
/*      次のページに続く        */
```

20

```
/* 前ページからの続き */

#include <stdio.h>
#include<math.h>
/*マクロ定義*/
#define        EPS    (1.0e-5)     /*微小量、ニュートン法の収束条件 */
#define        LOOPMAX     1000      /*    繰り返しの最大回数*/

int  main()
{
        /* 変数宣言 */
        double  input;          /*  入力される実数,ニュートン法の初期値*/
        double  approx;         /*  平方根の近似値 */
        double  error;          /*  平方根の近似値の二乗と、
                                    入力された値の差の絶対値*/
        int     kaisuu;         /*繰り返し回数*/

/*      次のページに続く        */
```

21

```
/* 前ページからの続き */
        /* 平方根を求めるべき実数値の入力 */
        printf("平方根を求めます。¥n");
        printf("正の実数を入力して下さい。¥n");
        scanf("%lf", &input);

        /*  入力値が正しい範囲であるかチェック  */
        if(input ==0.0)
          {     /*input=0.0のときは、明らかに0が平方根であるので*/
                /*ニュートン法を利用しなくとも良い*/
                printf("%6.2f の平方根は%6.2fです。¥n", input, 0.0);
                return  0;
          }
        else if(input<0.0)
          {
                /*inputが負のとき*/
                printf("負の数なので、実数の平方根はありません。¥n");
                return  -1;
          }
        /* これ以降では、inputは正の実数*/
/*      次のページに続く        */
```

22

```
/* 前ページからの続き */
        /*ニュートン法の初期設定*/
        approx = input;         /*ニュートン法の初期値を入力値に設定*/
        error = fabs(approx*approx - input);
        kaisuu = 0;
        /* ニュートン法の繰り返し処理 */
        while(error > EPS) /* 差がEPSより大きい間は繰り返す*/
           {
                if(kaisuu>=LOOPMAX)
                   {
                        break; /* 繰り返し回数の上限を超えたので終了 */
                   }
                /* ニュートン法の途中経過の表示 */
                printf("x%d = %15.8f ¥n", kaisuu, approx);
                 /* ニュートン法の漸化式の計算 */
                approx = ( approx + (input/approx) ) / 2.0;
                error = fabs(approx*approx - input);

                /* 次の繰り返し処理のための準備 */
                kaisuu++;
           }
/*      次のページに続く          */
```

23

```
/* 前ページからの続き */

        /* 計算結果の出力 */
        printf("%6.2f の平方根は%15.8fです。¥n", input, approx);
        return  0;
}
```

24

実行例1

```
$./mysqrt
平方根を求めます。
正の実数を入力して下さい。
2.0
x0=       2.00000000
x1=       1.50000000
x2=       1.46666667
x3=       1.41421569
2.00の平方根は          1.41421356です。
$
```

25

実行例2

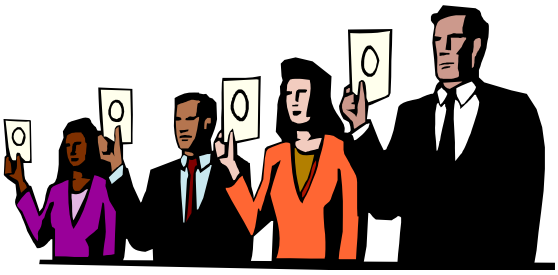
```
$./mysqrt
平方根を求めます。
正の実数を入力して下さい。
0.0
   0.00の平方根は     0.00です。
$
```

実行例3

```
$./mysqrt
平方根を求めます。
正の実数を入力して下さい。
-1.0
負の数なので、実数の平方根はありません。
$
```

26

第7回 繰り返しⅡ
（回数による繰り返し）



1

今回の目標

- for文による繰り返し処理を理解する。
- 多重ループを理解する。

☆等差数列を計算するプログラムを作る。

2

等差数列

初項が a 、公差が d の
　等差数列 a_i ($i = 0, 1, 2, \ldots, n$)
を第 n 項まで計算する。

$$a_0 = a,\quad a_1 = a_0 + d,\quad a_2 = a_1 + d,$$
$$\cdots\ ,\quad a_n = a_{n-1} + d$$

一般項 a_i は以下のような漸化式をみたす。

$$a_i = a_{i-1} + d$$

3

for文

条件式（論理式）が真である間、命令を繰り返し実行する。

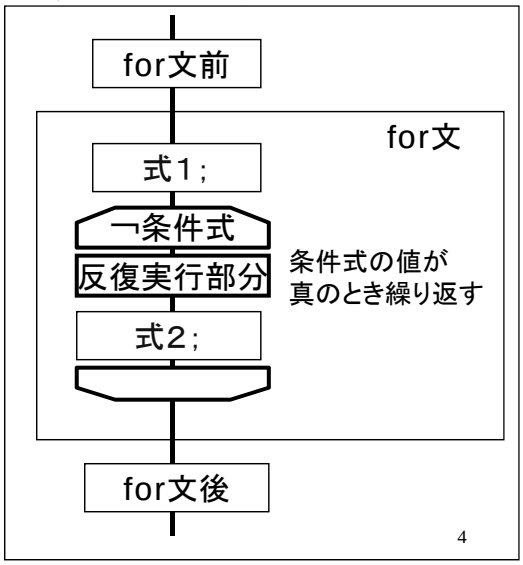
書式

```
for(式1;条件式;式2)
    {
      反復実行部分
    }
```

式1　：ループカウンタの初期化
条件式：反復を**続ける**条件（偽になったら終了）
式2　：ループカウンタのインクリメント

for文は
ループカウンタを
一個所に記述できる。

for文のフローチャート



4

for文

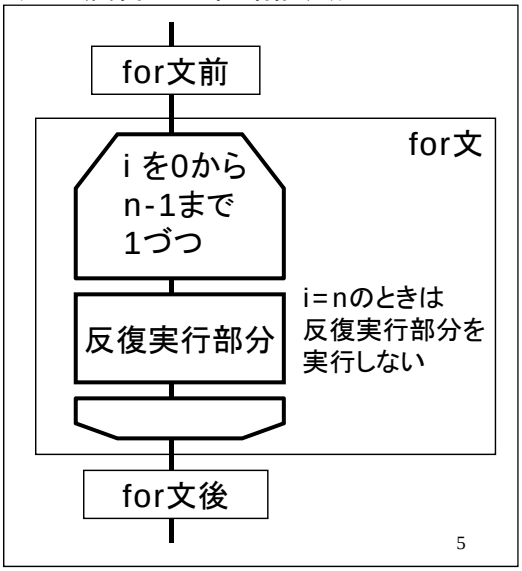
典型的な使い方

```
for(i=0; i<n; i++)
    {
        反復実行部分
    }
```

i=0 :ループカウンタ i の初期化
i<n :反復を続ける条件
（i が 0,1,...,n-1 のとき）
i++ :ループカウンタ i の
インクリメント

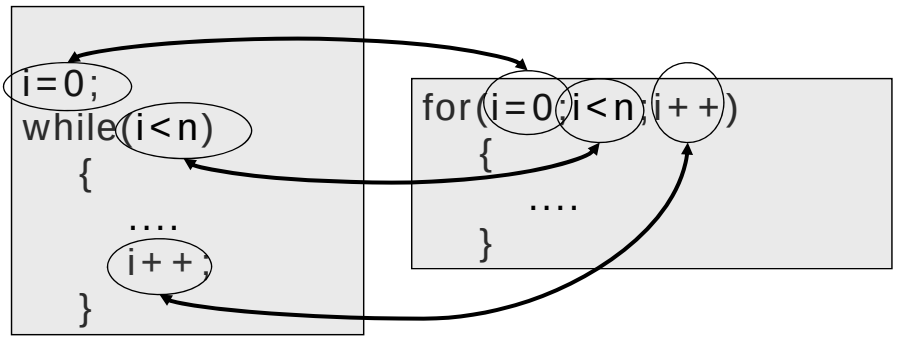
n回繰り返しの定番パターンとして記憶すると良い。
不等号に注意。

for文のフローチャート
（この演習での省略記法）



5

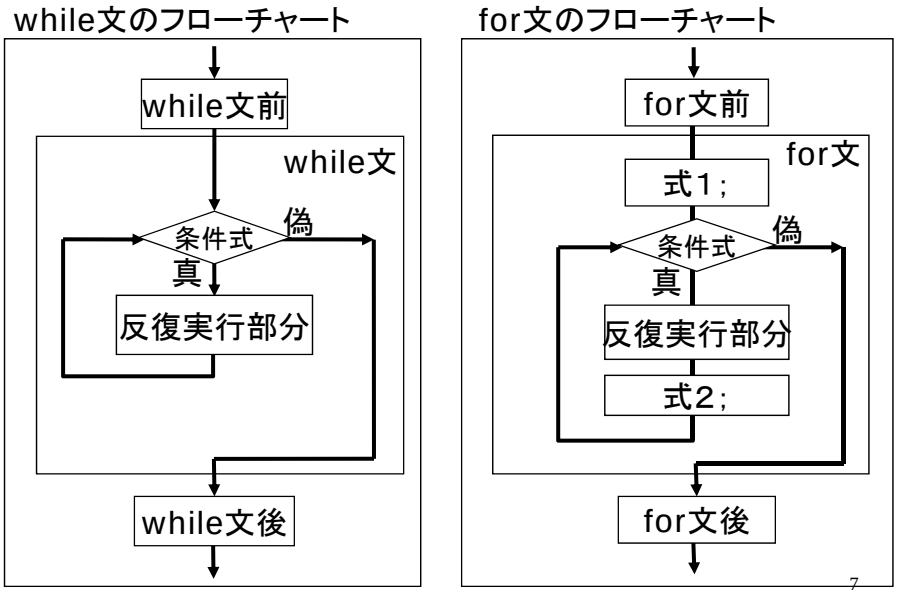
while 文との比較



for文では、回数指定の繰り返しの制御記述を、
一個所にまとめている。

6

while文とfor文のフローチャート



7

while文とfor文の書き換え

```
式1;
while(条件式)
    {
        反復実行部分
        式2;
    }
```

⟺

```
for(式1;条件式;式2)
    {
        反復実行部分
    }
```

回数で制御する繰り返しのときには、
制御に必要な記述がfor文の方が
コンパクトにまとまって理解しやすい。

逆に、繰り返しを論理値で制御するときは、
while文で書くと良い。

8

練習1

```
/* for_test.c      回数指定反復実験（コメント省略）*/
#include<stdio.h>
int  main()
{
        int  n; /* 反復回数 */
        int  i;  /* ループカウンタ */

        printf("反復回数を入力して下さい¥n");
        scanf("%d", &n);
        for(i=0; i<n; i++)
          {
              printf("反復 %d回目 ¥n", i);
          }
        return  0;
}
```

9

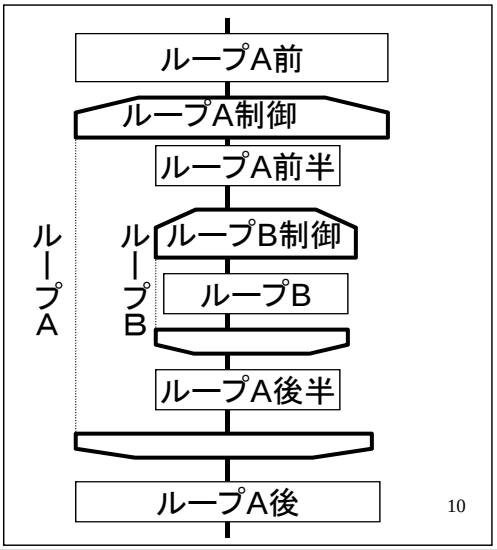
多重ループ

ループ構造の反復実行部分に、小さいループ構造が入っている制御構造。

典型的な例

```
for(式A1;条件式A;式A2)
  {
    ループA前半
    for（式B1;条件式B;式B2）
      {
        ループB
      }
    ループA後半
  }
```

多重ループのフローチャート



10

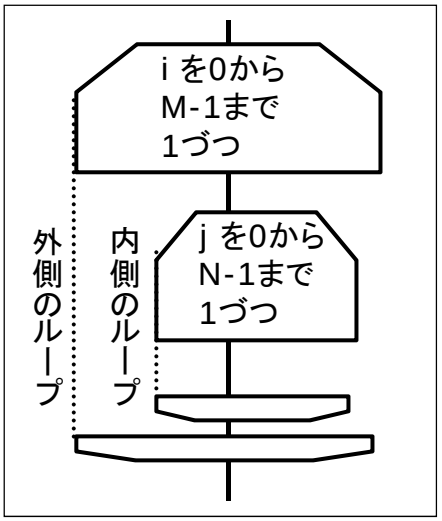
多重ループとループカウンタ

典型的な例

```
#define M 10 /*外側の反復回数*/
#define N 10 /*内側の反復回数*/

int  i;  /*外側のループのカウンタ*/
int  j;  /*内側のループのカウンタ*/

for(i=0; i<M; i++)
    {
        /* 外側のループ */

        for(j=0; j<N; j++)
            {
                /* 内側のループ */
            }
    }
```

多重ループでは、ループごとに
別のループカウンタを用いる。

多重ループのフローチャート



11

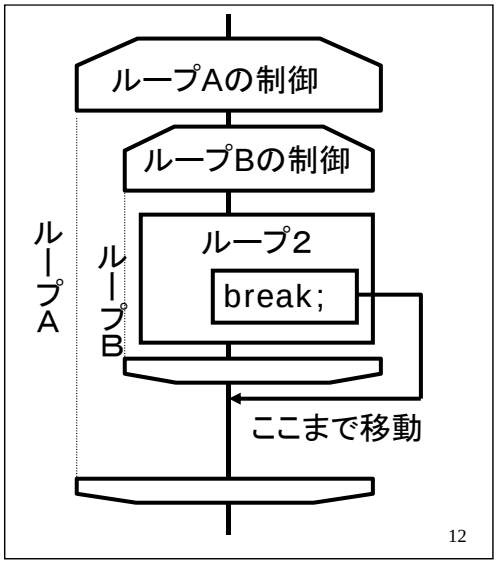
多重ループとbreak文

典型的な例

```
for(式A1;条件式A;式A2)
 {
  for(式B1;条件式B;式B2)
   {
    ( break; )
   }
 }
```

注意:
多重ループ内のbreak文は、
一つだけ外側のループに
実行を移す。

多重ループのフローチャート



12

練習2

```
/*多重ループ実験　multi_loop.c　　コメント省略　*/
#include<stdio.h>
int　main()
{
　　int　i;　/* 外側のループカウンタ*/
　　int　j;　/* 内側のループカウンタ*/

　　printf(" 九九を（半分だけ）表示¥n");

/*　次に続く　*/
```

13

```
　　for(i=1; i<10; i++)
　　　{
　　　　printf("%1dの段:", i);
　　　　for(j=1; j<10; j++)
　　　　　{
　　　　　　printf("%1d*%1d　= %2d　", i, j, i*j);
　　　　　　if(i==j)
　　　　　　　{
　　　　　　　　break;
　　　　　　　}
　　　　　}
　　　　printf("¥n");
　　　}
　　return 0;
}
```

14

等差数列を表示するプログラム

```
/*
        作成日: yyyy/mm/dd
        作成者:本荘太郎
        学籍番号:B00B0xx
        ソースファイル:sequence.c
        実行ファイル:sequence
        説明:初項a、公差dの等差数列を
                第n項まで表示するプログラム。
                ただし、初項は第0項であると考える。
        入力:初項a、公差d、最終項の番号nをこの順に
                標準入力から入力する。
                初項と公差は任意の実数、
                最終項の番号は正の整数とする。
        出力:標準出力に数列の各項の値を出力する。
                数列の各項の値は実数である。
*/
/*      次のページに続く      */
```

15

```
/*      続き    */
#include <stdio.h>

int  main()
{       /* 変数宣言 */
        double      seq;        /* 数列の各項 */
        double      first;      /* 初項 */
        double      difference; /* 公差 */
        int     n;      /* 項数 */
        int     i;      /* ループカウンタ */

        /*初項の入力*/
        printf(“初項aの値を入力して下さい。¥n");
        printf(“a= ？¥n");
        scanf("%lf", &first);
/*      次のページに続く      */
```

```
/*     続き    */
       /* 公差の入力 */
       printf(“公差を入力して下さい。¥n");
       printf(“d= ？¥n");
       scanf("%lf", &difference);
       /* 項数の入力 */
       printf(“項数を入力して下さい。¥n”);
       printf(“n=?¥n”);
       scanf(“%d”,&n);

       /* 入力データのチェック */
       if (n<=0){
              /* 不正な入力:項数が0以下 */
              printf(“項数は正の整数で与えてください¥n”);
              return -1;
       }
/*     次のページに続く     */
```

```
/* 続き */
       /*数列の計算と出力*/
       /*初期設定(初項)*/
       seq=first;
       printf(“a%2d: %6.3f¥n”, 0, seq);

       /*繰り返し処理*/
       for(i=0;i<n;i++)
          {
             seq=seq+difference;
             printf(“a%2d: %6.3f¥n”, i+1, seq);
           }

       return 0;
}
```

実行例1

```
$make
gcc  sequence.c  -o  sequence
$ ./sequence
初項を入力して下さい
a= ？
-7
公差を入力して下さい
d= ？
3
項数を入力して下さい
n= ？
2
a0: -7.000
a1: -4.000
a2: -1.000
$
```

実行例2

```
$./sequence
初項を入力して下さい
a= ？
-7
公差を入力して下さい
d= ？
3
項数を入力して下さい
n= ？
-2
項数は正の整数で与えてください
$
```

20

第8回 繰り返しⅢ
（繰り返し応用）

1

今回の目標

- 配列とfor文の組み合わせ方を理解する。

☆与えられたデータの平均値を
求めるプログラムを作る。

2

平均

n 個のデータ $x_0, x_1, \ldots, x_{n-1}$ の平均値

$$\overline{x} = \frac{x_0 + x_1 + \cdots + x_{n-1}}{n} = \frac{\sum_{i=0}^{n-1} x_i}{n}$$

を計算する。

3

for文と配列

for文と配列は大変相性が良い。
例えば、配列dataの中身を出力する場合に、
ループカウンタ（int型の変数）を配列の添え字として用いれば、
プログラムが非常にシンプルになる。

```
printf("%d¥n",data[0]);
printf("%d¥n",data[1]);
printf("%d¥n",data[2]);
printf("%d¥n",data[3]);
...
printf("%d¥n",data[9]);
```

⇔

```
for(i=0; i<10; i++)
    {
        printf("%d¥n",data[i]);
    }
```

配列の添え字には、int型の定数だけでなく、
int型の式が使える。
ループカウンタとの組合せは典型的な使い方。

4

練習1

```
/*配列読み出し実験  print_array.c      コメント省略  */
#include<stdio.h>

/* マクロ定義 */
#define  SIZE  10 /* 配列の要素数 */

int  main()
{
      int  i; /* ループカウンタ*/
      int  data[SIZE];  /* データを格納する配列 */

      /* データの初期化 */
      data[0]=0;
      data[1]=1;
      data[2]=2;
/*  次に続く  */
```

```
      data[3]=3;
      data[4]=4;
      data[5]=5;
      data[6]=6;
      data[7]=7;
      data[8]=8;
      data[9]=9;

      /* 配列の中身の表示 */
      for(i=0; i<SIZE; i++)
            {
                  printf("data[%2d]=%2d¥n",i,data[i]);
            }

      return 0;
}
```

6

配列へのデータの入力

配列にデータを入力する場合にも、
forループを用いると便利である。

例えば，int型配列dataにデータを格納する場合に、
ループカウンタを配列の添え字として用いて、
繰り返しscanf関数を呼び出せば、プログラムが単純になる。

```
scanf("%d", &data[0]);
scanf("%d", &data[1]);
scanf("%d", &data[2]);
scanf("%d", &data[3]);
...
scanf("%d", &data[9]);
```

⇔

```
for(i=0; i<10; i++)
    {
        scanf("%d", &data[i]);
    }
```

7

練習２

```
/*配列へのデータ格納実験　scan_array.c　　コメント省略
*/
#include<stdio.h>

/* マクロ定義 */
#define  SIZE  3 /* 配列の要素数 */

int  main()
{
      int  i; /* ループカウンタ*/
      int  data[SIZE];  /* データを格納する配列 */

/*  次に続く  */
```

8

```
	/* データの入力 */
	for(i=0; i<SIZE; i++)
		{
			scanf("%d", &data[i]);
		}

	/* 配列の中身の表示 */
	for(i=0; i<SIZE; i++)
		{
			printf("data[%2d]=%2d¥n",i,data[i]);
		}

	return 0;
}
```

9

ベクトルの足し算

2つのn次元ベクトル

$$\mathbf{a}=\begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{pmatrix} \qquad \mathbf{b}=\begin{pmatrix} b_0 \\ b_1 \\ \vdots \\ b_{n-1} \end{pmatrix}$$

の足し算を行うプログラムを作る。

$$\mathbf{a}+\mathbf{b}=\begin{pmatrix} a_0+b_0 \\ a_1+b_1 \\ \vdots \\ a_{n-1}+b_{n-1} \end{pmatrix}$$

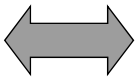
10

ベクトルの足し算

ベクトル **a** , **b** はプログラム中では配列に保存される。
演算結果はベクトル **c** に保存する（プログラム中ではこれも配列）。

ベクトルの足し算は、成分ごとの足し算を繰り返すことで計算される。

$$c_0 = a_0 + b_0$$
$$c_1 = a_1 + b_1$$
$$\vdots$$
$$c_{n-1} = a_{n-1} + b_{n-1}$$



$$c_i = a_i + b_i$$
$$(i = 1, 2, ..., n-1)$$

11

練習3

```
/*ベクトルの足し算  add_vector.c      コメント省略  */
#include<stdio.h>

/* マクロ定義 */
#define  SIZE  3 /* 配列の要素数 */

int  main()
{
      int  i; /* ループカウンタ*/
      double  vector_a[SIZE];  /* 入力ベクトルa */
      double  vector_b[SIZE];  /* 入力ベクトルb */
      double  vector_c[SIZE];  /* ベクトルの和c=a+b */

/*  次に続く  */
```

12

```
	/* ベクトルの成分の入力 */
	printf("ベクトルaを入力して下さい¥n");
	for(i=0; i<SIZE; i++)
		{
			scanf("%lf", &vector_a[i]);
		}

	printf("ベクトルbを入力して下さい¥n");
	for(i=0; i<SIZE; i++)
		{
			scanf("%lf", &vector_b[i]);
		}

/*  次に続く  */
```

13

```
	/* ベクトルの足し算 */
	for(i=0; i<SIZE; i++)
		{
			vector_c[i]=vector_a[i]+vector_b[i];
		}

	/* 結果の表示 */
	for(i=0; i<SIZE; i++)
		{
			printf("c[%d]=%6.2f¥n",i,vector_c[i]);
		}

	return 0;
}
```

14

平均値を求めるプログラム

```
/*
        作成日: yyyy/mm/dd
        作成者:本荘太郎
        学籍番号:B00B0xx
        ソースファイル:average.c
        実行ファイル:average
        説明:n個の実数に対し、その平均値を求めるプログラム。
        入力:まずデータ数として、標準入力から
                データの個数を表す1つの正の整数nを入力。
                続いて、データとして、標準入力からn個の実数を
                任意の順番で入力。
        出力:標準出力に入力されたn個のデータの平均値を出力。
                平均値は実数である。

*/
/*      次のページに続く      */
```

15

```
#include <stdio.h>
/*マクロ定義*/
#define    DATANUM     1000  /* データ個数の上限 */
int  main()
{       /*      変数宣言                */
        int     n;      /*  データ数 */
        int     i;      /* ループカウンタ、配列dataの添字 */
        double  ave;        /* データの平均値 */
        double  sum;        /* データの総和 */
        double  data[DATANUM];   /*データを入れる配列*/

        /*データ数の入力*/
        printf("データ数を入力して下さい。¥n");
        printf("n= ?¥n");
        scanf("%d", &n);
/*      次のページに続く      */
```

16

```
        /*データ数nのチェック*/
        if(n<=0)
          {
              /*  不正な入力:データ数が0以下 */
              printf("データが無いのですね。¥n");
              return  -1;
          }
        /* これ以降では、nは正の整数*/
        if(n>DATANUM)
          {
              /* 不正な入力:上限オーバー*/
              printf("データが多すぎます。¥n");
              return  -1;
          }
        /* これ以降では、nは1からDATANUMまでの整数*/
/*      次のページに続く      */
```

```
        /* 標準入力から配列へデータを読み込む/
        for(i=0; i<n; i++)
           {
              /* n個の実数データの入力  */
              printf("data[%d] = ？", i);
              scanf("%lf", &data[i]);
           }

/*      次のページに続く      */
```

18

```
        /* 総和の計算 */
        /* 初期設定 */
        sum=0.0;
        for(i=0; i<n; i++)
           {
              sum = sum+data[i];
           }

        /*平均の計算  */
        ave = sum/(double)n;

        /*平均値の出力*/
        printf("平均値は %6.2fです。¥n", ave);
        return 0;
}
```

19

実行例1

```
$make
gcc  average.c  -o  average
$ ./average
データ数を入力して下さい。
n= ？
3
data[0]= ？4.0
data[1]= ？2.0
data[2]= ？-3.0
平均値は  1.00 です。
$
```

20

実行例2

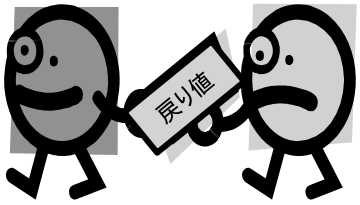
```
$./average
データ数を入力して下さい。
n= ？
0
データが無いのですね。
$
```

実行例3

```
$./average
データ数を入力して下さい。
n= ？
2000
データが多すぎます。
$
```

21

第9回関数Ⅰ
（簡単な関数の定義と利用）



1

今回の目標

- C言語における関数を理解する。
- 仮引数と実引数の役割について理解する。
- 戻り値について理解する。
- 関数のプロトタイプ宣言を理解する。
- 複数の仮引数を持つ関数を理解する。

☆階乗を求める関数を利用して、組み合わせの数を求めるプログラムを作成する

2

ライブラリ関数の使い方（復習）

書式

関数名（式）

単独で使う場合

関数名（式）;

値を変数に代入する場合

変数＝関数名（式）;

ライブラリ関数:
誰かがあらかじめ作っておいてくれたプログラムの部品。
通常ヘッダファイルと一緒に用いる。
コンパイルオプションが必要なものもある。

3

ライブラリ関数使用例

単文として記述する関数

```
printf("辺1:¥n");
```

printf ： 指定された文字列を標準出力に出力するライブラリ関数

式の中で使う関数

```
diag = sqrt(2.0)*edge*2.0;
```

sqrt:平方根を求めるライブラリ関数

今回は、ライブラリ関数 sqrt などと同じように、式の中で使うことができるような関数を自分で作る方法、使う方法について学ぶ。

4

標準ライブラリの数学関数（一部）

sin(x)	$\sin x$	: x の正弦
cos(x)	$\cos x$	: x の余弦
tan(x)	$\tan x$	: x の正接
log(x)	$\log_e x$	: x の（自然）対数
exp(x)	e^x	: e の x 乗（eは自然対数の底）
sqrt(x)	$\sqrt{x}$	: x の平方根
pow(x , y)	x^y	: x の y 乗
fabs(x)	$\|x\|$	: x の絶対値

x や y は double型の式

ヘッダファイル読み込み（プログラム先頭で）
#include <math.h>

コンパイル時（Makefile）mライブラリの指定
LDFLAGS = -lm

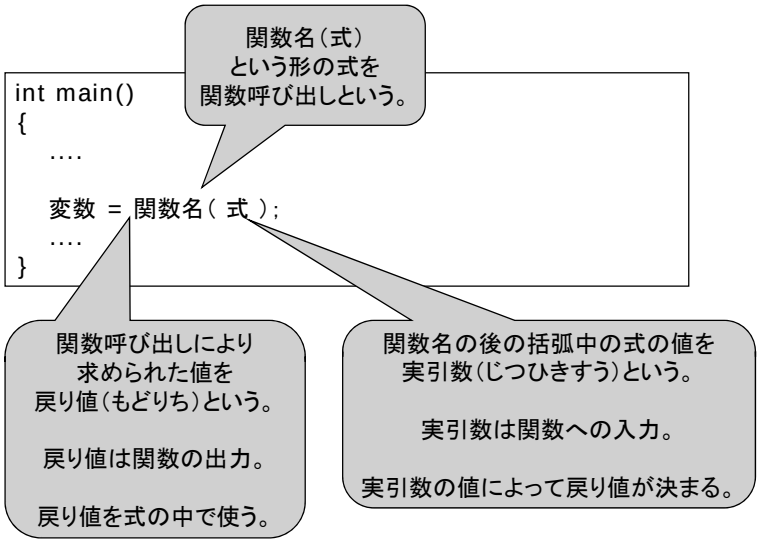
5

練習1

```
/*  数学関数実験  hypo1.c   コメント省略   */
/*      数学関数を用いるので、-lmのコンパイルオプションが必要 */
#include <stdio.h>
#include <math.h>

int  main()
{
   double  base;    /* 直角三角形の底辺の幅 */
   double  height;  /* 直角三角形の高さ */
   double  hypo;    /* 直角三角形の斜辺の長さ */

   printf("底辺？¥n");
   scanf("%lf",  &base);
   printf("高さ？¥n");
   scanf("%lf",  &height);

   hypo = sqrt( pow(base, 2.0) + pow(height, 2.0) );

   printf("底辺:%f , 高さ:%f のとき : 斜辺:%f ¥n",
      base, height, hypo);

   return  0;
}
```

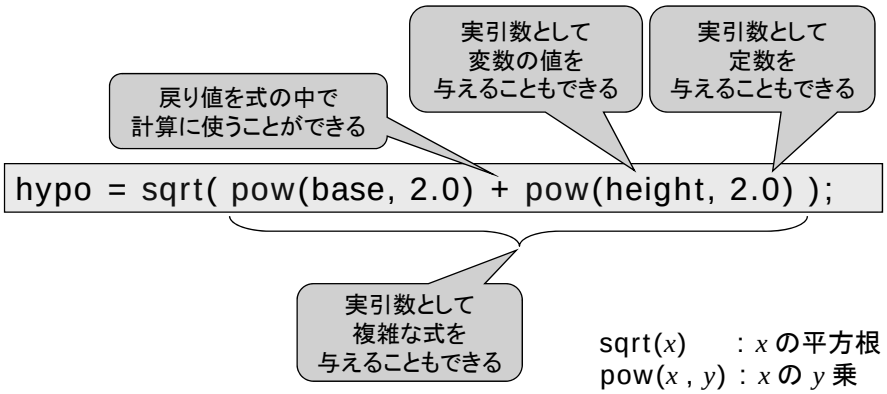
6

関数の利用法（関数呼び出し）



7

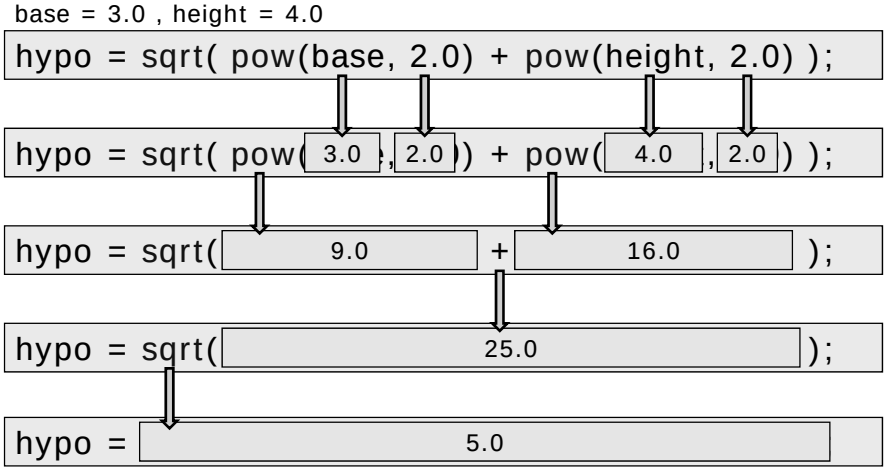
関数呼び出しを含む式



sqrt(x)　　：x の平方根
pow(x , y)：x の y 乗

8

関数呼び出しを含む式の計算



9

関数の定義

書式

```
/*  関数の説明 */
戻り値の型   関数名（仮引数の型  仮引数）
{
   return   戻り値を計算する式;
}
```

関数名は自分で命名する
（関数の意味を反映した名前にすること）

実引数の値を
受け取るための変数

仮引数の値を使って戻り値を計算する
「戻り値の型」の式

関数定義の例

```
/* 実引数を2乗した値を求める関数の定義 */
double  square(double  x)
{
   return  x*x;
}
```

10

自作の関数を含むプログラムの構成

書式

```
/* プログラム全体の説明 */
#include <.....>

戻り値の型　関数名(仮引数の型　仮引数); /* 関数の説明 */

int  main()
{
   *****
   return  0;
}

/*  関数の説明 */
戻り値の型　関数名(仮引数の型　仮引数)
{
   return    戻り値を計算する式;
}
```

関数のプロトタイプ宣言
（セミコロンを忘れずに！）

注意:
自分で作成した関数のプロトタイプ宣言を
ファイルの先頭部分（プログラム全体の説明の後）
に記述する。

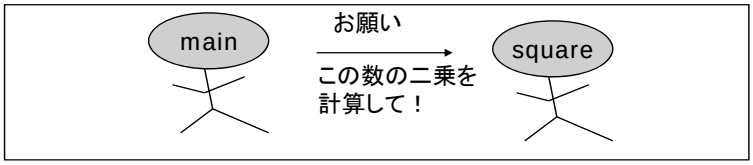
関数の定義
（ここはセミコロン無し）

11

イメージ

以前は、main関数1つしかなかった。

main以外の関数定義があると



分業制にできる。
大きなプログラムを書くには、必要な技術。

12

練習2

```
/*  関数定義実験  hypo2.c   コメント省略   */
/*     数学関数を用いるので、-lmのコンパイルオプションが必要 */
#include <stdio.h>
#include <math.h>


/* 関数のプロトタイプ宣言 */
double  square(double  x) ; /* 実引数を2乗した値を求める関数 */


int  main()
{
   double  base;    /* 直角三角形の底辺の幅 */
   double  height;  /* 直角三角形の高さ */
   double  hypo;    /* 直角三角形の斜辺の長さ */

   printf("底辺？¥n");
   scanf("%lf",  &base);
   printf("高さ？¥n");
   scanf("%lf",  &height);

   hypo = sqrt( square(base) + square(height) ) ;
```

13

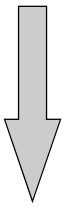
```
/* 続く */
```

```
/*  続き */

   printf("底辺:%f , 高さ:%f のとき : 斜辺:%f ¥n",
      base,  height,  hypo);

   return  0;
}


/* 実引数を2乗した値を求める関数の定義 */
double  square(double  x)
{
   return  x*x ;
}
```

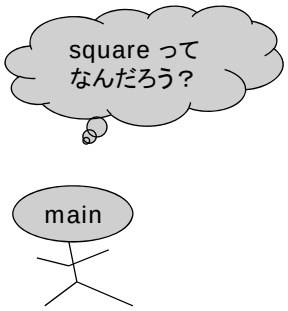
14

プロトタイプ宣言の役割

プログラムは、
上から下に実行されるので、
プロトタイプ宣言が無いと。



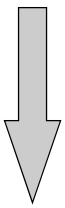
```
int main()
{
   ・・・
   hypo = ・・・ square(base) ・・・ ;
   ・・・
   return   0;
}

double   square(double x)
{
    return  x*x;
}
```



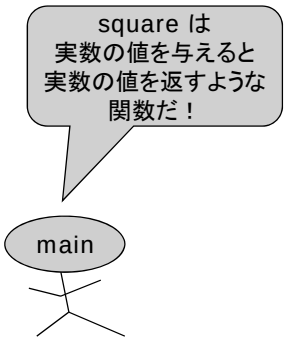
15

プロトタイプ宣言の役割

プロトタイプ宣言があると、
関数であることがわかる。



```
double  square(double  x) ;

int main()
{
   ・・・
   hypo = ・・・ square(base) ・・・ ;
   ・・・
   return   0;
}

double   square(double  x)
{
   return  x*x ;
}
```



16

自作関数の呼び出しを含む式

```
double square(double x)
{
   return   x*x;
}
```

仮引数の名前は
呼び出し時は気にしない

関数呼び出し時に指定された実引数の値が
仮引数に代入されて、戻り値の計算が行われる。

```
hypo = sqrt( square(base) + square(height) );
```

実引数には変数、定数など
任意の式を与えることができる

17

関数呼び出しを含む式の計算

```
double square(double x)
{
   return   x*x;
}
```

base = 3.0 , height = 4.0

```
hypo = sqrt( square(base) + square(height) );
hypo = sqrt( square( 3.0 ) + square( 4.0 ) );
```

x = 3.0

```
return  x  *  x  ;
return  3.0  *  3.0  ;
return  9.0  ;
```

x = 4.0

```
return  x  *  x  ;
return  4.0  *  4.0  ;
return  16.0  ;
```

```
hypo = sqrt( 9.0  +  16.0 );
```

18

複数の引数を持つ関数の定義

書式

```
/*  関数の説明 */
戻り値の型　関数名（仮引数1の型　仮引数1，仮引数2の型　仮引数2，... ）
{
        return　戻り値を計算する式;
}
```

仮引数の値を使って戻り値を計算する
「戻り値の型」の式

関数定義の例

```
/* 二つの実数値の2乗和を求める関数の定義*/
double  sqsum(double a, double b)
{
   return   square(a)+square(b) ;
}
```

19

練習3

```
/*  関数定義実験  hypo3.c　コメント省略　*/
/*　　数学関数を用いるので、-lmのコンパイルオプションが必要 */
#include <stdio.h>
#include <math.h>

/* 関数のプロトタイプ宣言 */
double   square(double  x) ; /* 実引数を2乗した値を求める関数 */
double   sqsum(double  a, double b) ;
/*二つの実数値の2乗和を求める関数*/

int  main()
{
   double  base;    /* 直角三角形の底辺の幅 */
   double  height;  /* 直角三角形の高さ */
   double  hypo;    /* 直角三角形の斜辺の長さ */

   printf("底辺？¥n");
   scanf("%lf",  &base);
   printf("高さ？¥n");
   scanf("%lf",  &height);

   hypo = sqrt( sumsq(base, height) ) ;
```

20

```
/* 続く */
```

```
/*  続き */

   printf("底辺:%f , 高さ:%f のとき : 斜辺:%f ¥n",
      base,  height,  hypo);

   return  0;
}


/* 実引数を2乗した値を求める関数の定義 */
double   square(double  x)
{
   return  x*x ;
}

/* 二つの実数値の2乗和を求める関数の定義*/
double  sqsum(double a, double b)
{
   return   square(a)+square(b) ;
}
```

21

平面上の線分の長さを求めるプログラム

```
/*
   作成日:yyyy/mm/dd
   作成者:本荘  太郎
   学籍番号:B0zB0xx
   ソースファイル: lineseg2d.c
   実行ファイル: lineseg2d
   説明:平面上の線分の長さを求めるプログラム。
        数学関数を利用するため、コンパイルオプション –lm が必要。
   入力:標準入力から二次元平面上の2つの点(p、qとする)の座標を入力。
        点pのx座標、y座標、点qのx座標、y座標の順に
        4個の実数値(doubleで扱える任意の値)を空白または改行で区切って入力する。
        ただし、点pと点qは等しい座標を持つ点ではないものとする。
   出力:標準出力に点pと点qを二つの端点とする線分の長さ(正の実数値)を出力。
*/
#include <stdio.h>
#include <math.h>

/* 関数のプロトタイプ宣言 */
double  square(double  x) ; /* 実引数を2乗した値を求める関数 */
double  sqsum(double  a, double b) ; /*二つの実数値の2乗和を求める関数*/
double  distance(double x1, double y1, double x2, double y2) ;
/* 点 (x1,y1) と点 (x2,y2) の間の距離を求める関数 */
```

22

```
/*  次に続く */
```

```
/* 続き */

int  main()
{
   double  p_x;     /* 一方の端点(点p)のx座標  */
   double  p_y;     /* 一方の端点(点p)のy座標  */
   double  q_x;     /* 他方の端点(点q)のx座標  */
   double  q_y;     /* 他方の端点(点q)のy座標  */
   double  length_pq; /* 線分pqの長さ */

   printf("点pの座標?¥n");
   scanf("%lf",  &p_x);
   scanf("%lf",  &p_y);
   printf("点qの座標?¥n");
   scanf("%lf",  &q_x);
   scanf("%lf",  &q_y);

   /* 入力値チェック */
   if ( p_x == q_x  && p_y == q_y )
     {
      /*不正な入力のときには、エラー表示してプログラム終了*/
      printf("与えられた二点の座標が等しいため、");
      printf("この二点を端点とする線分は存在しません。¥n");
      return  -1;
     }
   /* 正しい入力のとき、これ以降が実行される。*/

/*  次に続く */
```

23

```
/*  続き  */

   length_pq = distance(p_x, p_y, q_x, q_y); /* 線分の長さは端点間の距離に等しい */

   printf("点p : (%6.2f,%6.2f) ¥n",  p_x, p_y);
   printf("点q : (%6.2f,%6.2f) ¥n",  q_x, q_y);
   printf("線分pqの長さは%6.2f です。¥n",  length_pq);

   return  0;
}
/* main関数終了 */

/* 実引数を2乗した値を求める関数
   仮引数 x : 2乗すべき値（任意の実数値）
   戻り値    : xの2乗(非負の実数値)を返す。
*/
double   square(double  x)
{
   return  x*x ;
}
/* 関数 square の定義終 */

/*  次に続く */
```

24

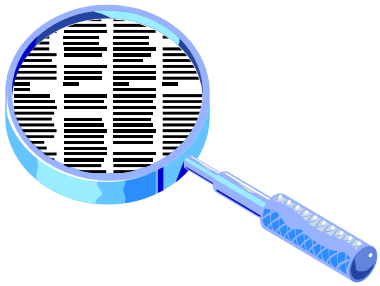
```
/*  続き  */

/* 二つの実数値の2乗和を求める関数仮引数
   仮引数  a：2乗和を求めるべき値（任意の実数値）
   仮引数  b：2乗和を求めるべき値（任意の実数値）
   戻り値    ：aの2乗とbの二乗の和(非負の実数値)を返す。
*/
double  sqsum(double a, double b)
{
   return   square(a)+square(b) ;
}
/* 関数 sqsum の定義終 */

/* 点 (x1,y1) と点 (x2,y2) の間の距離を求める関数
   仮引数  x1：一方の点のx座標(任意の実数値)
   仮引数  y1：一方の点のy座標(任意の実数値)
   仮引数  x2：他方の点のx座標(任意の実数値)
   仮引数  y2：他方の点のy座標(任意の実数値)
   戻り値    :点 (x1,y1) と点 (x2,y2) の間の距離(非負の実数値)を返す。
 */
double  distance(double x1, double y1, double x2, double y2)
{
   return   sqrt(sqsum(x2-x1, y2-y1) ) ;
}
/* 関数 distance の定義終 */

/* 全てのプログラム(lineseg2d.c)の終了 */
```

実行例

```
$make
gcc  lineseg2d.c  -o lineseg2d

$ ./lineseg2d
点pの座標?
-2   1
点qの座標?
1   5
点p：( -2.00,  1.00)
点q：(  1.00,  5.00)
線分pqの長さは  5.00 です。

$
```

26

第10回関数 Ⅱ
（ローカル変数とスコープ）



1

今回の目標

- 戻り値の計算に条件分岐や繰り返し処理を必要とするような、複雑な定義を持つ関数について理解する。
- 関数のローカル変数とスコープの役割について理解する。

☆階乗を求める関数を利用して、組み合わせの数を求めるプログラムを作成する

2

組み合わせの数を求める公式

$$ {}_nC_m = \frac{n!}{m! \times (n-m)!} $$

階乗の計算が何度も出てくる。
階乗を計算する関数があれば、
組み合わせの数は簡単に計算できる。

3

階乗を求める関数の定義（失敗例）

$$ x! = 1 \times 2 \times \cdots \times x $$

だけど・・・

NG

```
/* 実引数の階乗を求める関数の定義 */
int   factorial(int  x)
{
   return  1 * 2 * 3 * ・・・ * (x-1) * x  ;
}
```

C言語では、「・・・」を使って
式を省略することはできない。

場合分けを含む関数の定義

実引数にどのような値が与えられたかによって
戻り値を計算する式が異なるような関数を定義できる。

書式:

```
戻り値の型     関数名(仮引数宣言, ... )
{
   if ( 条件 )
     {
       return   条件が真のときの戻り値を計算する式;
     }
   else
     {
       return   条件が偽のときの戻り値を計算する式;
     }
}
```

仮引数を利用した条件式

5

場合分けを含む関数定義の例

$$\max 2(x, y) = \begin{cases} x & x > y \text{ のとき} \\ y & \text{それ以外} \end{cases}$$

```
/* 二つの実引数のうち、大きいほうの値を求める関数の定義 */
double   max2(double  x, double y)
{
   if ( x>y )
     {
       return   x;
     }
   else
     {
       return   y;
     }
}
```

x>y が真の場合に、
max2の戻り値を求める式

x>y が偽の場合に、
max2の戻り値を求める式

6

練習1

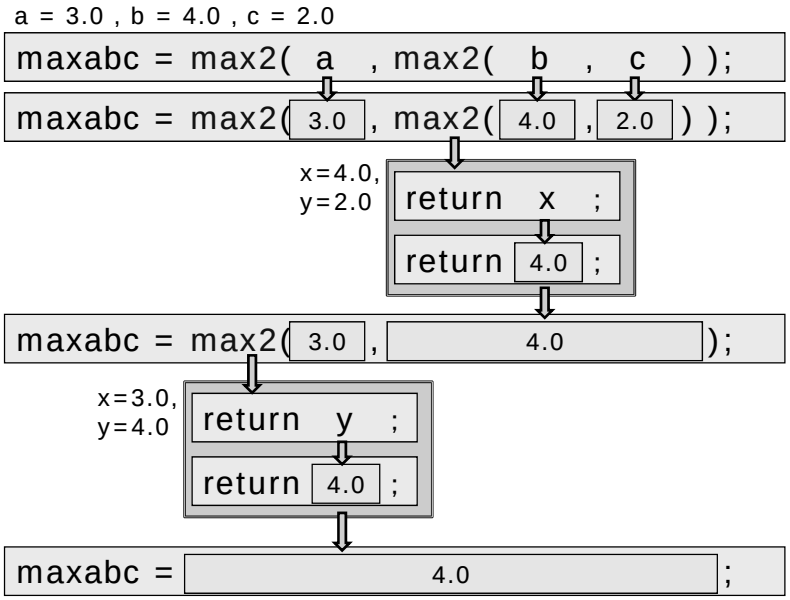
```
/*  条件分岐関数実験   max2.c   コメント省略       */
#include <stdio.h>

/* 関数のプロトタイプ宣言 */
double  max2(double  x, double  y) ;
/*二つの実引数のうち、大きいほうの値を求める関数*/

int  main()
{
   double  a;     /* 1つめの入力値  */
   double  b;     /* 2つめの入力値  */
   double  c;     /* 3つめの入力値  */
   double  maxabc;     /* a,b,cのうち最大の値  */

   printf("a?¥n");
   scanf("%lf",  &a);
   printf("b?¥n");
   scanf("%lf",  &b);
   printf("c?¥n");
   scanf("%lf",  &c);

/* 続く */
```

7

```
/*  続き */

   maxabc = max2(a, max2(b,c) );

   printf("a,b,c の最大値:%f¥n",  maxabc);

   return  0;
}


/* 二つの実引数のうち、大きいほうの値を求める関数の定義 */
double   max2(double  x, double y)
{
   if ( x>y )
     {
      return   x;
     }
   else
     {
      return   y;
     }
}
```

8

関数呼び出しを含む式の計算



9

ローカル変数を持つ関数の定義

ローカル変数を利用することで、
戻り値の計算を手順に分解して記述できる。

書式:

```
戻り値の型　　関数名（仮引数宣言, ... ）
{
   ローカル変数の宣言

   return　戻り値を計算する式;
}
```

ローカル変数:
この関数の戻り値を計算するのに必要な計算途中の値などを入れておくための変数

ローカル変数を利用した計算手順

仮引数の値と
計算されたローカル変数の値を使って
戻り値を計算する「戻り値の型」の式

10

ローカル変数を持つ関数定義の例

```
/* 点 (x1,y1) と点 (x2,y2) の間の距離を求める関数の定義 */
double  distance(double x1, double y1, double x2, double y2)
{
   double  x;       /* 二点のx座標の差 */
   double  y;       /* 二点のy座標の差 */
   double  sqsum; /* 座標の差の二乗和 */

   x = x2 – x1;
   y = y2 – y1;
   sqsum = square(x) + square(y);

   return  sqrt(sqsum) ;
}
```

ローカル変数の宣言

ローカル変数を利用した計算手順

11

練習2

```
/*  関数実験   lineseg2.c   コメント省略          */
/*     数学関数を用いるので、-lmのコンパイルオプションが必要 */
#include <stdio.h>
#include <math.h>

/* 関数のプロトタイプ宣言 */
double  square(double  x) ; /* 実引数を2乗した値を求める関数 */
double  distance(double x1, double y1, double x2, double y2) ;
/* 点 (x1,y1) と点 (x2,y2) の間の距離を求める関数 */

int  main()
{
   double  p_x;    /* 点pのx座標  */
   double  p_y;    /* 点pのy座標  */
   double  q_x;    /* 点qのx座標  */
   double  q_y;    /* 点qのy座標  */
   double  length; /* 線分pqの長さ */

   printf("点pの座標?¥n");
   scanf("%lf",  &p_x);
   scanf("%lf",  &p_y);
   printf("点qの座標?¥n");
   scanf("%lf",  &q_x);
   scanf("%lf",  &q_y);

/* 続く */
```

12

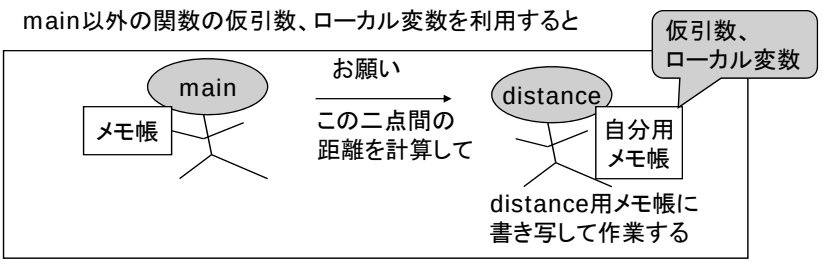
```
/*  続き  */

   length = distance(p_x, p_y, q_x, q_y);
   printf("線分pqの長さ:%f¥n",  length);
   return  0;
}

/* 実引数を2乗した値を求める関数の定義 */
double   square(double  x)
{
   return  x*x ;
}

/* 点 (x1,y1) と点 (x2,y2) の間の距離を求める関数の定義 */
double  distance(double x1, double y1, double x2, double y2)
{
   double  x;       /* 二点のx座標の差 */
   double  y;       /* 二点のy座標の差 */
   double  sqsum; /* 座標の差の二乗和 */

   x = x2 – x1;
   y = y2 – y1;
   sqsum = square(x) + square(y);

   return   sqrt(sqsum) ;
}
```

13

イメージ

以前は、main関数で宣言した変数しかなかった。

main以外の関数の仮引数、ローカル変数を利用すると



関数はローカル変数と仮引数のみを使って作業を行う。
間違ってmainの変数を書き換えてしまうのを防げる。

14

ローカル変数のスコープ（有効範囲）

関数定義の書式:

```
戻り値の型　　関数名（仮引数宣言, ... ）
{
    ローカル変数の宣言

    return　戻り値を計算する式;
}
```

関数定義の仮引数や、関数定義内で宣言したローカル変数は、
宣言した関数定義の内部だけで有効。

したがって、異なる2つの関数の定義で同じローカル変数名を用いても、
それぞれの関数内で別々の変数としてあつかわれる。
（main関数で宣言した変数とも区別される）

15

変数のスコープ（有効範囲）

```
int  main()
{
    main関数内の変数宣言
    *****
    return  0;
}
```

main関数で
宣言した変数の
有効範囲

```
戻り値の型　　関数1（仮引数宣言, ... ）
{
    ローカル変数の宣言
    *****
    return  ～ ;
}
```

関数1の
仮引数、ローカル変数の
有効範囲

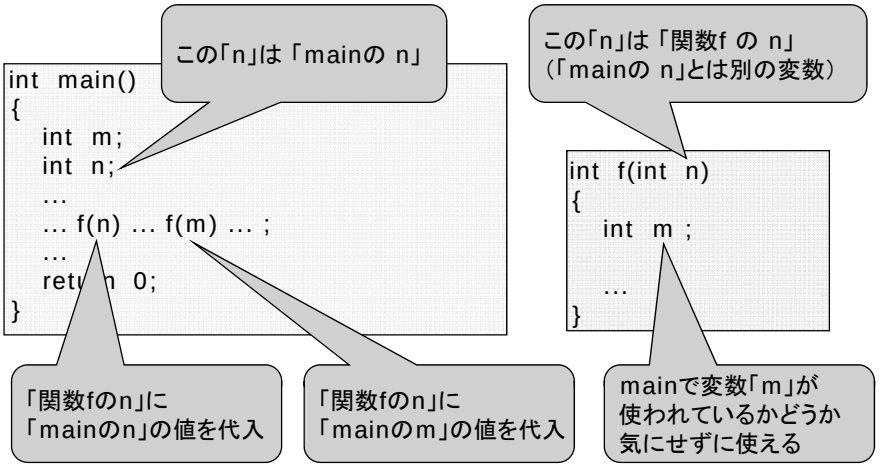
```
戻り値の型　　関数2（仮引数宣言, ... ）
{
    ローカル変数の宣言
    *****
    return  ～ ;
}
```

関数2の
仮引数、ローカル変数の
有効範囲

16

スコープ（有効範囲）の利用

変数のスコープを利用すると、
他の関数の定義で使われている変数や仮引数と区別できる。



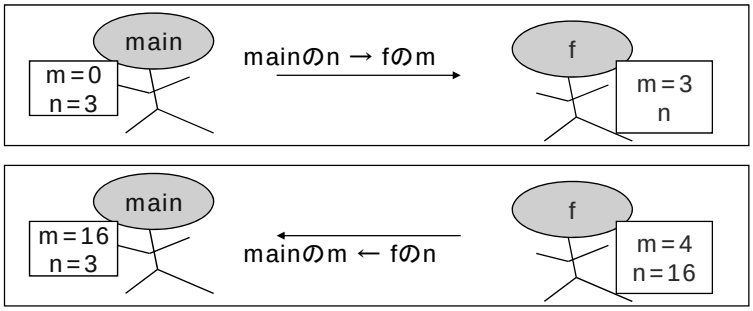
17

練習３

```
/* スコープ実験 test_scope.c コメント省略*/
#include <stdio.h>
int f(int m);

int main()
{
   int m;
   int n;

   m = 0 ;
   n = 3 ;
   printf("（mainの）m = %d ¥n", m);
   printf("（mainの）n = %d ¥n", n);

   m = f( n );

   printf("（mainの）m = %d ¥n", m);
   printf("（mainの）n = %d ¥n", n);

   return 0;
}

/* 次に続く */
```

18

```
/* 続き */
int f(int m)
{
   int n ;

   m = m + 1;
   n = m * m;

   return n;
}
```



19

繰り返しを含む関数の定義

ローカル変数を利用することで、
戻り値の計算式を、繰り返し処理を含む手順に分解して記述できる。

書式:

```
戻り値の型     関数名(仮引数宣言, ... )
{
   ローカル変数の宣言

   while (...)
    {
      ...
    }

   return   戻り値を計算する式;
}
```

ローカル変数:
この関数の戻り値を計算するのに必要な計算途中の値などを入れておくための変数

ローカル変数を利用した繰り返し処理(while や for)を含む計算手順

仮引数の値と
計算されたローカル変数の値を使って
戻り値を計算する「戻り値の型」の式

20

繰り返しを含む関数定義の例

```
/* 実引数の階乗を求める関数の定義 */
int  factorial(int n)
{
   int   i;      /* ループカウンタ */
   int   fact;  /* 1からiまでの積（iの階乗）*/

   fact = 1;
   for ( i = 1; i <= n ; i++ )
     {
       fact = fact * i ;
     }

   return   fact ;
}
```

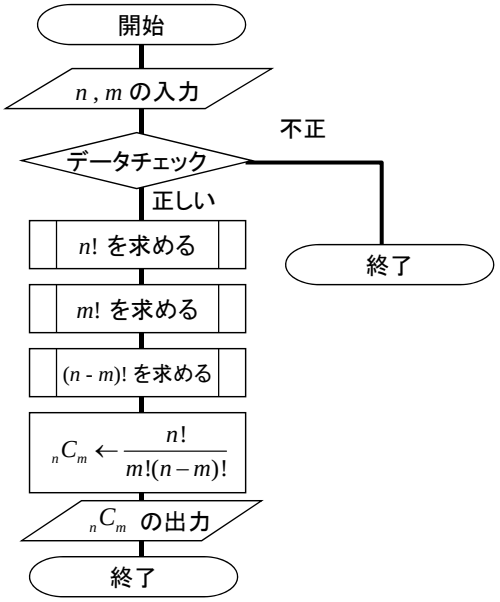
ローカル変数の宣言

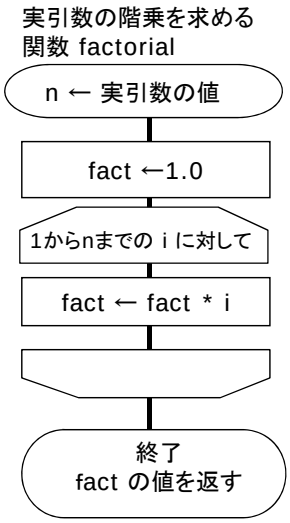
ローカル変数を利用した繰り返し計算手順

21

関数を利用したプログラムのフローチャート

組み合わせの数 ${}_nC_m$ を求める





22

組み合わせの数を求めるプログラム

```
/*
   作成日:yyyy/mm/dd
   作成者:本荘 太郎
   学籍番号:B00B0xx
   ソースファイル:combi.c
   実行ファイル:combi
   説明:組み合わせの数 nCm を求めるプログラム。
   入力:標準入力から2つの正の整数 n,m を入力。(n,mともに15以下とする)
   出力:標準出力に組み合わせの数 nCm を出力。組み合わせの数は正の整数。
*/
#include <stdio.h>

/* プロトタイプ宣言*/
int   factorial(int  n);   /* 階乗を計算する関数 */

int   main()
{
   /*変数宣言*/
   int   n;        /*nCm の n*/
   int   m;        /*nCm の m*/
   int   com;      /*組み合わせの数 nCm*/

/*  次に続く */
```

23

```
/* 続き */

   printf("組み合わせの数 nCm を計算します。¥n);
   printf("Input  n=? ");
   scanf("%d", &n);
   printf("Input  m=? ");
   scanf("%d", &m);

   /* 入力値チェック */
   if ( n<0 || 15<n || m<0 || 15<m || n<m )
     {
       /*不正な入力のときには、エラー表示してプログラム終了*/
       printf("不正な入力です。¥n");
       return  -1;
     }
   /* 正しい入力のとき、これ以降が実行される。*/

   /* 組み合わせの数を計算 */
   com = factorial(n) / ( factorial(m)*factorial(n-m) );

   printf("%d  C %d = %5d¥n", n, m, com);

   return 0;
}
/* main関数終了 */

/*  次に続く */
```

24

```
/* 続き */

/*
   階乗を求める関数
   仮引数 n : 階乗を求める値（0以上15未満の整数値とする。）
   戻り値   : nの階乗(正の整数値)を返す。
*/
int   factorial(int n)
{
   /*  ローカル変数宣言  */
   int   i;            /*ループカウンタ*/
   int   fact;         /* 1からiまでの積(iの階乗）*/

   fact = 1;           /* 0の階乗=1  であるので1を代入*/
   for(i=1; i<=n; i++ )
     {
        fact = fact * i ; /* 1からiまでの積 = (1から(i-1)までの積)×i */
     }

   /* 関数 factorial のローカル変数 fact の値(1からnまでの積)を戻す */
   return  fact;
}
/* 関数factorialの定義終 */
/* 全てのプログラム(combi.c)の終了 */
```

25

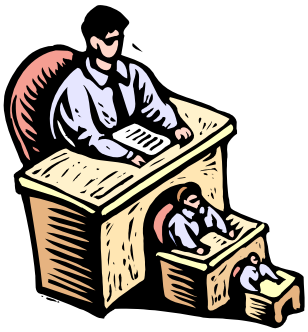
実行例

```
$make
gcc  combi.c  -o combi
$ ./combi
組み合わせの数 nCm を計算します。
Input  n=? 4
Input  m=? 3
4C3  = 4
$
```

```
$./combi
組み合わせの数 nCm を計算します。
Input  n=? 4
Input m=? 5
不正な入力です。
$
```

26

第１１回関数Ⅲ
（関数の応用）



1

今回の目標

- 副作用を持つ関数について理解する。
- 再帰的な関数定義について理解する。

☆階乗を求める関数を再帰的定義により記述する。

2

ライブラリ関数の使い方（復習）

書式

関数名（式）

単独で使う場合

関数名（式）;

値を変数に代入する場合

変数＝関数名（式）;

ライブラリ関数:
誰かがあらかじめ作っておいてくれたプログラムの部品。
通常ヘッダファイルと一緒に用いる。
コンパイルオプションが必要なものもある。

3

ライブラリ関数使用例

単文として記述する関数

```
printf("辺1:¥n");
```

printf ： 指定された文字列を標準出力に出力するライブラリ関数

式の中で使う関数

```
diag = sqrt(2.0)*edge*2.0;
```

sqrt:平方根を求めるライブラリ関数

ライブラリ関数には、sqrt のように式の中で使う関数と、
printf のように単文として記述する関数がある。

4

通常の関数

数学関数などの普通の関数は、
仮引数に受け取った値を使って戻り値を計算することを目的とする。

関数の入力：実引数として与えられた値 （仮引数に受け取る）
関数の出力：戻り値 （関数呼び出しを含む式の中で利用する）

通常の関数の定義例：

```
/* 実引数を2乗した値を求める関数の定義 */
double  square(double  x)
{
   return  x*x;
}
```

戻り値の計算に必要な処理だけを行う

通常の関数の利用例：

式の中に関数呼び出しを書く

```
hypo = sqrt( square(base) + square(height) ) ;
```

5

副作用を持つ関数

仮引数に受け取った値を使って戻り値を計算する処理以外の処理を
関数の副作用と呼ぶ。
仮引数や戻り値を持たない関数でも、副作用によって外部とやりとりを行える。
（普通の関数は副作用を持たない。）

副作用の例： 標準入出力を利用する副作用

```
int  scanInt()
{
   int   input;
   scanf("%d", &input);
   return   input;
}
```

標準入力から値を読み込む副作用

```
void   printInt(int  x)
{
   printf("%d¥n", x );
   return ;
}
```

標準出力へ値を出力する副作用

6

戻り値の無い関数

戻り値の計算を目的としないような関数を書くこともできる。

戻り値の無い関数の定義例:

void という特別な型を指定

```
void   printInt(int  x)
{
   printf("%d¥n", x );
   return ;
}
```

return の後には式を書かない

戻り値の無い関数の利用例:

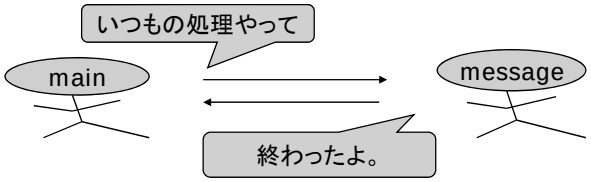
```
printInt(100);
```

関数呼び出しを単文として書く

7

練習1

```
/* side_effect.c  練習1コメント省略 */
#include<stdio.h>
void  message ();

int main()
{
   message( );
   message( );
   return  0;
}

void  message()
{
   printf("こんにちは！¥n");
   return ;
}
```



8

main関数

mainも副作用を持つ関数の一つ。

```
int  main()
{
   int  age;      /*年齢*/

   printf("年齢は？¥n");
   scanf("%d",&age);
   printf("年齢は %d  歳です。¥n",age);

   return 0;
}
```

main関数の中でだけ利用できるローカル変数の宣言

このプログラムで行う処理の内容（標準入出力などの副作用を含む）

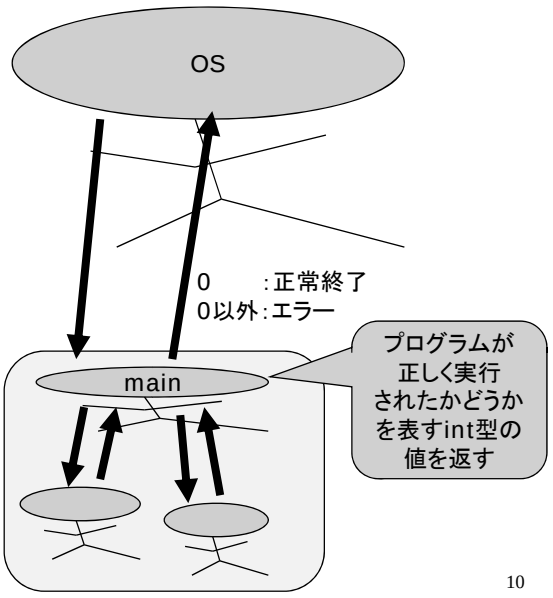
mainは特別な関数で、一つのプログラムに必ず1つだけなければいけない。
プログラムの実行はmain関数の最初から行われる。

9

関数mainの型とOS

```
/* ～.c  */

int  main()
{
   …
   …
   return  0;
}
```

main関数は、
OSとのやりとりを
司る大元の関数。
プログラムに必ず1つ
しかも1つだけ存在する。



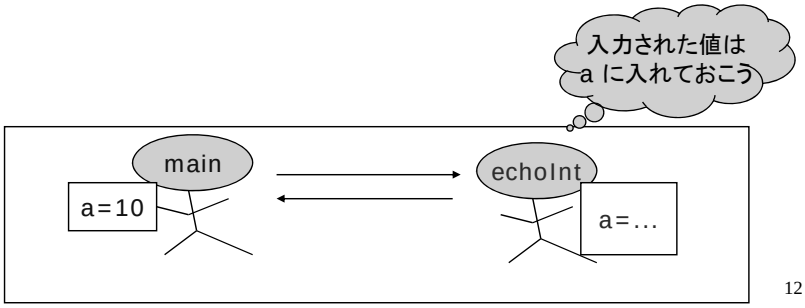
10

練習2

```
/* test_scope2.c 練習2 コメント省略 */
#include<stdio.h>
void echoInt();

int main()
{
   int a;

   a = 10;
   printf("echoInt()実行前¥n");
   printf("mainの変数aの値は %d です。¥n",a);
   echoInt();
   printf("echoInt()実行後¥n");
   printf("mainの変数aの値は %d です。¥n",a);

   return 0;
}

/* 次に続く */
```

11

```
/* 続き */
void echoInt()
{
   int a;

   printf("整数値を入力してください。");
   scanf("%d", &a);
   printf("入力された値（変数aの値）は %d です。¥n", a);

   return ;
}
```



12

階乗 $n!$の2つの数学的表現

（1）繰り返しによる表現

$$n! = 1 \times 2 \times \cdots \times i \times \cdots \times n$$

$$= \prod_{i=1}^{n} i$$

（上記は $n \geqq 1$ のとき。0! は1 。）

（2）漸化式による表現

$$n! = \begin{cases} 1 & n = 0 \text{ のとき} \\ n \times (n-1)! & n \geq 1 \text{ のとき} \end{cases}$$

13

再帰

関数が自分自身を呼び出す事。

C言語では、再帰的な関数を扱える。

書式:

```
戻り値の型　　関数名（仮引数宣言, … ）
{
   if ( 条件 )
     {
       return　この関数を利用せずに戻り値を計算する式;
     }
   else
     {
       return　この関数を再帰的に利用して、戻り値を計算する式;
     }
}
```

特別な値が実引数に与えられた場合は、自分自身を呼び出さない。（再帰の基礎）

それ以外の値が実引数に与えられた場合には、自分自身を呼び出す式を使って戻り値を計算する。

14

再帰関数例

```
int    factorial(int n)
{
   if (n==0)
     {
      return   1 ;
     }
   else
     {
      return  n * factorial(n-1);
     }
}
```

0!=1 なので、
実引数が 0 の場合は
自分自身を呼び出さない。
（再帰の基礎）

0以外の値が実引数に
与えられた場合には、
$n! = n \times (n-1)!$
であることを利用して階乗を求める。
（関数の再帰呼び出し）

⇕ 漸化式をそのままプログラムにできる。

$$n! = \begin{cases} 1 & n = 0 \text{ のとき} \\ n \times (n-1)! & n \geq 1 \text{ のとき} \end{cases}$$

15

再帰の典型的な書き方

書式だけを抽出

```
int   関数名(int n)
{
   if(n==0)
     {
      /*再帰関数の基礎*/
      return  ????;
     }
   else
     {
      /*再帰部分*/
      return   関数名(n-1);
     }
}
```

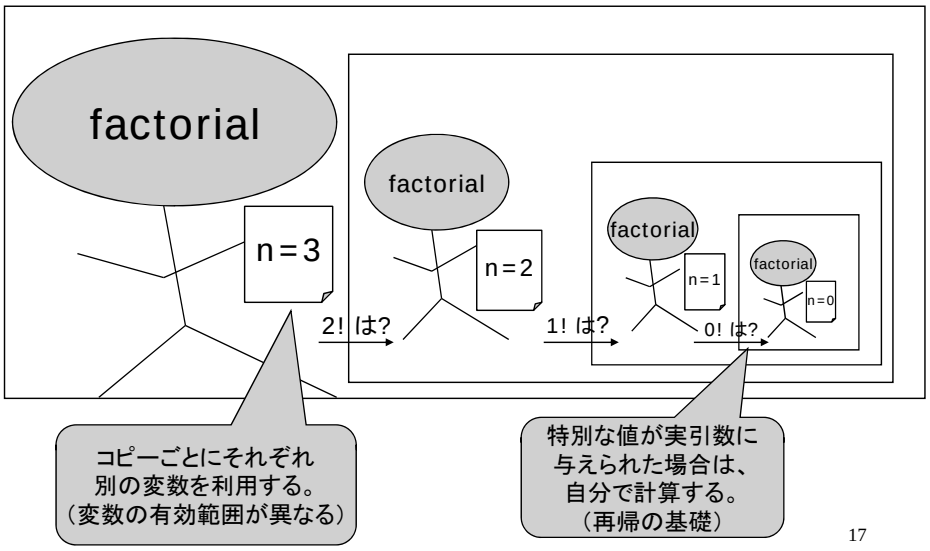
必ず「再帰の基礎」（出口）
を作ること。

再帰呼び出しは、
必ず出口に近づくようにすること。
（n が正の整数ならば、
n よりは n-1 のほうが0に近い）

16

イメージ

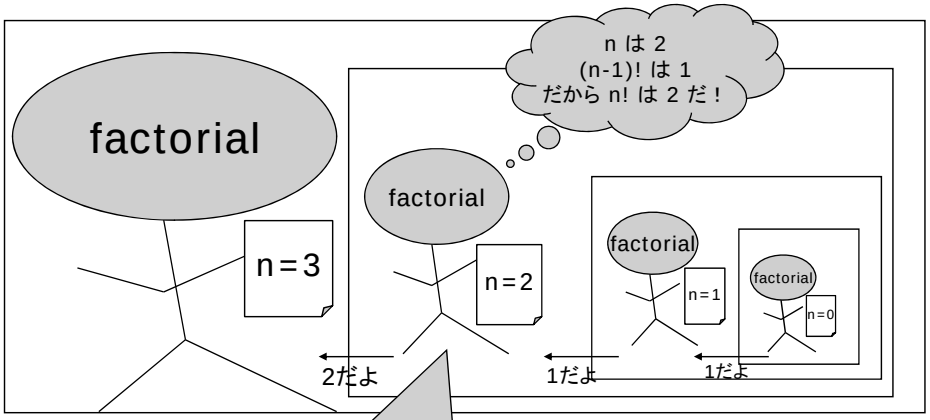
再帰関数は、自分自身のコピーに仕事を押し付ける。



17

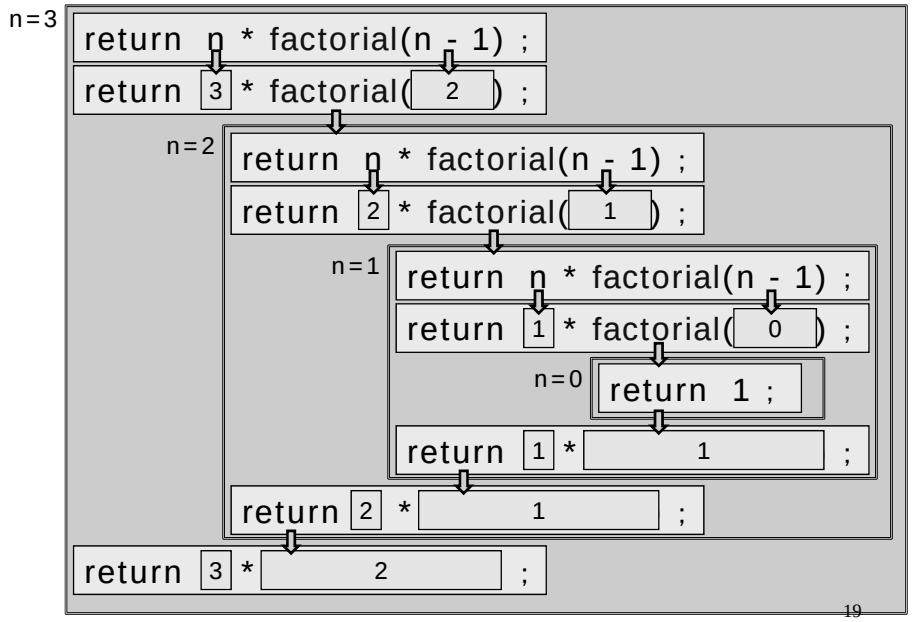
イメージ

再帰関数は、自分自身のコピーに仕事を押し付ける。



18

再帰的関数の呼び出しを含む式の計算



19

終わらない再帰関数

再帰関数は、必ず基礎（出口）部分に辿りつけないといけない。

再帰の基礎部分が無い再帰関数

```
/*終わらない再帰関数1*/
int    factorial(int n)
{
   return   n * factorial(n-1);
}
```

NG

再帰関数は、ちょっと注意を怠ると、すぐに終わらないプログラムになってしまうので注意する事。

プログラムが終わらないときには、プログラムを実行しているkterm上で、コントロールキーを押しながら、cキーを押す。（C－c）

出口に近づかない再帰関数

```
/*終わらない再帰関数2*/
int    factorial(int n)
{
   if (n==0)
     {
      return  1;
     }
   else
     {
      return  n * factorial(n);
     }
}
```

NG

20

再帰的に階乗を求めるプログラム

```
/*
   作成日:yyyy/mm/dd
   作成者:本荘太郎
   学籍番号:B0OB0xx
   ソースファイル:fact_rec.c
   実行ファイル:fact_rec
   説明:階乗を求める再帰的関数を用いて、与えられた値の階乗を計算する。
   入力:標準入力から0以上15以下の整数nを入力。
   出力:標準出力にn!の値を出力。n!は正の整数である。
*/
#include<stdio.h>

/*プロトタイプ宣言*/
int  factorial(int n);       /*階乗n!を求める再帰的な関数*/
int  scanSmallInt(int  limit);  /* 標準入力から小さな非負整数を読み込む関数 */

int    main()
{
   int    n;       /* 階乗を求めるべき値（入力）*/
   int    fac;     /* nの階乗の計算結果（n!）*/
```

21

```
/*  次のページに続く  */
```

```
/* 続き */

   printf("入力された値nの階乗 n! を計算します。¥n");
   printf("n=? ¥n");
   n = scanSmallInt(15);

   /* 入力値チェック */
   if(n==-1)
     {
       printf("nには0から15までの整数を入力してください。¥n");
       return  -1;
     }
   /* これ以降ではnは0から15までの整数 */
   /* nの階乗の計算 */
   fac = factorial(n);

   printf("%d!  =  %10d ¥n", n, fac);

   return 0;
}
/* main関数終了 */

/*  次のページに続く  */
```

22

```
/* 続き */
/*
   階乗を求める再帰的関数
   仮引数 n : 階乗を求める値（0以上15以下の整数値とする。）
   戻り値    : nの階乗(正の整数値)を返す。
*/
int   factorial(int n)
{
   /* 漸化式に基づく、階乗の再帰的定義 */
   if (n==0)
     {
      /*再帰の基礎。0! = 1 であることを利用。*/
      return  1 ;
     }
   else
     {
      /* 再帰呼び出し。n! = n*(n-1)!  であることを利用。 */
      return  n * factorial(n-1);
     }
}
/* 関数factorialの定義終了 */

/*  次のページに続く  */
```

23

```
/* 続き */
/*
   標準入力から小さな非負整数を読み込む関数
   仮引数 limit : 入力値として許す整数の上限値(正の整数)
   戻り値        : 入力された値が0以上limit以下ならばその値
                   (0以上limit以下の整数)。そうでなければ、-1を返す。
*/
int   scanSmallInt(int  limit)
{
   int   input;  /* 標準入力から入力された(0以上limit以下の)値 */

   scanf("%d", &input);
   if ( 0<=input && input <= limit )
     {
      return  input;
     }
   else
     {
      return  -1;
     }
}
/* 関数scanSmallIntの定義終了 */

/* 全てのプログラム(combi.c)の終了 */
```

24

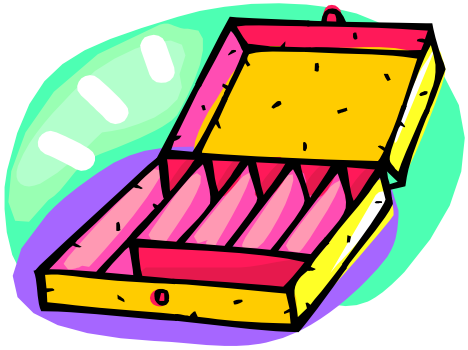
実行例

```
$make
gcc  fact_rec.c  -o       fact_rec
$ ./fact_rec
入力された値nの階乗 n! を計算します。
n=?
5
5!  =         120
$
```

```
$ ./fact_rec
入力された値nの階乗 n! を計算します。
n=?
-1
nには0から15までの整数を入力してください。
$
```

25

第12回構造体



1

今回の目標

- 構造体を理解する。
- 構造体の定義の仕方を理解する。
- 構造体型を理解する。
- 構造体型の変数、引数、戻り値を理解する。

☆複素数同士を足し算する関数を作成し、その関数を利用するプログラムを作成する。

2

複素数の足し算

複素数は実部と虚部の2つの実数で、
表現される。

$$z = a + bi$$

2つの複素数　$z_1 = a_1 + b_1 i$　と
$z_2 = a_2 + b_2 i$　の
和　$z_3 = a_3 + b_3 i$　は、次式で与えられる。

$$z_3 = z_1 + z_2$$
$$= (a_1 + a_2) + (b_1 + b_2)i$$

3

構造体

構造体とは、いくつかのデータを
1つのまとまりとして扱うデータ型。
プログラマが定義してから使う。
構造体型の変数、引数、戻り値等が利用できるよ
うになる。
（他の言語ではレコード型と呼ぶこともある。）

ひとまとまりのデータ例

複素数：実部と虚部	名刺：所属、名前、連絡先
点：x座標、y座標	日付：年、月、日、曜日
2次元ベクトル：x成分、y成分	本：題名、著者、ISBN

4

構造体型の定義
（構造体テンプレートの宣言）

宣言

```
struct  構造体タグ名
{
        型1    メンバ名1;
        型2    メンバ名2;
        型3    メンバ名3;
           :
};
```

構造体を構成する要素をメンバといいます。

これを
構造体テンプレートという。

int,double,char
や
int *,double *,char*
や
既に定義した構造体型等

例

```
struct complex
{
      double real;
      double imag;
};
```

関数の記述と似ているが
セミコロンを忘れずに。

5

構造体型の変数の用意の仕方
（構造体型の変数宣言）

宣言

```
struct 構造体タグ名      変数名;
```

ここに空白がある。

例

```
struct complex      z;
```

この2つで、一つの型を表わしているので注意すること。

参考

```
int     i;
double  x;
```

6

構造体のイメージ

既存の型

構造体テンプレート

```
struct complex
{
      double  real;
      double  imag;
};
```

セミコロンを忘れずに。

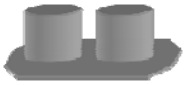
雛形の作成。

struct complex型の雛形

7

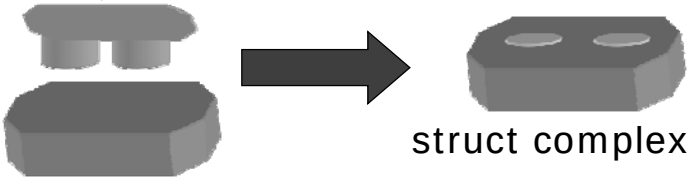
構造体型の変数宣言

```
struct complex  z1;
struct complex  z2;
```



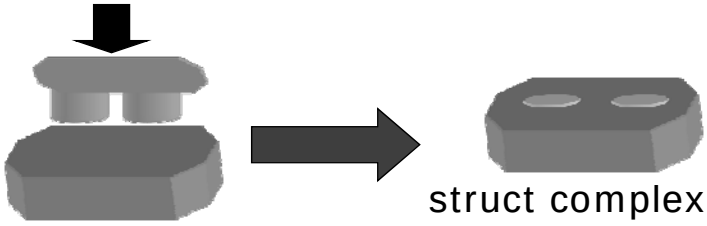
struct complex型の雛形

雛形を用いて、プレスする。



z1

struct complex型の変数



z2

struct complex型の変数

構造体のイメージ2


```
struct card
{
        char        initial;
        int         age;
        double      weight;
};
```

雛形の作成。

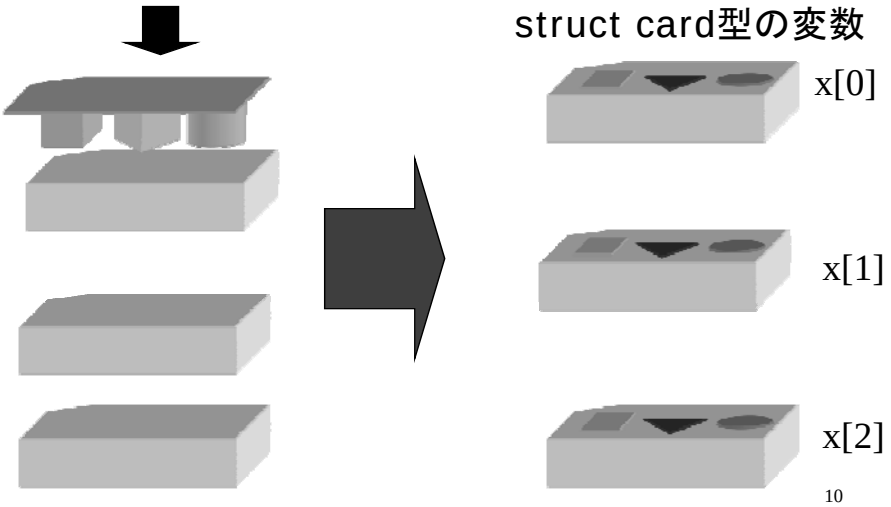
struct card 型の雛形

いろいろな型のデータを
ーまとまりであつかうときには、
構造体はとくに便利。

9

構造体型の配列宣言

```
#define MAXCARD   3
struct card    x[MAXCARD];
```



10

構造体のメンバの参照

struct 型の変数のメンバの参照の仕方

書式

変数名. メンバ名

ドット（演算子の一つ）

これらを、メンバ名を定義している 型の変数として扱える。

例

z1.real　　これはdouble 型の変数である。

x[0].inital　　これはchar 型の変数である。

11

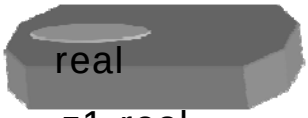
参照のイメージ

struct complex　　z1;

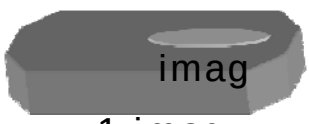
struct complex型の雛形

struct complex型の変数

z1

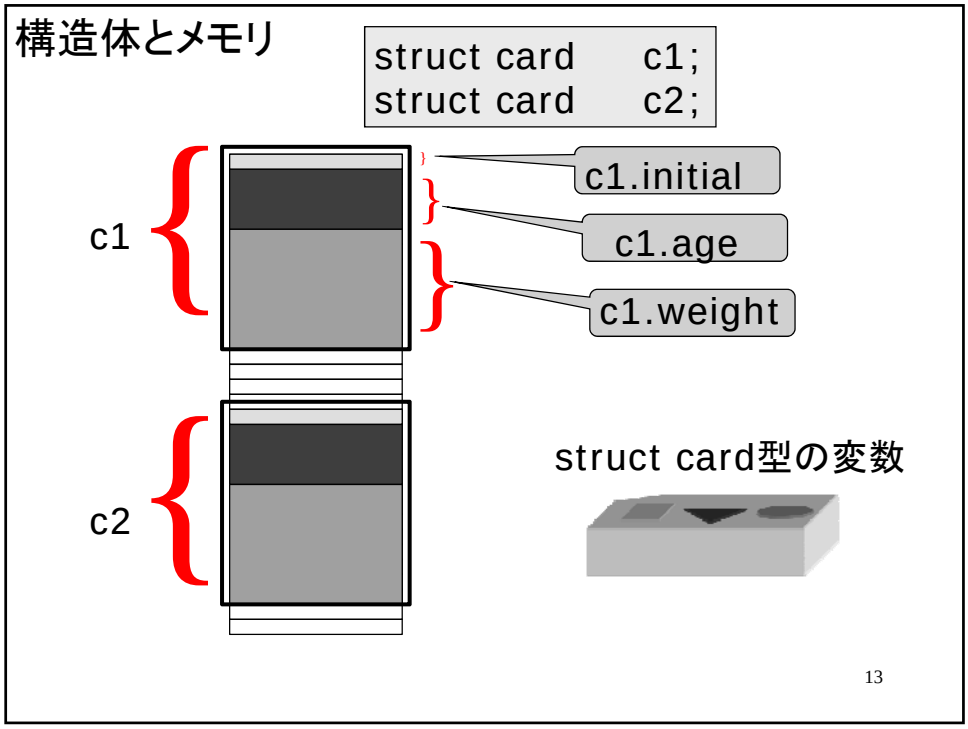


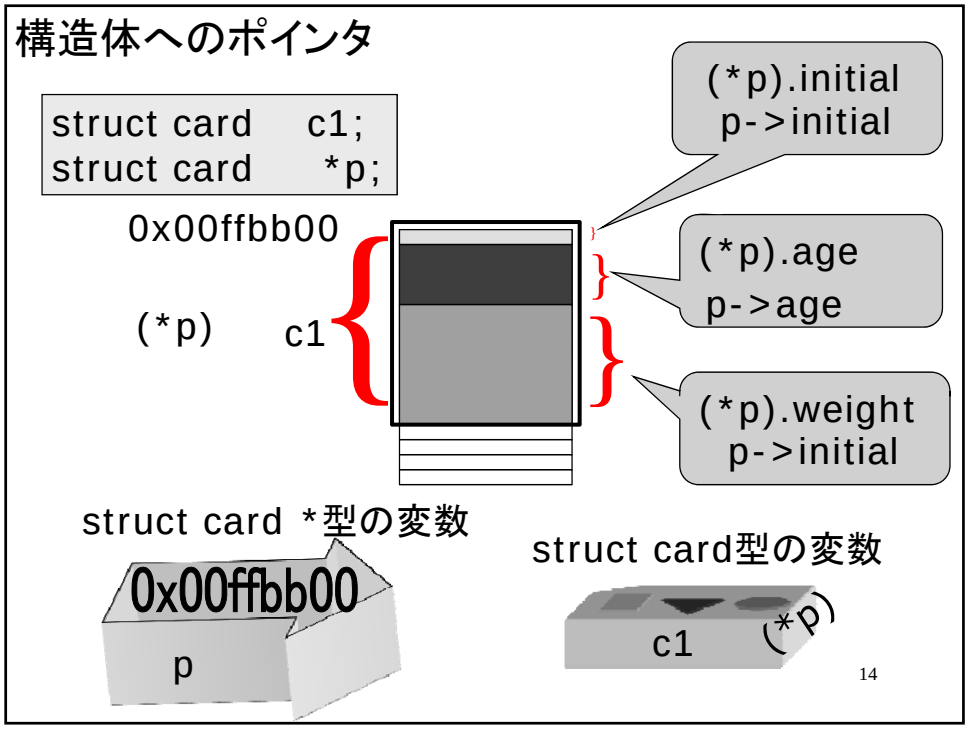
z1.real



z1.imag

12


構造体とメモリ
struct card c1;
struct card c2;
c1
c2
c1.initial
c1.age
c1.weight
struct card型の変数
13


構造体へのポインタ
struct card c1;
struct card *p;
0x00ffbb00
(*p)
c1
(*p).initial
p->initial
(*p).age
p->age
(*p).weight
p->initial
struct card *型の変数
0x00ffbb00
p
struct card型の変数
c1
(*p)
14

演算子.の結合力

演算子.の結合力は他のどの演算子よりも強い。

.(ドット演算子) ＞ ++ -- ＞ * &

x[0].age++; は (x[0].age)++; の意味

struct card * p; のとき、

*p.age; は *(p.age); の意味になってしまう

両方間違い。(メンバageは、ポインタではない。)

(*p).age; 型的には、これが正しい。

(ソースの可読性の向上のため)他の演算子と一緒に使うときには、括弧を用いて意図を明確にすること。

15

構造体と代入演算子1 (構造体への値の入れ方1)

全てのメンバに値を代入する。

```
struct complex  z1;

(z1.real)=1.0;
(z1.imag)=2.0;
```

OK

間違いの例

z1=1.0+2.0i; NG

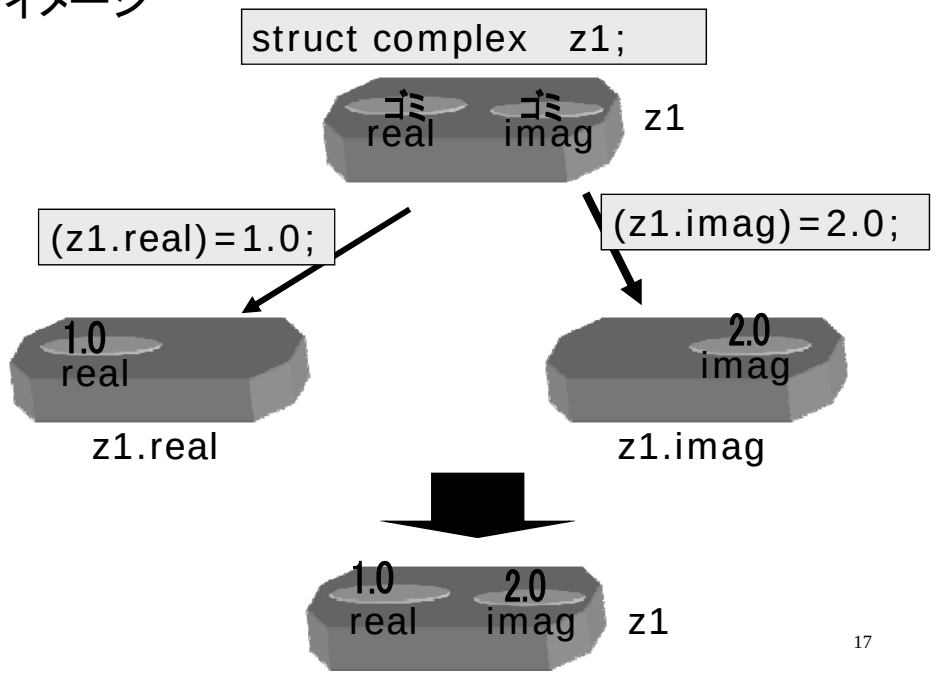
複素数だからってこんなふうにはかけない。

z1=(1.0,2.0); NG

ベクトル風にもかけない。

16

イメージ



17

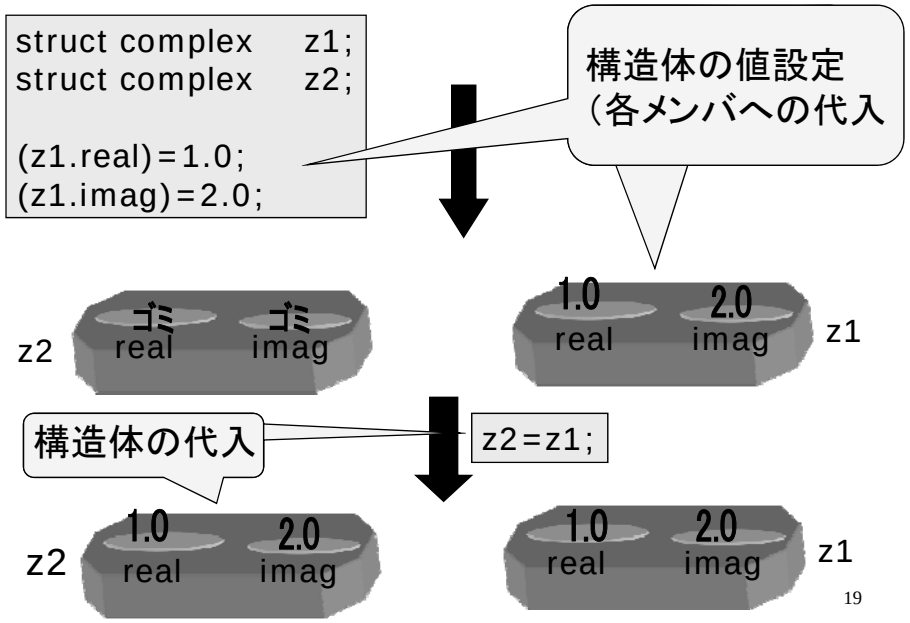
構造体と代入演算子2
（構造体への値の入れ方2）

同じ型の構造体同士で代入する。

```
struct complex      z1;
struct complex      z2;

(z1.real)=1.0;
(z1.imag)=2.0;

z2=z1;
```

18

イメージ



19

練習

```
/*test_struct.c   構造体実験    コメント省略*/
#include <stdio.h>

struct complex
{
      double  real;
      double  imag;
};

/*     次に続く      */
```

20

```
int    main()
{
       struct complex    z1;
       struct complex    z2;

       printf("メンバの読み込み¥n");
       printf("z1=  (real?) + (imag?)i    ");
       scanf("%lf", &(z1.real) );
       scanf("%lf", &(z1.imag) );

       printf("読み込み後¥n");
       printf("z1=%4.2f+(%4.2f)i¥n",
                          z1.real, z1.imag);
       printf("z2=%4.2f+(%4.2f)i¥n",
                          z2.real, z2.imag);

/* 続く*/
```

21

```
/*     続き    */
       printf("z2=z1実行中¥n");
       z2 = z1;

       printf("代入後¥n");
       pritnf("z1=%4.2f +(%4.2f)i¥n",
                          z1.real, z1.imag);
       pritnf("z2=%4.2f +(%4.2f)i¥n",
                          z2.real, z2.imag);

       return 0;
}
```

22

複素数の和を求めるプログラム

```
/*
        作成日:yyyy/mm/dd
        作成者:本荘太郎
        学籍番号:B00B0xx
        ソースファイル:addcomp.c
        実行ファイル:addcomp
        説明:構造体を用いて、2つの複素数の和を
                求めるプログラム。
        入力:標準入力から、2つの複素数z1とz2を入力。
                z1の(実部、虚部)、z2の(実部、虚部)の順
            で4つの実数を入力する。
        出力:標準出力にその2つの複素数の和を出力する。
                和は複素数であり、その実部と虚部は実数。
*/
/*続く*/
```

23

```
/* 続き */
#include <stdio.h>

/*構造体テンプレート定義*/
struct complex /*複素数を表わす構造体*/
{
        double  real;  /*実部*/
        double  imag; /*虚部*/
};
/* プロトタイプ宣言 */
struct  complex  scan_complex();
/*標準入力から複素数を読み込む関数*/

void  print_complex(struct complex  z);
/*標準出力へ複素数を出力する関数*/

struct complex  add_complex (struct complex z1,
                                        struct complex z2);
/*2つの複素数の和を求める関数*/
/*続く*/
```

24

```
/* main関数の定義 */
int     main()
{       /*ローカル変数宣言*/
        struct complex  z1;   /* 入力された複素数1 */
        struct complex  z2;   /* 入力された複素数2 */
        struct complex  sum; /* 二つの複素数の和 */

        /* 足し合わせるべき二つの複素数の入力 */
        z1 = scan_complex();
        z2 = scan_complex();

        /* 複素数の足し算 */
        sum = add_complex (z1, z2);

/*main関数続く*/
```

25

```
/*続き  main関数*/

        /* 計算結果の出力 */
        print_complex(z1);
        printf("+");
        print_complex(z2);
        printf("=");
        print_complex(sum);
        printf("¥n");

        /*正常終了*/
        return 0;
}
/*main関数終了*/
/*続く*/
```

26

```
/*続き */

/* 標準入力から複素数を受け取る関数。
   標準入力から実部、虚部の順にdouble 値を受け取る。
   仮引数 :なし
   戻り値:読み込まれた二つの実数値をそれぞれ実部、虚部とする複素数。
*/
struct complex  scan_complex()
{
        /*ローカル変数宣言*/
        struct complex  z;/*読み込まれる複素数*/
        /*入力処理*/
        scanf("%lf", &(z.real));     /*実部*/
        scanf("%lf", &(z.imag));   /*虚部*/

        return z;
}
/*関数 scan_complexの定義終了 */

/*続く*/
```

27

```
/* 続き */

/*複素数を「( 実部+(虚部)i )」の形式で標準出力に出力する関数。
   仮引数 z:表示すべき複素数
   戻り値:なし
*/
void  print_complex(struct complex  z)
{
        /*出力処理*/
        printf(" (  %4.1f +(%4.1f)i )", z.real, z.imag);
        return;
}
/*関数 print_complexの定義終了 */

/*続く*/
```

28

```
/* 続き */
/* 2つの複素数の和を求める関数
   仮引数 z1,z2:2つの複素数。
   戻り値:2つの複素数の和(z1+z2)
*/
struct complex  add_complex (struct complex  z1,
                             struct complex  z2)
{
        /*ローカル変数宣言*/
        struct complex  sum;/*2つの複素数の和を蓄える*/
        /*計算処理*/
        (sum.real)=(z1.real)+(z2.real);     /*実部の計算*/
        (sum.imag)=(z1.imag)+(z2.imag);/*虚部の計算*/

        return sum;
}
/* 関数 add_complex の定義終了      */
/* プログラム addcomp.c  の終了 */
```

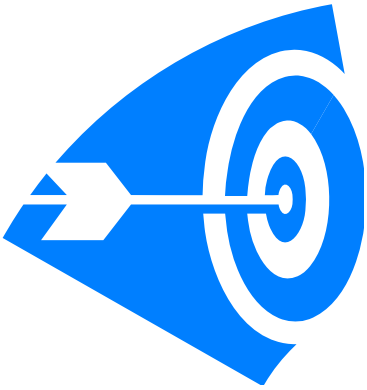
29

実行結果

```
$./addcomplex
2つの複素数z1,z2を入力して下さい。
( 4.0+( 6.0)i)=( 1.0+( 2.0)i)+( 3.0+( 4.0)i)
$
```

30

第13回 ポインタ



1

今回の目標

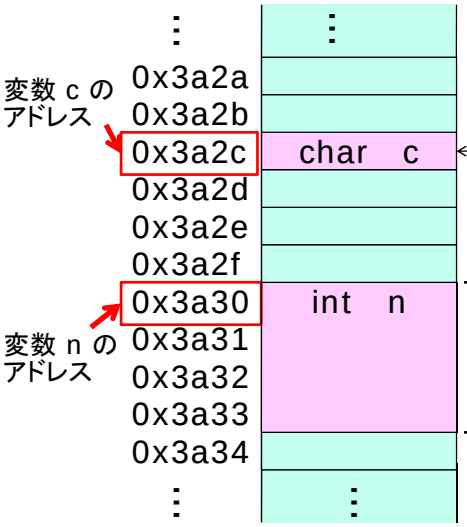
- C言語におけるポインタを理解する。
- 変数のアドレスを理解する。
- ポインタ型を理解する。
- アドレス演算子、間接演算子の効果を理解する。

☆他の関数で定義されたベクトルの和を求める関数を作成する。

2

変数とアドレス

ad･dress〔 ədrés 〕
住所、宛て先



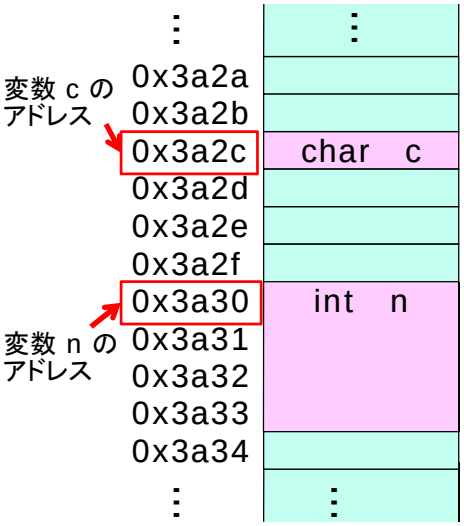
- メインメモリ内では、1バイトごとに一つのアドレスが割り当てられている。
- アドレスは16進数（0x****）で表される。
- ある変数に割り当てられた領域の先頭アドレスを、その変数のアドレスと呼ぶ。

char型は 1 バイト
int型は 4 バイト

3

アドレス演算子

&：変数に割り当てられたアドレスを求める演算子。
前置の単項演算子。



書式

& 変数名

左の例の場合、
&c は 0x3a2c
&n は 0x3a30

4

アドレスを表示するprintf文の変換仕様

printf文には、アドレスを表示するための変換仕様がある。

%p ⟷ アドレス

変数 a のアドレスを表示させる書式

```
printf("%p",&a);
```

アドレスを表示するために&をつける

変数 a の型に関わらず、
上の書式で a のアドレスが出力される。

5

練習1

```
/* address.c  アドレス表示実験     */
#include <stdio.h>
int main()
{
    /*変数宣言*/
    int n;
    double x;
    /*変数への値の代入*/
    n=1;
    x=0;
    /*変数のアドレスと値を表示*/
    printf("int型の変数nのアドレス: %p¥n", &n);
    printf("int型の変数nの値: %d¥n", n);
    printf("¥n");

    printf("double型の変数xのアドレス: %p¥n",  &x);
    printf("double型の変数xの値: %f ¥n",   x);
    return 0;
}
```

6

ポインタ

pointer〔pɔ́intə〕
指す人、助言、ヒント

ポインタとは、変数のアドレスを入れる変数である。

ポインタの型は
これまでのどの型(char,int,double)とも異なる。

7

ポインタと記号＊(その1):ポインタの宣言

変数のアドレスを入れるための変数(ポインタ)の用意の仕方。

宣言

```
データ型　＊ポインタの名前;
```

例

```
char *p;
```
文字型の変数のアドレス専用

```
int *q;
```
整数型の変数のアドレス専用

```
double *r;
```
実数型の変数のアドレス専用

ポインター＝変数のアドレスを入れるための入れ物
(ただし、用途別)

pは(char *)型の変数と考えてもよい。
char型と(char *)型は異なる型。

8

ポインタへのアドレスの代入

```
int    n;
int    *p;  /*ポインタ*/

p=(&n);
/*  p には n のアドレスが入る*/
```

ポインタpがある変数nのアドレスを蓄えているとき、
ポインタpは変数nを指すという。

あるいは、pは変数nへのポインタであるという。

9

ポインタと記号＊（その2）:間接演算子

ポインタに、変数のアドレスを代入すると、間接演算子＊でそのポインタが指す変数の値を参照できる。

書式

＊ポインタ

＊：ポインタに格納されているアドレスに割り当てられている変数を求める演算子。
前置の単項演算子

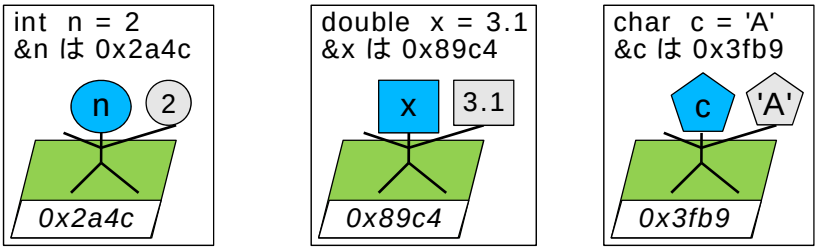
```
char  a;
char  b;
char  *p;

p=(&a);  /* pはaを指す。*/

b=(*p);  /*これは「b=a;」と同じ。*/

(*p)=b;  /*これは「a=b;」と同じ。*/
```

ポインタpがある変数yのアドレスを蓄えているとき、(*p)はあたかも変数yのように振舞う。

10

変数を住居に例えると...

- アドレスは住所
- 変数名は住人の名前
- 変数の値は住人の持ち物




```
n = 3;
```
は、「n さんに値 3 を渡して」

```
p = &n;
(*p) = 3;
```
は、「住所 (&n) の住人に値 3 を渡して」

11

練習2

```
/* test_pointer.c   ポインター実験  コメント省略     */
#include <stdio.h>

int main()
{
    /*変数宣言*/
    int  i;    /*整数が入る変数*/
    int  j;
    int  *p;  /*アドレスが入る変数(ポインタ)*/
    int  *q;

    /* 変数への値の代入 */
    i=1;
    j=2;

/* 次へ続く  */
```

12

```
/*続き*/
    /*実験操作*/
    p=(&i);  /* ポインタpへ変数iのアドレスを代入 */
    q=(&j);  /* ポインタqへ変数jのアドレスを代入 */

    printf("アドレス代入直後¥n");

    printf("iの中身は、%d¥n", i);
    printf("iのアドレスは、%p¥n", &i);
    printf("pの中身は、%p¥n", p);
    printf("pの指す変数の中身は、%d¥n", *p);
    printf("¥n");

    printf("jの中身は、%d¥n" ,j);
    printf("jのアドレスは、%p¥n", &j);
    printf("qの中身は、%p¥n", q);
    printf("qの指す変数の中身は、%d¥n", *q);
    printf("¥n¥n");

/* 次へ続く  */
```

13

```
/*続き*/

    /* ポインタを用いた、変数の値の操作 */
    (*q) = (*q)+(*p);
    printf("(*q)=(*q)+(*p); を実行¥n");
    printf("¥n");

    printf("iの中身は、%d¥n" ,i);
    printf("iのアドレスは、%p¥n", &i);
    printf("pの中身は、%p¥n", p);
    printf("pの指す変数の中身は、%d¥n", *p);
    printf("¥n");

    printf("jの中身は、%d¥n", j);
    printf("jのアドレスは、%p¥n" ,&j);
    printf("qの中身は、%p¥n", q);
    printf("qの指す変数の中身は、%d¥n", *q);

    return 0;
}
```

14

演算子＆と＊の結合力

演算子＆、＊の結合力は、算術演算子よりつよく、
インクリメント演算やデクリメント演算よりよわい。

++ -- ＞ &(アドレス演算子) ＊(間接演算子) ＞ / ＊(算術演算子) ＞ + −

*p++;		*(p++);	
	は		の意味
*p+1;		(*p)+1;	

1つの式内でインクリメント演算子と間接演算子を使うときには、
括弧を用いて意図を明確にすること。

15

関数とポインタ

関数の仮引数や戻り値として、アドレスを用いることができる。

書式

```
戻り値の型　関数名(仮引数の型　＊　仮引数名)
{
    return　戻り値;
}
```

```
void　func_ref(int　*p)
{
   return;
}
```

アドレスを渡して関数を呼び出す方法。
この場合の仮引数はpで、intへのポインタ型。
(int *)型であって、int型ではないことに注意。

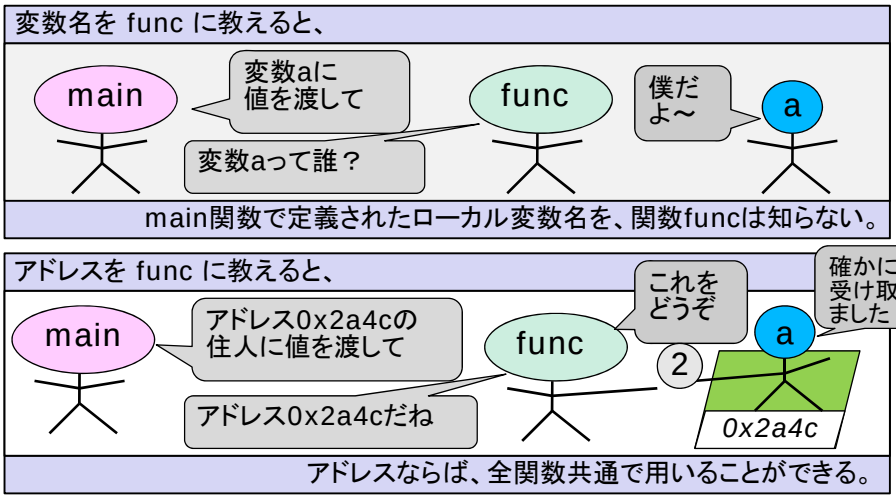
```
int main()
{
   func_ref(&i);
   a=i;
}
```

仮引数がポインタ型の場合、呼び出す際にはアドレスを指定しなければならない。

16

イメージ

main関数で定義された変数を、他の関数（func）で書き換えたい。



アドレスを受け取ることで、
他の関数内のローカル変数の内容を変更できる。

17

アドレスと scanf 文

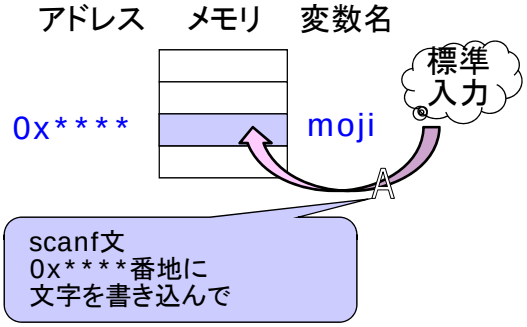
書式

```
scanf("%c", 文字を入れる変数のアドレス);
scanf("%d", 整数を入れる変数のアドレス);
scanf("%lf", 実数を入れる変数のアドレス);
```

例

```
char  moji;
scanf("%c",&moji)
```

&がついているので
アドレス

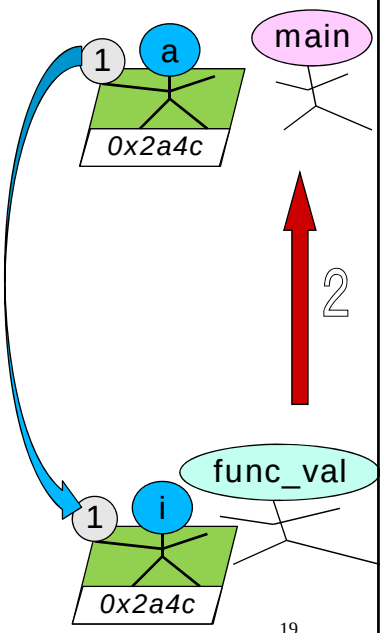


関数 scanf は、アドレスを渡されることで
変数の値を書き換えることができる。

18

値による呼び出し（call by value）

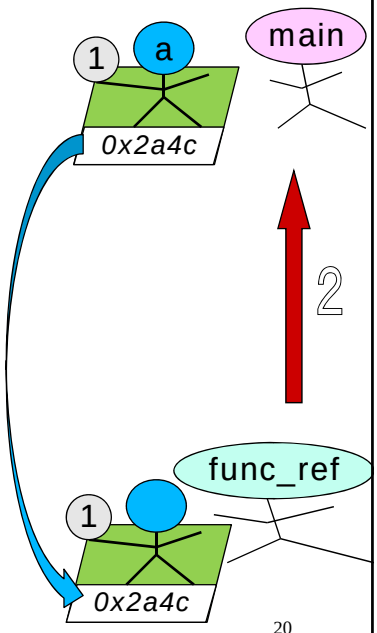
```
/*test_cbv.c 値による呼び出し実験*/
#include <stdio.h>
int  func_val(int);

int  main()
{
    int i=1;
    int j=0;

    printf("i=%d  j=%d¥n", i, j);

    j=func_val(i);

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

int  func_val(int   i)
{
    i++;
    return  i;
}
```



19

アドレスによる呼び出し（call by address）

```
/*test_cba.c  アドレスによる呼び出し実験*/
#include <stdio.h>
int  func_ref(int *p);

int  main()
{
    int i=1;
    int j=0;

    printf("i=%d  j=%d¥n", i, j);

    j=func_ref(&i);

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

int  func_ref(int   *p)
{
    (*p)++;
    return (*p);
}
```



20

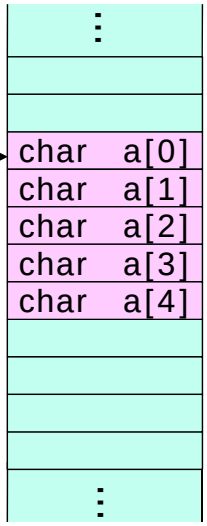
配列とアドレス

C言語では、配列名は先頭の要素のアドレスを指す。

例えば、

```
#define  MAX  5
char  a[MAX];
```

と宣言するとアドレスの連続した5個のchar変数がメモリ上に確保され、その先頭のアドレスがaにはいる。
つまり、aには「&a[0]の値（アドレス）」が保持されている。

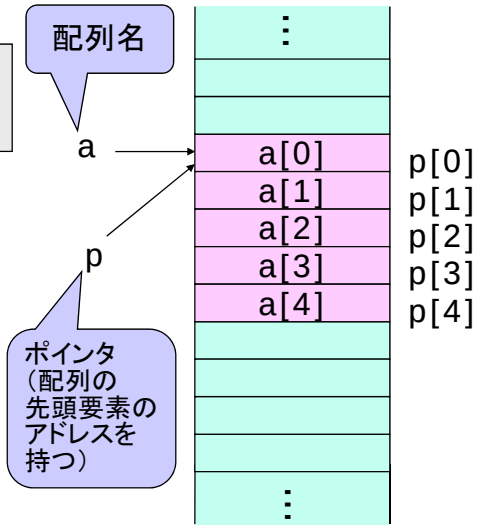


21

ポインタによる配列の別名

```
char a[MAX];
char *p;
p=a;
```

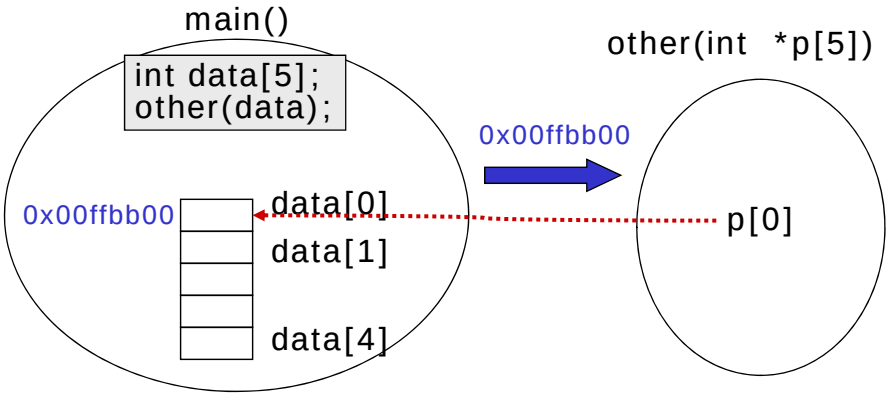
とすると、
a[i]とp[i]は
同じ変数（配列要素）を表す。



22

関数間での配列の引き渡し1

他の関数に配列(data[**])の
先頭要素のアドレス(data,すなわち、&data[0])を渡すことで、
配列全体を参照可能な状態にできる。

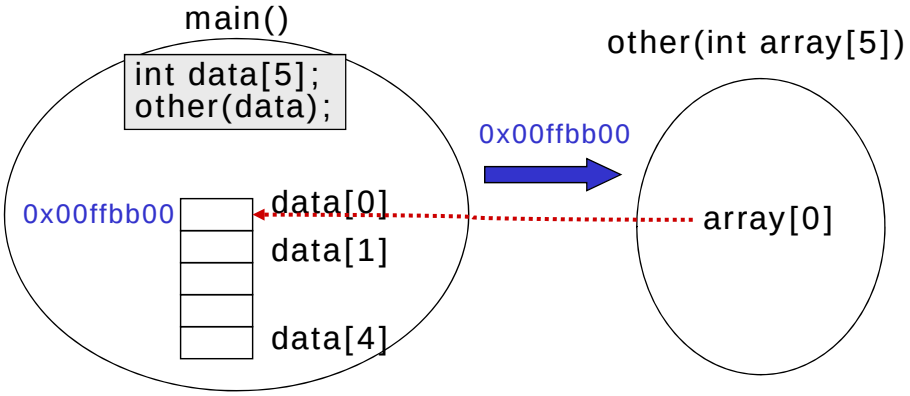


main 関数で定義された配列 data に、
関数 other では別名として p が与えられていると考えてもよい。

23

関数間での配列の引き渡し2

受け取る側では、仮引数に配列を記述しても良い。
この場合、引き渡されたアドレスが、引数の配列要素の先頭
アドレスになる。
(すなわち、「array=data」の代入が行なわれる。)

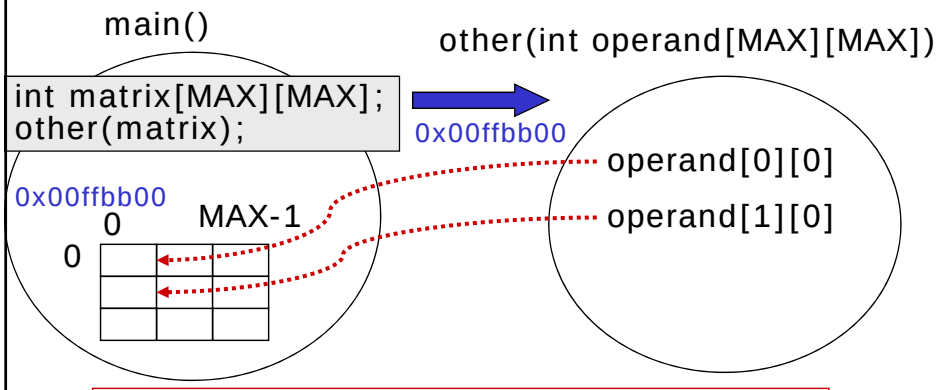


注意:
呼び出し側の配列の内容が書き換わるかもしれない。
十分に注意して関数を設計すること。

24

関数間での2次元配列の引き渡し

2次元配列では、先頭アドレスの他に大きさも指定する必要がある。



注意:
呼び出し側の配列の内容が書き換わるかもしれない。
十分に注意して関数を設計すること。

25

他の関数で定義された配列の和を計算する関数

```
/*
  作成日:yyyy/mm/dd
  作成者:本荘太郎
  学籍番号:b00b0xx
  ソースファイル: add_vector.c
  実行ファイル: add_vector
  説明:2つのN次元ベクトルの値を標準入力から受け取り、その和を出力する。
  入力:標準入力から N 個の実数値を2回受け取り、
       それぞれN次元ベクトル a, b の要素とする。
       aの要素が全て入力されてからbの要素が入力される。
  出力:標準出力に、N次元ベクトル a, b の和を出力する。
*/
#include <stdio.h>
#define  N  3   /* ベクトルの次元数 */

/* 標準入力からN次元ベクトルの値を取り込む関数  */
void  scan_vector(double  *p);
/* 標準出力にN次元ベクトルの値を出力する関数  */
void  print_vector(double  *p);
/* N次元ベクトルの和を計算する関数 */
void  add_vector(double  *p, double  *q, double  *r);

/* 次のページに続く */
```

26

```
/* 続き */
int  main()
{
   /* main関数のローカル変数宣言 */
   double  a[N]; /* 最初に入力されたベクトルの値 */
   double  b[N]; /* 次に入力されたベクトルの値 */
   double  c[N];  /* ベクトルaとベクトルbの和 */

   /* N次元ベクトル a, b の値を入力 */
   printf("%d次元ベクトル a の値を入力してください¥n", N);
   scan_vector(a);
   printf(" %d次元ベクトル b の値を入力してください¥n", N);
   scan_vector(b);

   /* N次元ベクトルの和を計算 */
   add_vector(a, b, c);

   /* 計算結果を出力 */
   printf("a+b=");
   print_vector(c);
   printf("¥n");
   return 0;
}
/* main 関数終了。次に続く。 */
```

27

```
/* 続き */

/* 標準入力からN次元ベクトルの値を入力する関数
  仮引数 p : N次元ベクトルの値を保持する配列の先頭アドレス
  戻り値なし
*/
void  scan_vector(double  *p)
{
   /* ローカル変数の宣言 */
   int  i; /* ループカウンタ、ベクトルを表す配列の添え字 */

   for(i=0; i<N; i++)
    {
      scanf("%lf", &p[i]);
    }
   return;
}

/* 次に続く */
```

28

```
/* 続き */

/* 標準出力にN次元ベクトルの値を出力する関数
  仮引数 p : N次元ベクトルの値を保持する配列の先頭アドレス
  戻り値:なし
*/
void  print_vector(double  *p)
{
   /* ローカル変数の宣言 */
   int  i; /* ループカウンタ、ベクトルを表す配列の添え字 */

   printf("(%6.2f", p[0]);
   for(i=1; i<N; i++)
    {
      printf(",%6.2f",p[i]);
    }
   printf(")");
   return;
}

/* 次に続く */
```

29

```
/* 続き */

/* N次元ベクトルの和を計算する関数
  仮引数 p, q : 被演算項を保持する配列の先頭アドレス
  仮引数 r : 演算結果のベクトルの値を保持する配列の先頭アドレス
  戻り値:なし
*/
void  add_vector(double  *p, double  *q, double  *r)
{
   /* ローカル変数の宣言 */
   int  i; /* ループカウンタ、ベクトルを表す配列の添え字 */

   for(i=0; i<N; i++)
    {
      r[i] = p[i] + q[i];
    }
   return;
}

/* プログラム終了 */
```

30