

第1回プログラミング入門

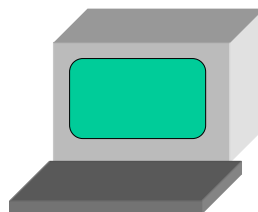
(教科書1～3章)



1

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。



2

今回の目標

- 現実の問題をプログラムにするまでの概要を理解する。
- 課題の提出法を習得する。

☆演習室から課題を提出する。

3

コンピュータの2つの側面

ソフトウェアとハードウェア

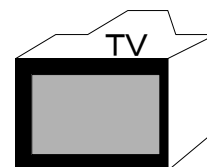
ソフトウェア

プログラム

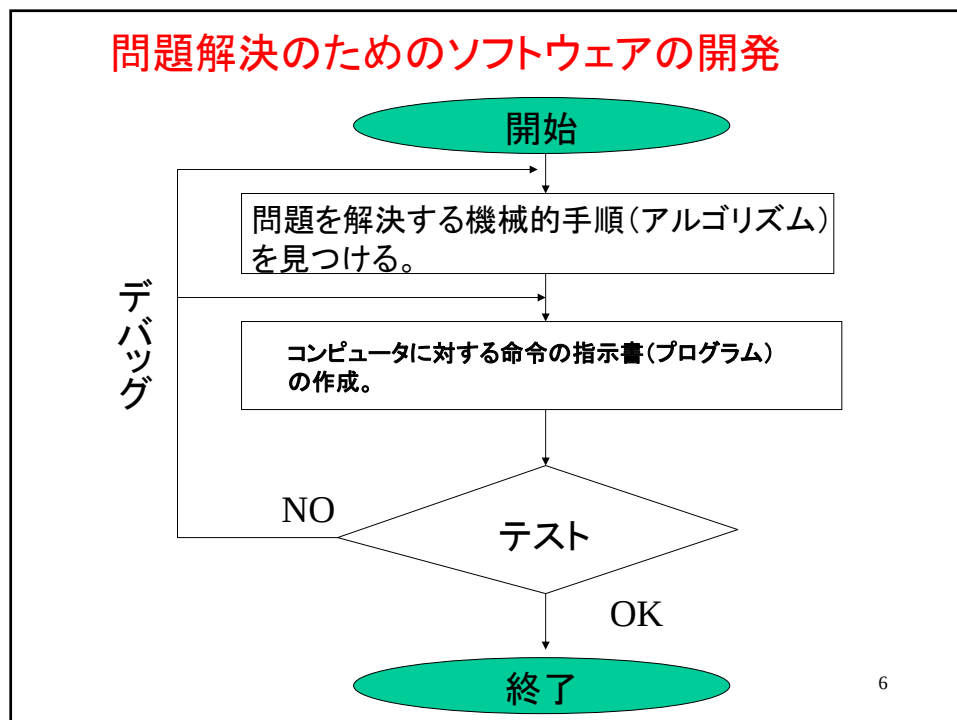
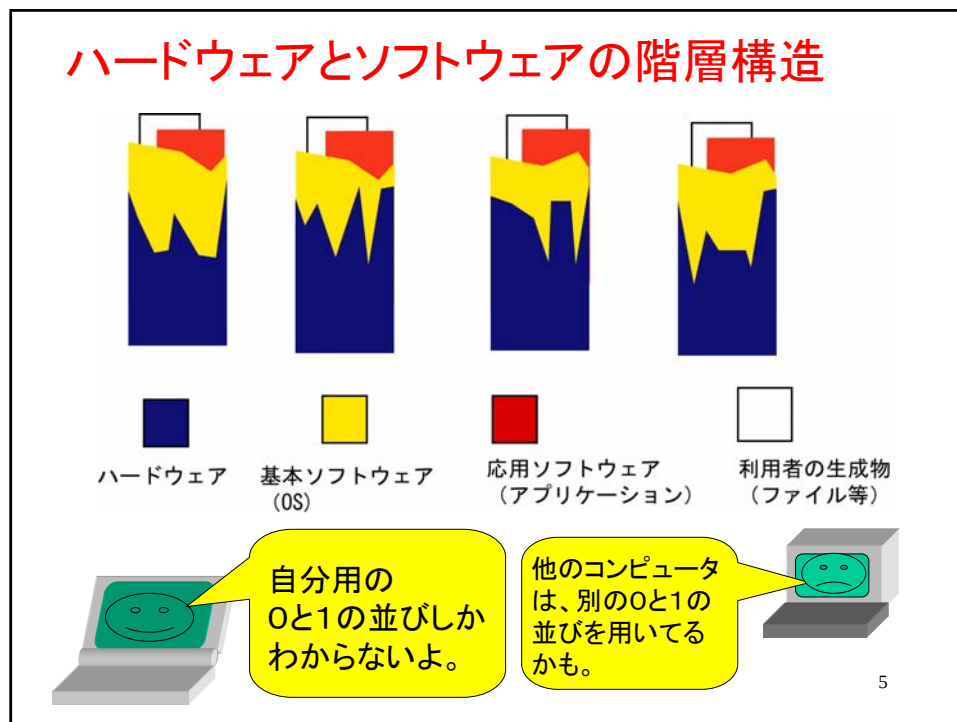
番組

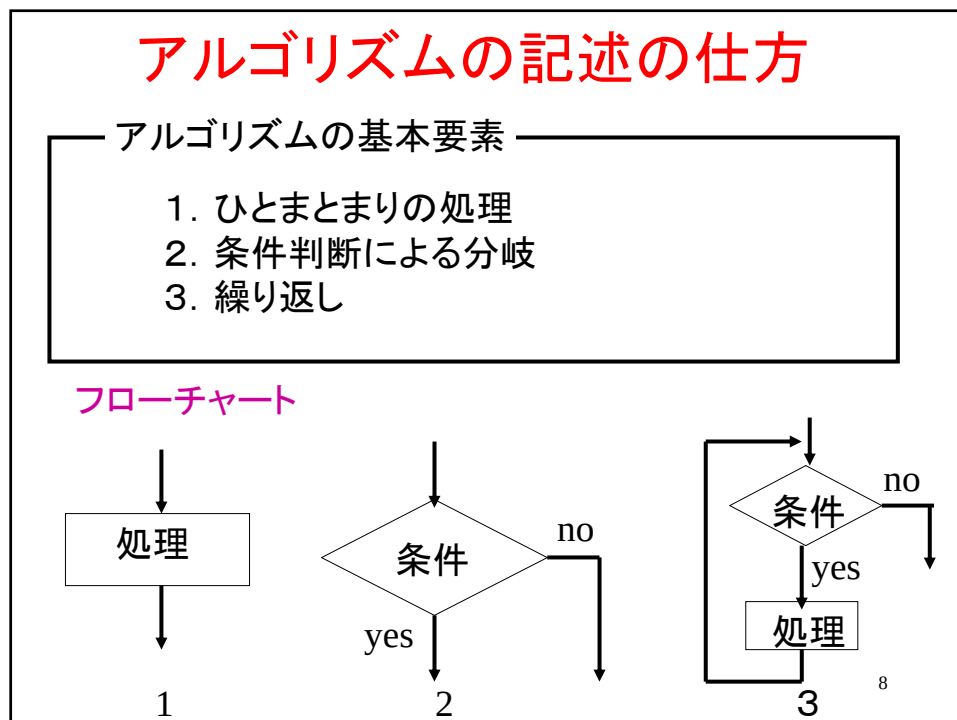
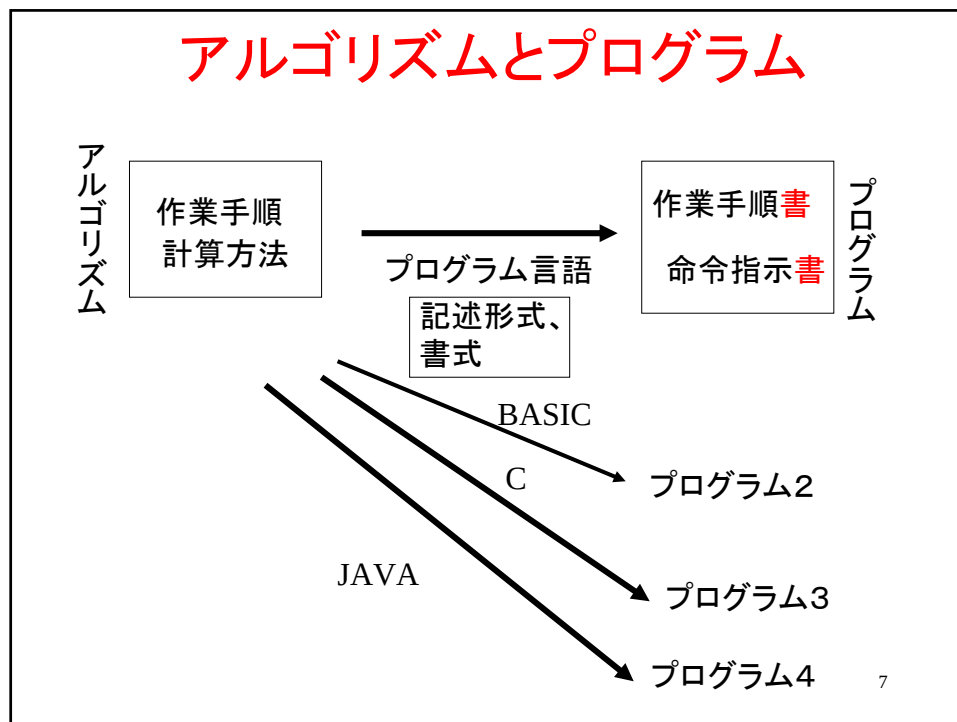
思考、記憶

ハードウェア

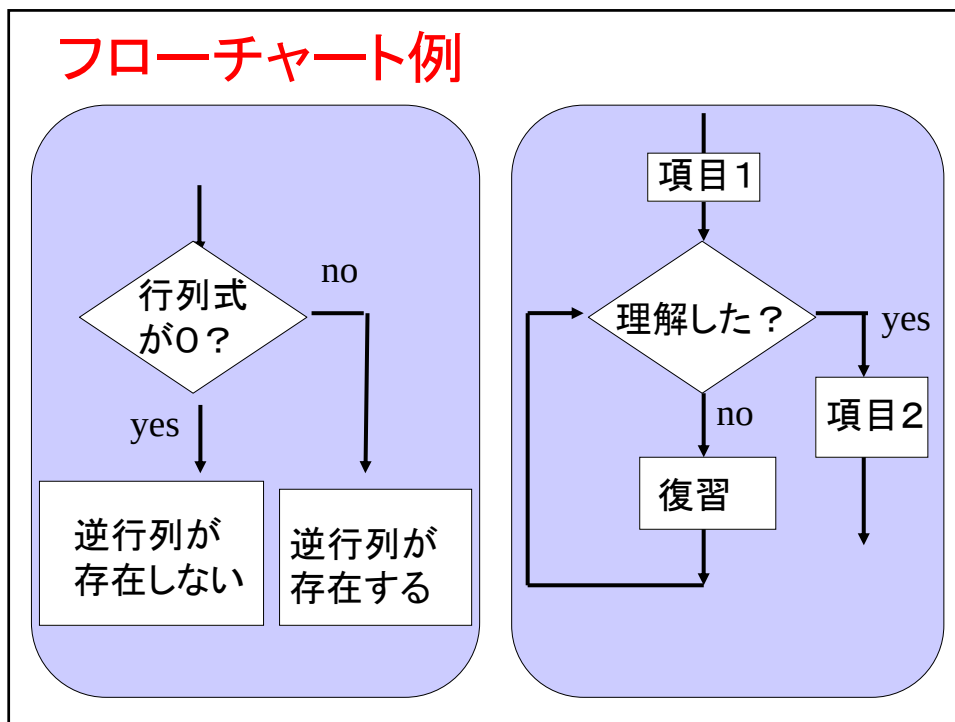


4

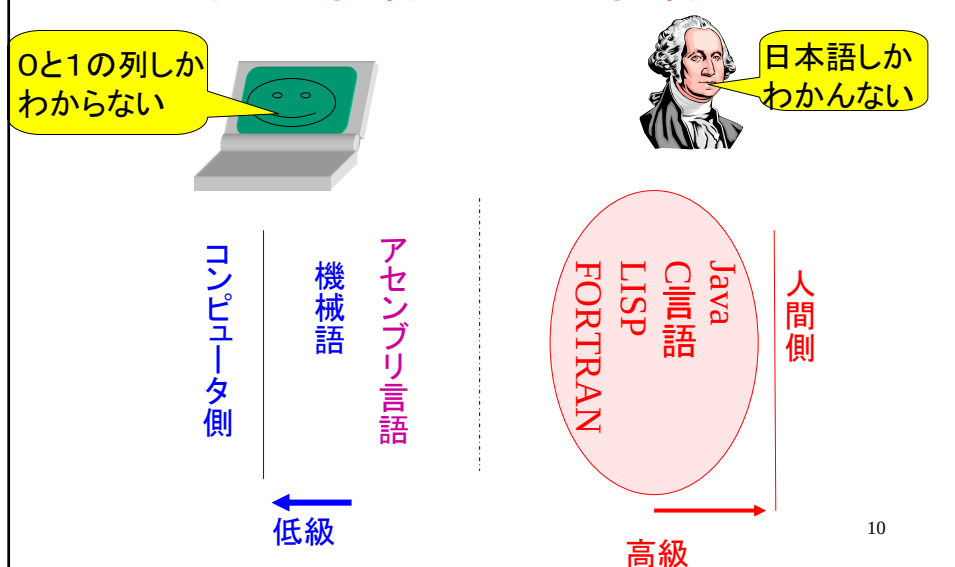




フローチャート例

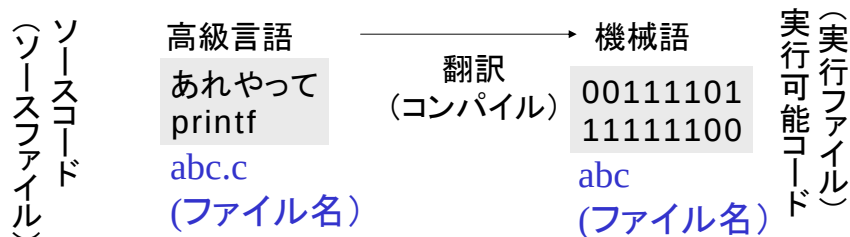


プログラミング言語の分類 (高級言語と低級言語)



10

コンパイラ



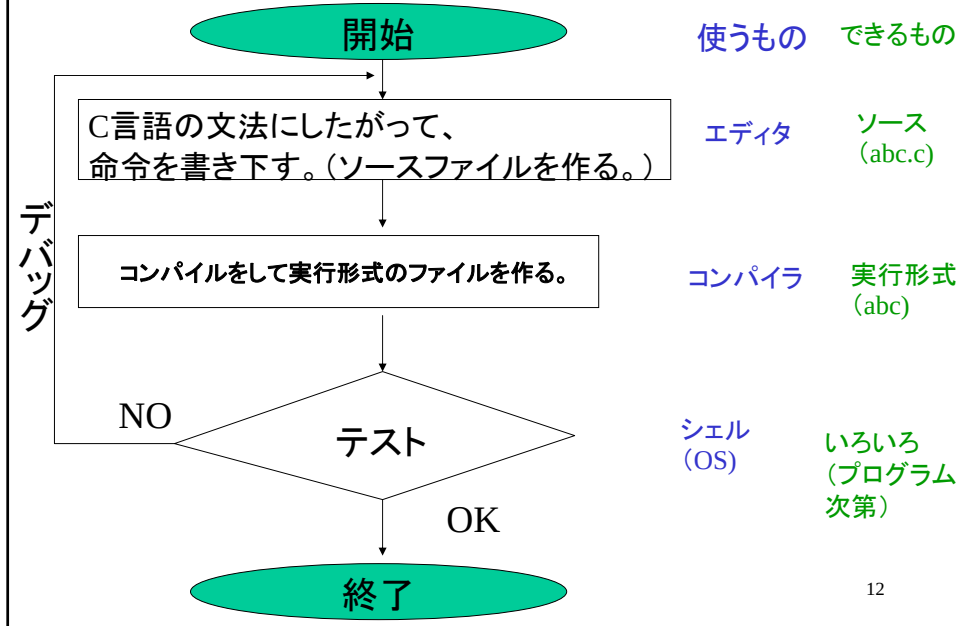
コンパイラ:

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。)

11

C言語でのプログラムの作り方



12

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。

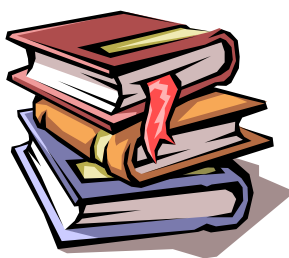
- 分かりやすい

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)₃

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)
- 本演習のスタイル規則に慣れる。

☆コンパイル可能で適切に実行できる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

[規則] C言語のソースファイルは、**main**という名前の関数が必要。

[規則] 括弧が重要 (括弧の種類も含めて)

[規則] 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である。
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「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
```

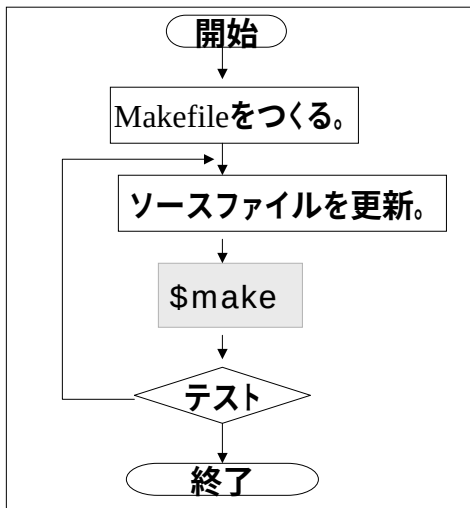
4

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)

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

Makefile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、以下のようにかければ良い。

```
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
```

```
$ gcc shortest.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A1参照)

```
/*
    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: tribial.c
    実行ファイル: tribial
    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;
}
```

世界1有名なCのプログラム (教科書p.31)

```
/*
    The most famous program written in C
    hello.c
*/

#include <stdio.h>

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

8

本演習スタイル版

```

/*
    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: 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;
}

```

9

Cの規則

コメント

```

/* The most famous program written in C */
/* hello.c */

```

[規則] コメントは「/*」と「*/」で囲む。

いろんなコメント

```

/*
課題 T02-1
ename.c
*/

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

```

間違っているコメントの例

正解

```

/*    正しいコメントです。    */
/* 間違いです。コンパイルできません。  */

```

間違い

11

関数

[規則] プログラムは**関数**で構成される。

入力 → **関数** → 出力

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 (例えば整数値)

関数書式

```

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

```

入力

出力

詳しくは、第9回で説明する。

12

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。
(教科書では省略されている。)

main関数の戻り値型：
本演習では省略しない。

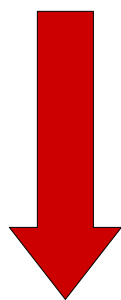
```
int main()
{
    return 0;
}
```

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。

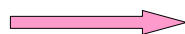
(スタイル規則参照)

13

C言語でのプログラムの実行順序



```
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、左から右に実行される。

14

多入力の関数



関数書式

戻り値型

```

戻り値の型 関数名 (型1 引数1、型2 引数2、...)
{
  関数の本体
  return 戻り値;
}
  
```

注意:
関数の出力 (戻り値) は必ず1つである。

詳しくは、第9回で説明する。

15

標準入出力

とりあえず、標準入力とはキーボード、標準出力とはディスプレイ (端末) の事だと思つとよい。

注意: 関数の入力 (引数) や、出力 (戻り値) とは無関係なので、注意して下さい。

もし、上記以外を標準入力、標準出力としたいときは、以下のようにリダイレクションを使います。

`/実行ファイル名 < 標準入力 (ファイル名) > 標準出力 (ファイル名)`

```
$. /hello > output
```

```
$lv output
Hello, world !
```

まだ、入力があるプログラムについては説明しない。
詳しくは次回以降で説明します。

16

悪いプログラム例1

[規則]文は「;」で終る。

コンパイルできないプログラム

```
#include <stdio.h>

/* 間違ったプログラム */
/* By kusakari */

#include <stdio.h>

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

「;」がないので間違い。
このままでは、
実行ファイルができない。

17

悪いプログラム例2

[規則]Cには**行**の概念がない。

```
#include <stdio.h>
/* さっきと同じプログラム */ /* By Honjo */ int main() {
printf("hello,world¥n");return 0;}

```

このプログラムは、コンパイルはできるが、
読みにくい。上のようなプログラムは、**悪い見本**。

改行は、適切に行なって、読みやすいプログラムに
すること。

18

複数の命令文と字下げ (インデント)

```
#include <stdio.h>
int main()
{
    printf("hello,");
    printf("world¥n");
    printf(" By kusakari¥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。)

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。
(スタイル規則参照)

19

エスケープ文字

[規則] ¥nはエスケープ文字である。

エスケープ文字列集。(教科書p.34)

¥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

逆の言い方をすると、
¥ (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。

20

文字の表現 (JISコード)

(教科書2章参照。 p.19)

[規則]文字を数字で指定できる。

¥ + '8進数'

¥x + '16進数'

'A' = ¥201 = ¥x41

21

ソースファイルにおける日本語の扱い

[規則]

日本語 (全角文字) は、
コメント内あるいは
printf文の「`"`」 (ダブルクォーテーション) で囲まれた内部
以外で用いないこと。

注意:

特に、全角スペースを上記以外の部分に書かないこと。
正しくコンパイルできない。

22

プリプロセッサへの指示

[規則]

#で始まる行は**プリプロセッサ**に指示を与える。

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

(7章のプリプロセッサの部分をよく読むこと。)

23

第3回簡単なデータの入出力



1

今回の目標

- 変数を理解する。
- 変数や定数の型を理解する。
- 入出力の方法を理解する。

☆入出力のあるプログラムを作成する。

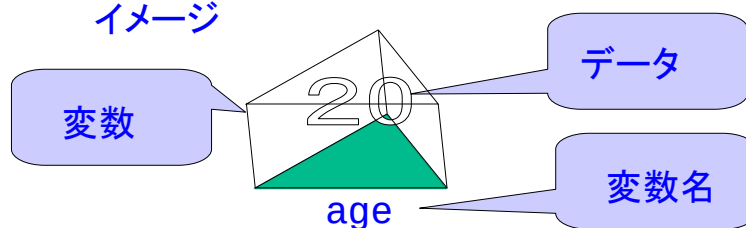
2

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



3

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。
特に、整数と実数も区別する。
(型という考え方が生じる)

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。

4

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B参照)

変数例

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。

(スタイル規則B-1参照)

main内で宣言する変数は英小文字と数字

_ (アンダースコア) だけを用い英大文字は用いない事。

(スタイル規則B-2参照)

5

予約語(キーワード)

予約語

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。

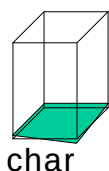
**注意: 変数名や関数名に予約語を用いてはいけない。
(予約語以外で、命名する。)**

6

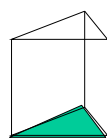
C言語の代表的なデータ型

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。

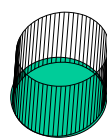
char	1バイトの整数型(1つの文字を表す型)
int	整数型
float	単精度浮動小数点型
double	倍精度浮動小数点型



char



int



double

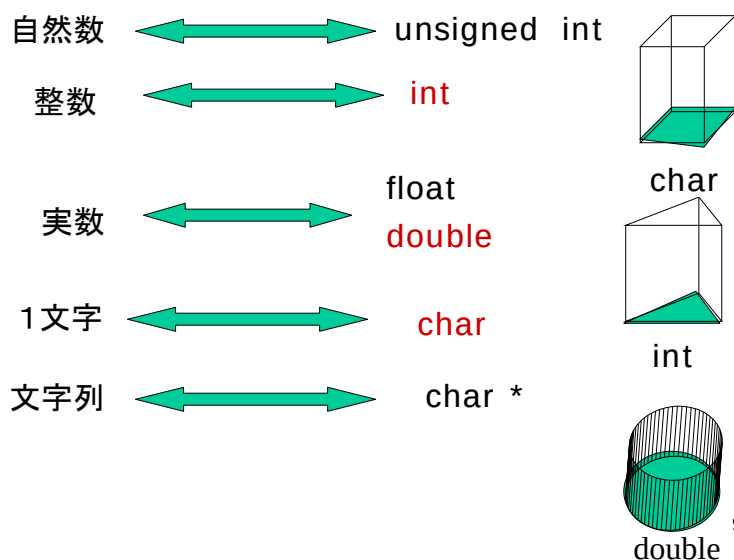
7

C言語のデータ型が扱える範囲

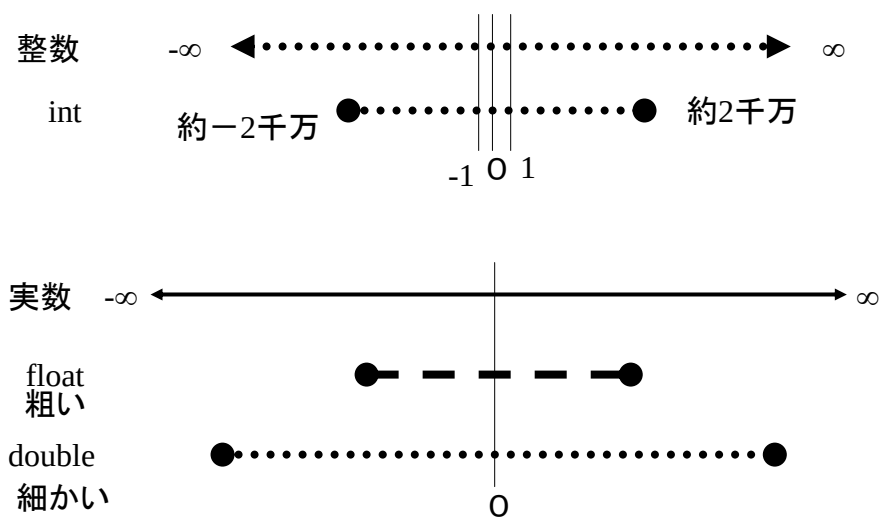
型	バイト長	表現範囲
char	1	0~255
int	4	-2147483648~2147483647
short int	2	-32768~32767
unsigned int	4	0~4294967295
float	4	$\pm 1.0 \times 10^{-37} \sim \pm 1.0 \times 10^{38}$
double	8	$\pm 1.0 \times 10^{-307} \sim \pm 1.0 \times 10^{308}$

8

数学の概念とC言語の型



数学とC言語の型の違い



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

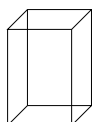
体重、
温度、等

明確に区別する
こと。

11

文字と文字列

char



char型には、半角文字1文字だけを保存できる。

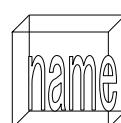
char



char



char



char *



C言語では、文字列は
終端文字¥0
で終わる。

12

変数宣言(1)

[規則] 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

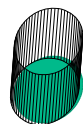
```
int age; /* 年齢 */
```

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。

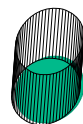
宣言の際には、必ずコメントを付けること。
(スタイル規則参照)

13

変数宣言(2)(同型複数の変数宣言)



x_pos



y_pos

変数宣言例2

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

14

変数宣言(3)(異なる型の変数宣言)

変数宣言例3

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

15

宣言場所(1)

[規則] 変数宣言は、関数の最初でまとめて行う。

典型的なmain関数

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

    /* 変数初期化 */
    .....
}
```

16

宣言場所(2)(不正な宣言)

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

    /*演算1*/
    f1=0;
    .....

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



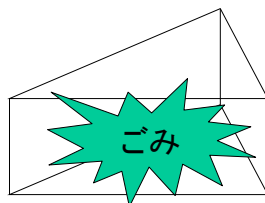
なお、main関数先頭以外での変数宣言については、第9回で説明。

17

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。

age



18

変数への代入(1)

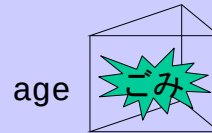
典型的なmain関数

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

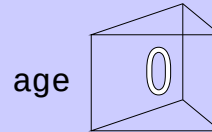
    /*変数への代入*/
    age=0;

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

この段階での状態



この段階での状態



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

19

変数への代入(2)

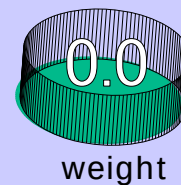
double型への代入

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

    /*変数への代入*/
    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)



20

変数の中身の表示

printf文を用いて、表示できる。

” ”間の文字列に特別な文字列を挿入して、
, (カンマ)を書き、
その後に変数名を書けばよい。

“変換仕様” =
“%” + “変換文字”

典型的な表示書式

```
printf(“.....%変換文字.....”, 変数名);
```

21

printf文

```
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

%変換文字 printf文の文字列内の%変換文字(変換仕様)は、
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

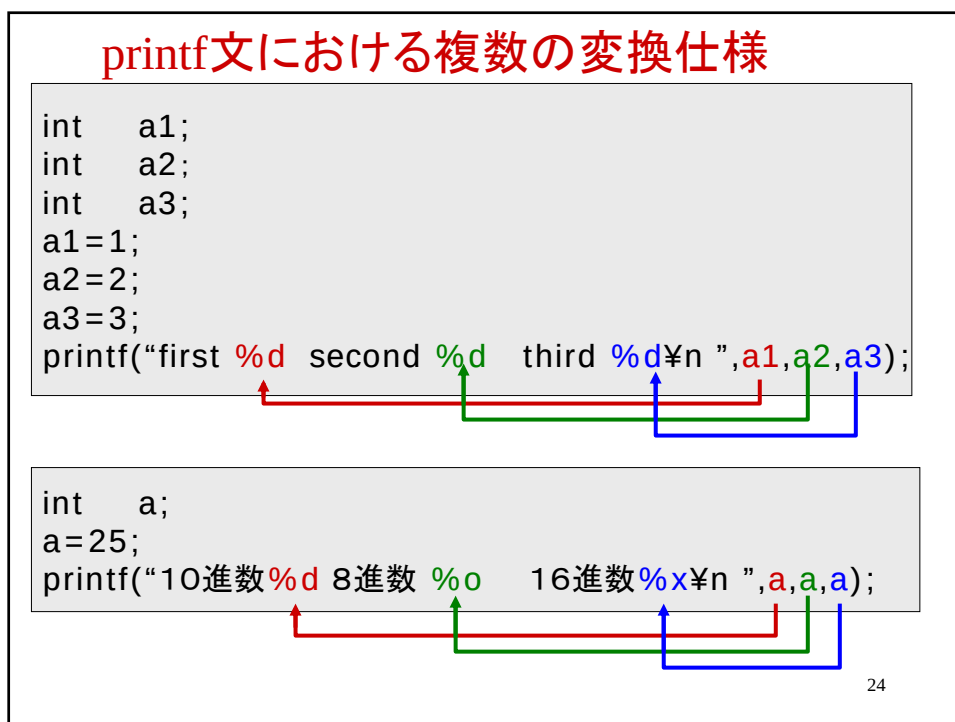
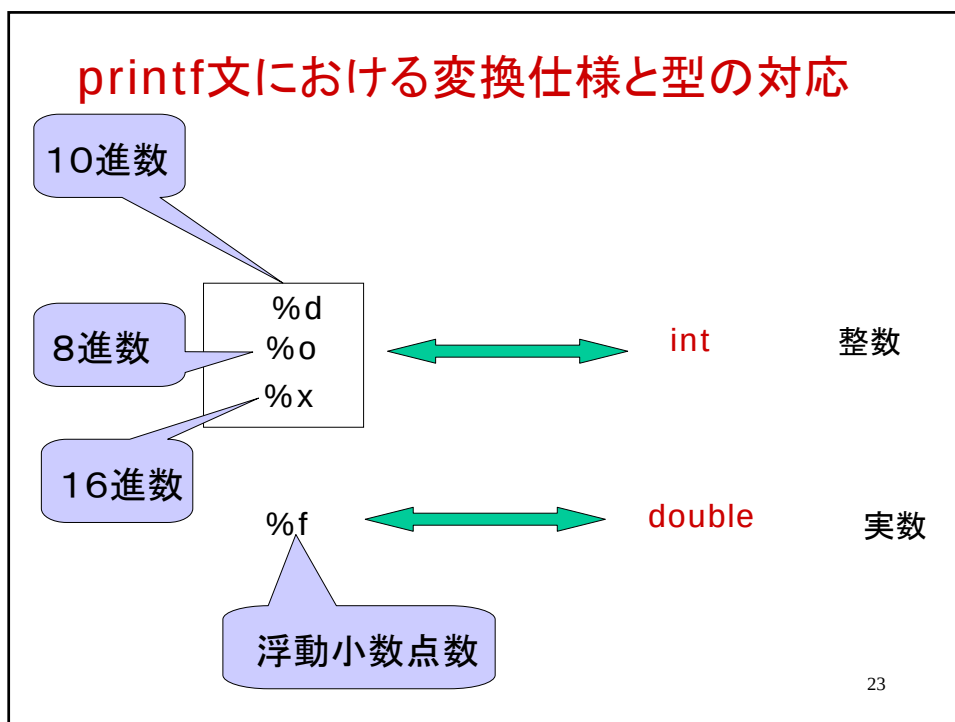
(double用)

%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

22



printf文に関するよくある間違い

```
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

25

練習1

```
/* 変数の中身表示実験 print_val.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;
}
```


変数に値を代入する (プログラム実行後)

演算子 '=' によって、
変数に値を代入するには、
ソースコード内に記述しないといけない。

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。

典型的な代入書式

```
scanf("%変換文字",&変数名);
```

27

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

%変換文字 scanf文の” ”内の%で始まる文字は変換仕様である。
scanfの” ”内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、第11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
相違に注意が必要。

28

scanf文における変換文字と型の対応

10進数

%d



int

整数

%lf



double

実数

浮動小数点数
(printf文の変換仕様との
違いに注意)

29

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);
```

30

scanf文に関するよくある間違い

```
int age;  
double weight;
```

```
scanf("%d",age);  
scanf("%lf",&weight);  
scanf("%d",&weight);  
scanf("%6.2lf",&weight);  
scanf("%1f",&weight);
```

&忘れ

カンマ忘れ

型の不一致

printf文用の
変換仕様の
誤用

'1'と'l'の違い

31

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。

32

練習2

```
/*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;
}
```

33

標準入力の変更 (ファイルから入力する)

実行

`$. /実行ファイル名 < 入力ファイル名
(データファイル)`

実行例

```
$. /test_stdio < test_stdio.in
1 3 5
$
```

データファイル
test_stdio.in

```
1
3
5
```

34

標準出力の変更 (ファイルへ出力する)

実行

```
$./実行ファイル名 > 出力ファイル名  
                (空ファイルでいい。)
```

実行例

```
./test_stdio > test_stdio.out  
2  
4  
6  
$lv test_stdio.out  
2 4 6
```

35

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  
1 3 5
```

36

入出力のあるプログラム例 (教科書p.36参照)

```
/*
    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: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:標準入力から年齢を入力する。
    出力:標準出力に年齢を出力する。
*/

/*    プログラム本体は次のページ    */
```

37

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

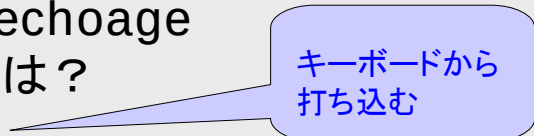
    /*    入力 処理    */
    printf("年齢は?¥n");
    scanf("%d",&age);

    /*    出力処理    */
    printf("年齢は %d 歳です。¥n",age);
    return 0;
}
```

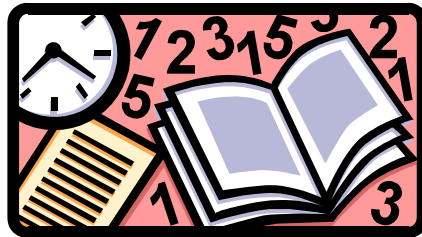
38

実行結果

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



第4回簡単な計算・プリプロセッサ



1

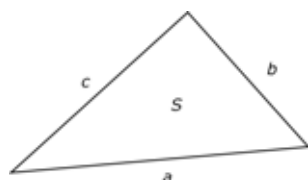
今回の目標

- 定数を理解する。
- 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た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。

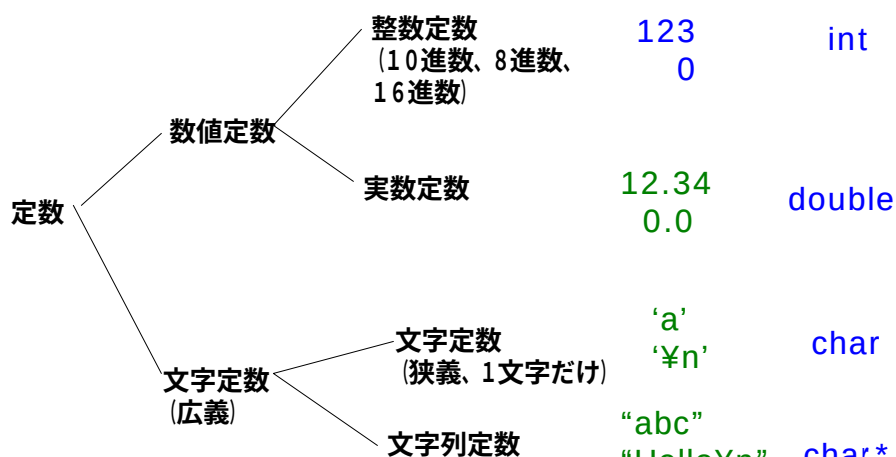
$$d = \frac{a + b + c}{2}$$

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

3

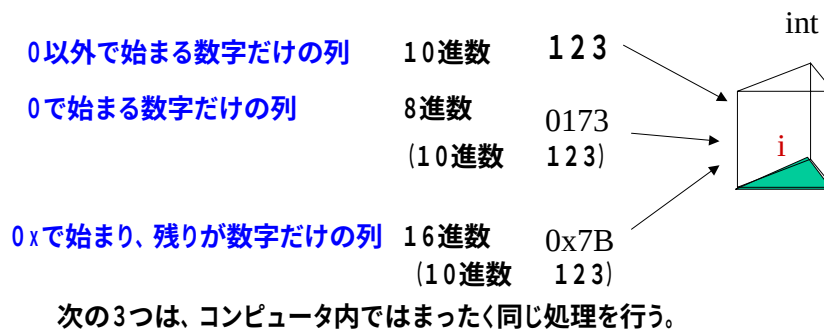
定数の分類と型

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



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

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

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

5

練習1

```
/* 整数定数実験 intconst.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;
}
```

プログラム (ソース) 内での 実数定数の表現

小数点を含む数字列

12.34

e、Eを含む形

1234e-2

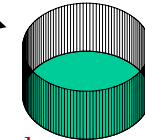
(意味 $1234 \times 10^{-2} = 12.34$)

e、Eと小数点を含む形

0.1234E+2

(意味 $0.1234 \times 10^{+2} = 12.34$)

double



d

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

```
/*実数定数実験 realconst.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)

演算子

式

変数、定数、それらの組み合わせ

単項演算子の書き方 (後置型)

式

演算子

二項演算子の書き方

式

演算子

式

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
	+	a+b	aの値とbの値の和
2項演算子 (四則演算とモジュロ演算)	-	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	$a \times b$	<code>edge1 * edge2</code>
	$a \cdot b$	
	ab (記号省略可)	<code>edge1edge2</code>

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	<code>edge1 * edge1</code>
	$a \wedge 2$	<code>pow(edge1, 2.0)</code>

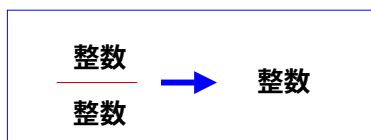
数学ライブラリ関数を用いる。

型に注意。教科書B. 6参照。
doubleのべき乗しか利用できない。

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。



```
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

結合規則と細則

[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。

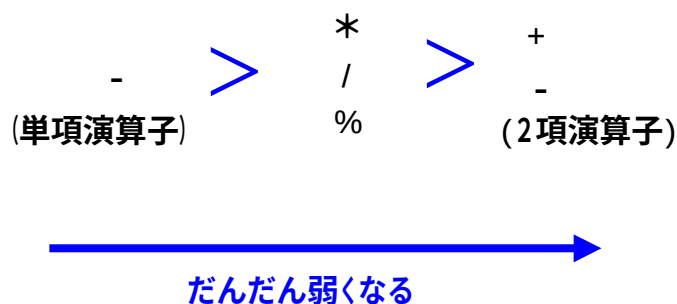
x

y

14

結合規則と細則

[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;` $\longrightarrow$ $x = (a/b) * c$ ○ $\times$
 $x = a/(b * c)$ とは違う。

`y = -a - b * -c;` $\longrightarrow$ $y = (-a) - (b * (-c))$ ○

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;
}
```


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)

変数=式

の形でつかわれる。

左辺は必ず変数で、右辺は式(定数や算術式等)。

`radius = 10.0;`



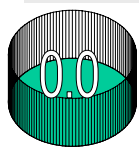
radius

10.0



radius

`area = 3.14 * radius * radius;`



area

3.14

×



radius

×



radius



area

19

型の表現能力

型の表現能力

低い ←

char

int

→ 高い

double

高い = 低い

`double = int;`



問題なし

低い = 高い

`int = double;`



切り捨てが起こる。

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E-3 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。

20

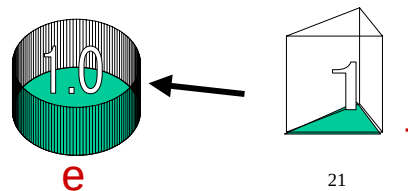
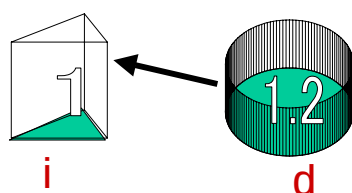
キャスト演算子 (型の変換法)

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;
```

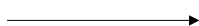


21

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)

22

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icolon「;」をつけない。

25

#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。

26

#define例

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



同じ効果

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

意味の分かりづらい
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。

27

#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

28

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い。)

29

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) ;

詳しくは、第9回で説明する。

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。

30

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;
```

31

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
LDLAGS=-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)

32

Makefile 例

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

33

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:B0zB0xx
    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 */
}
```

35

```
/* 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 */
```

36

```
/* 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;
}
```

37

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。

\$

標準入力
(キーボードから
打ち込む)

38

第5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 の行列の行列式を求めるプログラム
を作成する

2

行列式

$$\begin{vmatrix} x_{00} & x_{01} \\ x_{10} & x_{11} \end{vmatrix} \\ = x_{00}x_{11} - x_{01}x_{10}$$

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

#define 例

```
#define MEMBER 50
```

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

プログラムにおける
利用データ数の変
更が容易になる。



同じ効果

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

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. 変数は要素数分
用意される。
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、
不可)

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。

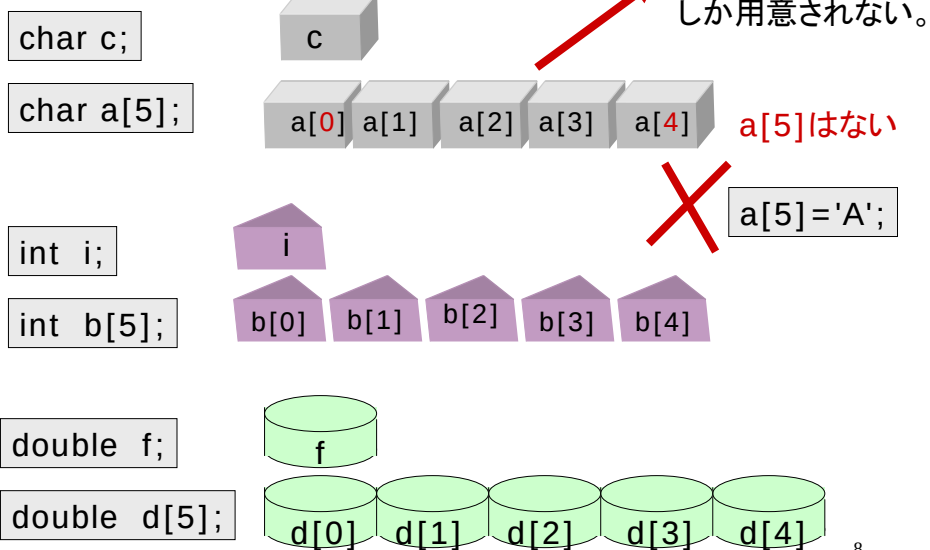
```
int i;
i = 3;
min = x[i];
```

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。

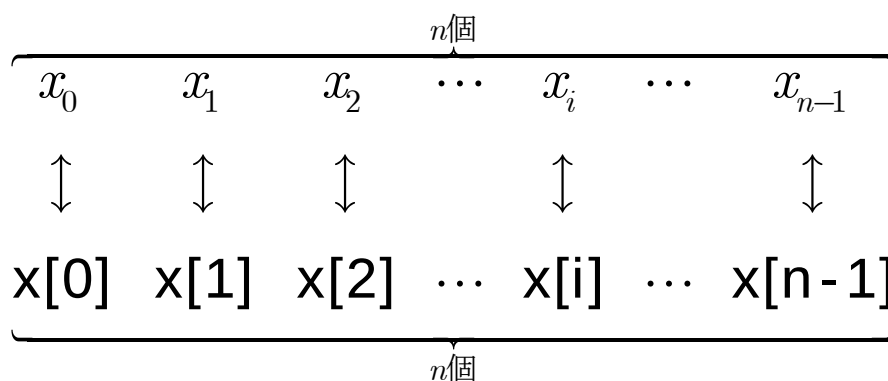
7

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>
#include <math.h>
#define SIZE 3
int main()
{
    double  a[SIZE];
    double  b[SIZE];
    double  c[SIZE];

    /*    次に行く    */
}
```

```
/* 配列要素への代入 */
a[0]=0.0;
a[1]=0.0;
a[2]=0.0;
b[0]=M_PI;
b[1]=0.0;
b[2]=0.0;
c[0]=0.0;
c[1]=0.0;
c[2]=0.0;

/*間違った代入*/
b[SIZE]=M_E; /*間違い*/
/*    次が続く    */
```

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

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

/*          続く          */
```



```
/* 配列要素の間違った利用 */  
printf("c[%d] = %f ¥n",SIZE,c[SIZE]);  
  
return 0;  
}
```

13

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];
```

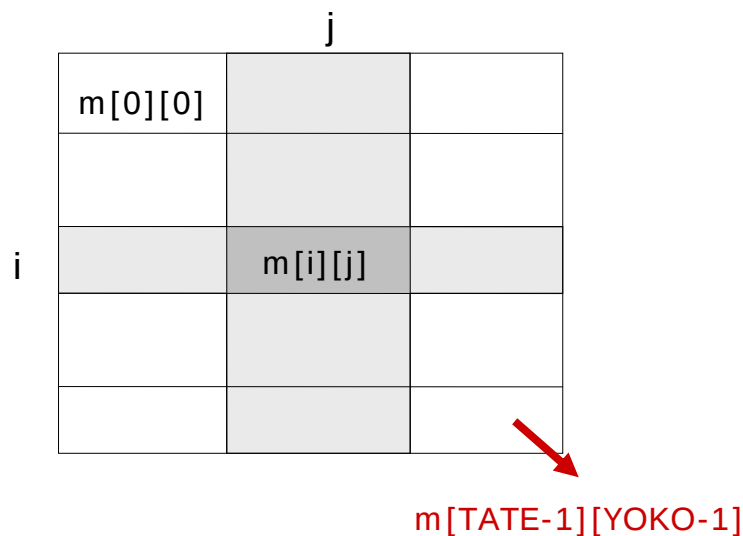
使いかた:

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。

14

2次元配列のイメージ



15

2次元配列でよくある間違い

✗ 間違い1

配列名[添字1,添字2]

`m[i, j]`

数学の座標のように、カンマで区切るのは間違い。

✗ 間違い2

配列の添字を入れ替えてしまう。

`m[i][j]` のつもりで、`m[j][i]` とする。

16

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
#define RETU 2
int main()
{
    double x[GYO][RETU]; /* 行列 */
    double tx[RETU][GYO]; /* 転置行列 */

    /* 次にく* /
```

```
/* 入力処理 */
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]);
/* 代入処理 */
tx[0][0]=x[0][0];
tx[0][1]=x[1][0];
tx[1][0]=x[0][1];
tx[1][1]=x[1][1];
/* 次にく* /
```

```
/*出力処理*/
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;
}
```

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];
```

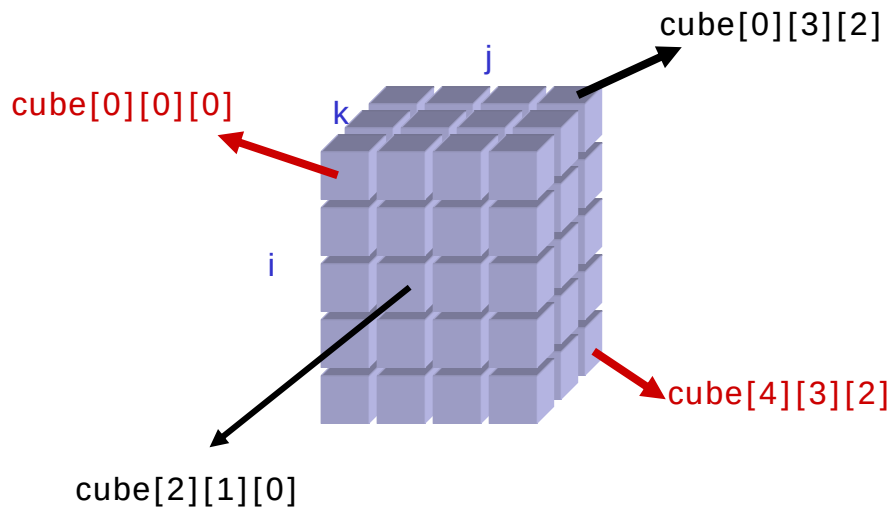
使いかた:

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。

20

3次元配列のイメージ



21

行列式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: determinant.c
    実行ファイル: determinant
    説明: 2×2行列の行列式を求めるプログラム。
    入力: 標準入力から行列の4つの成分を入力する。
           同じ行の要素が連続して入力されるとする。
    出力: 標準出力に行列式を出力する。
*/
/* 次のページに続く */

```

22

```
/* 続き */
#include <stdio.h>
/*マクロ定義*/
#define SIZE 2
int main()
{
    /* 変数宣言 */
    double det; /*行列式*/
    double matrix[SIZE][SIZE]; /*行列*/

    /* 次のページに続く */
}
```

23

```
/* 入力処理*/
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]);

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

```
/* 出力処理*/
printf("行列¥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;
}
```

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
    1.00  2.00
    3.00  4.00
の行列式は
-2.00です。
$
```

20

第6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 $ax^2 + bx + c = 0$ の実数解は、

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

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

判別式 $D = b^2 - 4ac < 0$ のとき、

実数解なし

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(式)
{
    選択実行部分1
}
```

条件 (式) が真なら選択実行部分1を実行し、
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。

偽: 0

真: 1 (0以外)

```
int  bool;

bool=1,
if(bool)
{
    ...
}
```

この部分には、
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。

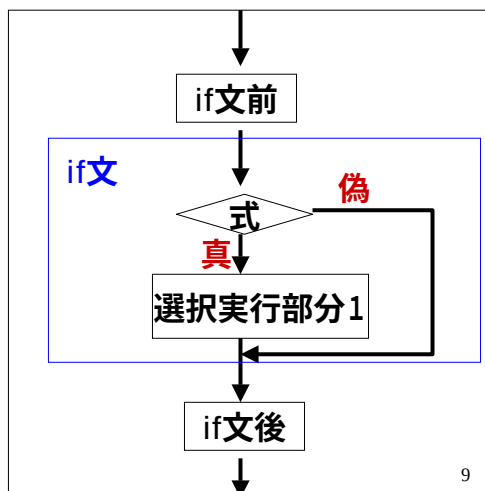
8

if文の動作1 (フローチャート)

書式

```
if(式)
{
    選択実行部分1
}
```

if文のフローチャート



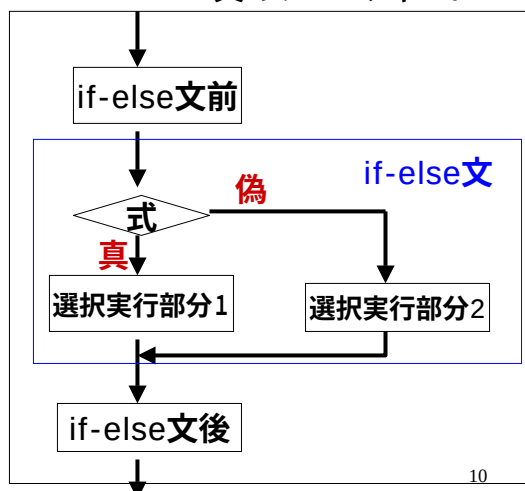
if-else文

C言語で、if 文と共に用い、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(式)
{
    選択実行部分1
}
else
{
    選択実行部分2
}
```

if-else文のフローチャート



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

$a \leq b$ a が b 以下のとき真

$a \geq b$ 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

間違い $a == b$

$a => b$



正しい $a \leq b$ a が b 以下のとき真

$a \geq b$ a が b 以上のとき真



他の間違い



$a < b < c$

$a = b > c$

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。

代入演算子

間違い

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



コンパイル時エラーにならない。

こう書くと、bの値が0以外有的时候に実行されます。

関係演算子 正しい

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



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。)

間違い

```
double a;
if(a <= 0)
{
    ...
}
```



正しい

```
double a;
if(a <= 0.0)
{
    ...
}
```



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%d",&a,&b);
    if(a==b)
    {
        printf("同じ数字です。 \n");
    }
    else
    {
        printf("異なる数字です。 \n");
    }
    return 0;
}

```

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、 それ以外の場合は偽。
A B	AまたはB (A OR B)	AとBが共に偽のときA Bは偽、 それ以外の場合は真。

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
前から評価されて、偽が現れたら偽に
決まるので、残りの式は評価されない。

式1 || 式2 || ... || 式n

式1から式nまですべてが偽なら偽
前から評価されて、真が現れたら真に
決まるので、残りの式は評価されない。

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。)

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
それ以外では偽

$(a == b) \&\& (b > c)$ aとbが等しく、かつ
bがcより真に大きいとき 真
それ以外では偽

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。

間違い

```
double a;  
if(a)  
{  
    .....  
}
```

**正しい**

```
double a;  
if(a!=0.0)  
{  
    .....  
}
```



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 */  
}
```

```

/* 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

```

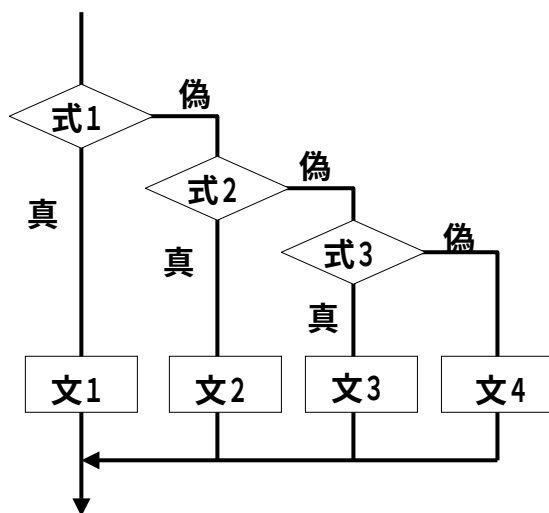
if( 式1)
{
    選択実行部分1
}
else if(式2)
{
    選択実行部分2
}
.
.
else if(式n)
{
    選択実行部分n
}
else
{
    選択実行部分 (n+1)
}

```

式は上から評価されて、真になった式に対応する選択実行部分が実行される。すべての式が偽なら、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(式1 )
{
    文1
}
else if(式2)
{
    文2
}
else if (式3)
{
    文3
}
else
{
    文4
}
  
```

25

多分岐2 (多重分岐)

選択実行部分中にも、if文を書く事ができる。

```

if( 式)
{
  選択実行部分1
}
  
```

選択実行部分1

ここに、
またif文を書ける。

```

if(a>b)
{
    printf("aはbより大きい。¥n");
    if(b>c)
    {
        printf("a>b>cの順序です。¥n");
    }
}
  
```

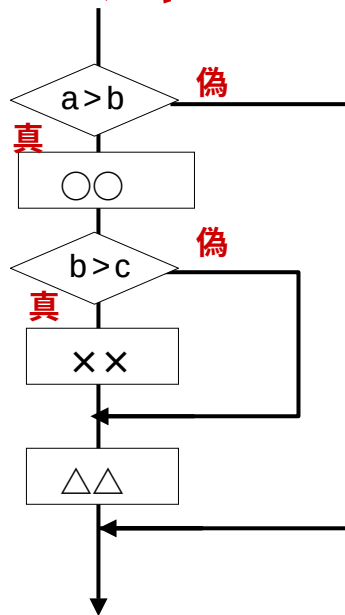
26

多重分岐のフローチャート

```

if(a>b)
{
  ○○
  if(b>c)
  {
    ××
  }
  △△
}

```



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:B0zB0xx
  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でない実数を入力する。
    b、cには任意の実数を入力する。
    a,b,cの順序に入力する。

  出力:
    標準出力に2つの解を出力する。
*/
/*      次のページに続く      */

```

```
#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 */
    double c; /* 定数項 */

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

    double sol1; /* 解1 */
    double sol2; /* 解2 */

    /* 続く */
}
```

29

```
/* 入力処理 */
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);

/* 続く */
```

30

```

/*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以外にしてください。¥n");
    return -1;
}
/* これ以降では a は0.0以外*/

/*    計算処理    */
dis=b*b - 4.0*a*c; /*判別式の計算*/
/*次のページに続く*/

```

31

```

if(dis >= 0.0)
{
    /*実数解が存在する。*/
    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。*/
    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;
}

```

32

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。
$
```

33

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。
$
```

34

第7回 条件による繰り返し



1

今回の目標

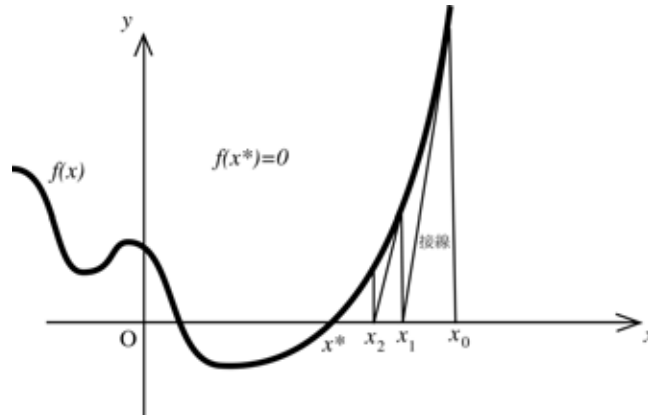
- アルゴリズムの基本となる制御構造 (順次、分岐、反復構造) を理解する。
- 繰り返し (反復構造、ループ) を理解する。
- ループからの様々な終了の方法を理解する。
- 繰り返しを用いたアルゴリズムに慣れる。

☆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

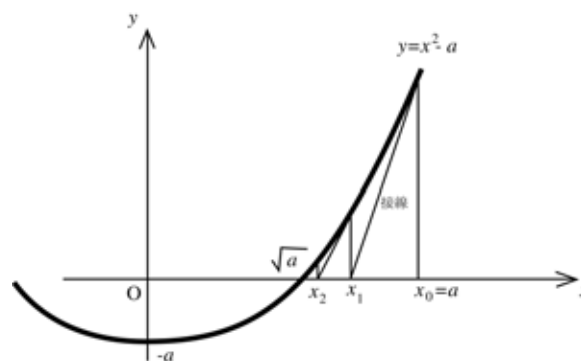


$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} \quad x_0, x_1, x_2, \dots, x_\infty \rightarrow x^*$$

3

ニュートン法による平方根の計算

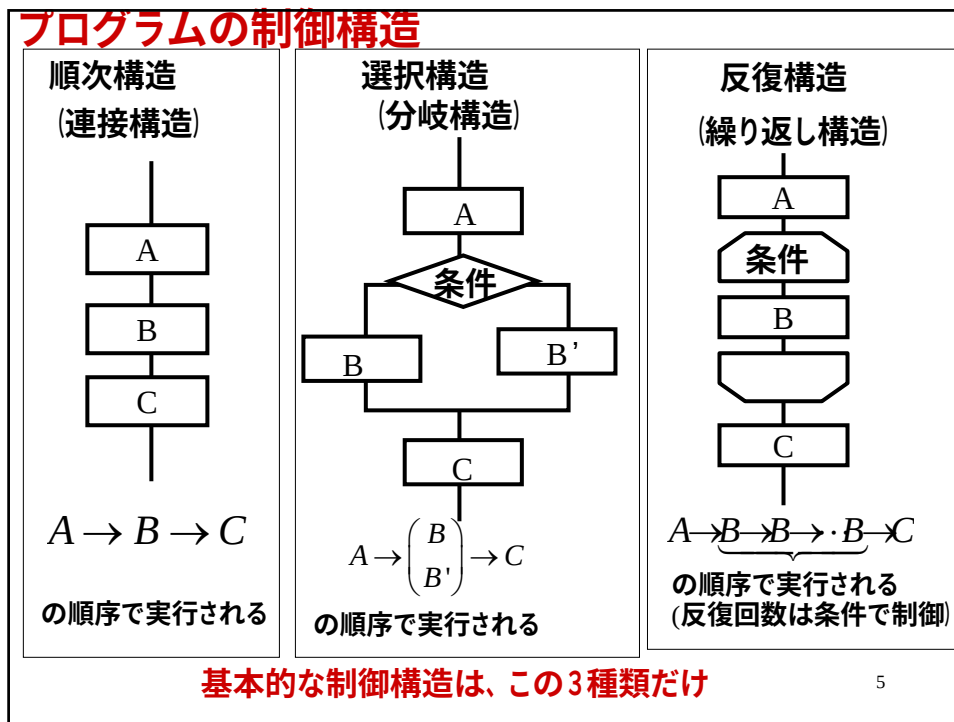
$\sqrt{a}$ は $f(x) = x^2 - a = 0$ の解なので、



$$x_{i+1} = x_i - \frac{x_i^2 - a}{2x_i} \quad a = x_0, x_1, x_2, \dots, x_\infty \rightarrow \sqrt{a}$$

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

4



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。

- 回数を指定して繰り返す。

主に、for文を用いると良い。(第8回で詳しく扱う。)

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。

while文

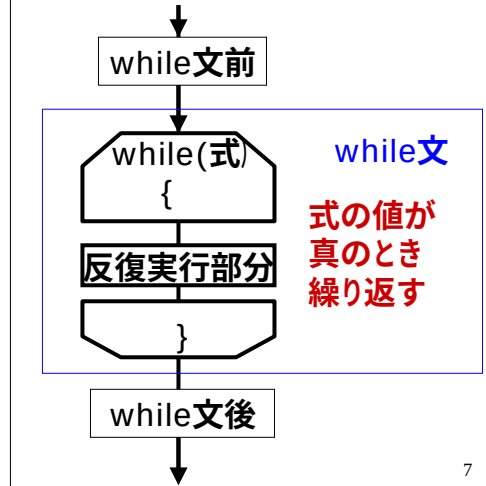
式 (論理式) が真である間、命令を繰り返し実行する。

書式

```
while(式)
{
    反復実行部分
}
```

式: 反復を続ける条件を表す論理式

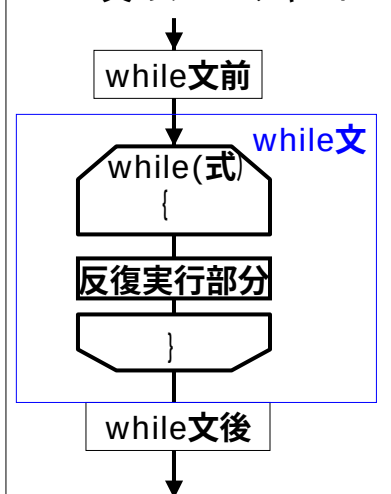
while文のフローチャート1



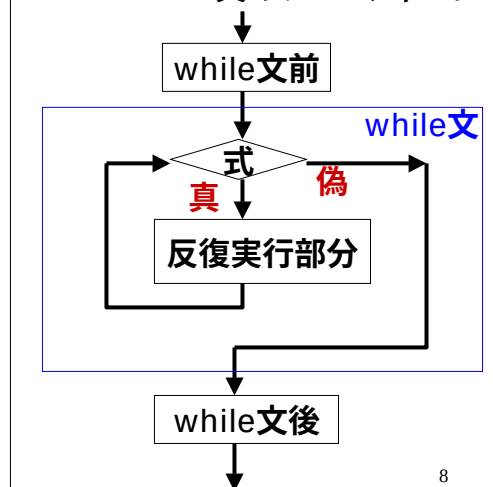
whileループのフローチャート

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1



while文のフローチャート2



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. ループカウンタを整数型で用意する(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)。

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;
}

```

break文

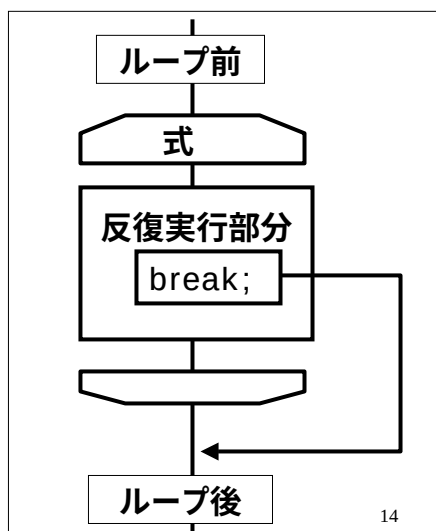
反復実行部分内で用い、breakに出会うと繰り返し文が終了する。(次の実行は、ループを閉じる右中括弧直後から)

書式

```

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

```



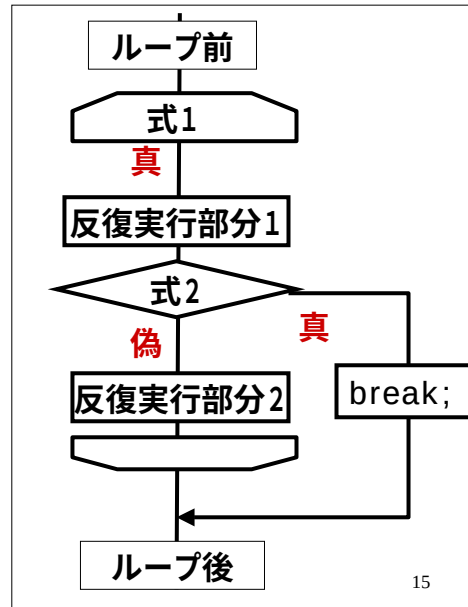
14

break文の典型的な使い方

典型的な使い方

```
while( 式1)
{
    反復実行部分1
    if(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。



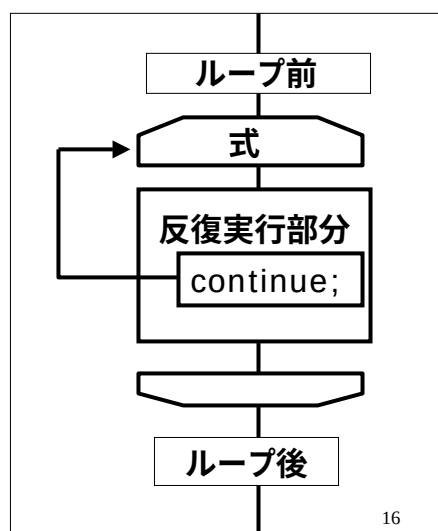
15

continue文

反復実行部分内で用い、continue文に出会うと次の繰り返しを実行する。(for文の場合、式3は実行される。)

書式

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



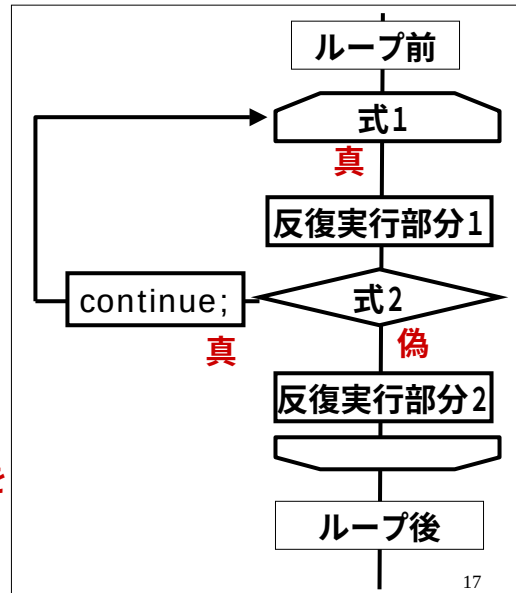
16

continue文の典型的な使い方

典型的な使い方

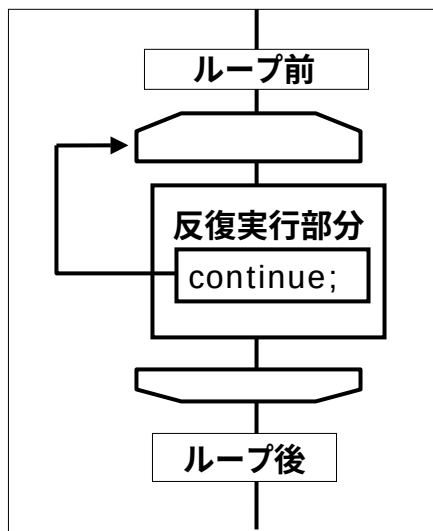
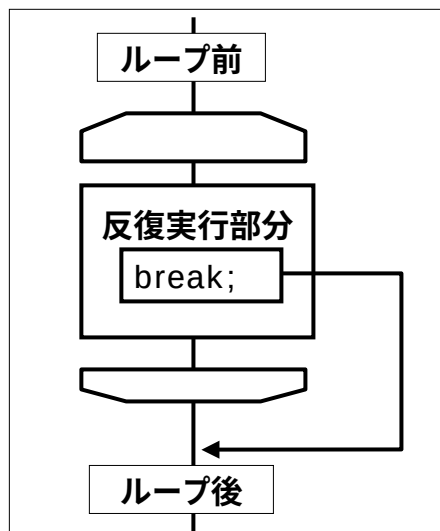
```
while(式1)
{
    反復実行部分1
    if(式2)
    {
        continue;
    }
    反復実行部分2
}
```

ループ中に反復実行文2を
実行するか選択できる。



17

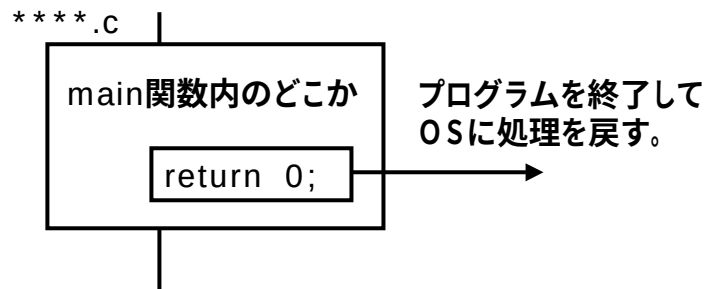
continue文とbreak文



18

return文

main関数で、return文を実行すると、
プログラムが終了する。
(詳しくは、第9回の関数で説明。)



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:B0zB0xx
  ソースファイル:mysql.c
  実行ファイル:mysql
  説明:ニュートン法を用いて平方根を求めるプログラム。
  近似値の階差の絶対値がEPS(1.0e-5)より小さいときに
  終了する。(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つの実数を入力する。(正、0、負いずれも可)
  出力:入力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を出力する。
        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  */
```

```
#include <stdio.h>
#include <math.h>
/*マクロ定義*/
#define EPS (1.0e-5) /*微小量、収束条件*/
#define LOOPMAX 1000 /* 繰り返しの上限值*/

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 old_x; /* 漸化式の右辺で用いるx*/
    double new_x; /* 漸化式の左辺で用いる変数x*/
    double sa; /* 階差の絶対値*/
    int kaisuu; /*繰り返し回数*/
    /* 次ページへ続く */
```

```
/* 入力処理*/
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);

/*入力値チェック*/
if(input == 0.0)
{
    /*input=0.0のときは、明らかに0が平方根*/
    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

```
/*ニュートン法の初期設定*/
old_x=input; /*ニュートン法の初期値をinputに設定*/
new_x=0.0;
sa=fabs(new_x-old_x);
kaisuu=0;
/*繰り返し処理*/
while(sa < EPS)/*階差がEPSより小さい時終了する*/
{
    printf("x%d = %15.8f ¥n",kaisuu,old_x);/*途中表示*/
    new_x=(old_x+input/old_x)/2.0; /*漸化式*/
    sa=fabs(new_x-old_x); /*階差の絶対値*/

    /* 次の繰り返しのための処理 */
    old_x=new_x;
    kaisuu++;
    if(kaisuu>=LOOPMAX)
    { /*繰り返し回数の上限を超えたので終了する*/
        break;
    }
}
/*次のページに続く*/
```

```
/* 続き */
/*出力処理/
printf("%6.2f の平方根は%15.8fです。¥n",input,new_x);
return 0;
}
```

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
x4=      1.41421356
  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8回 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。
- 配列とfor文の組み合わせ方を理解する。
- 多重ループを理解する。

☆平均値を求めるプログラムを作る。

2

平均

n 個のデータ $X_0, X_1, \dots, X_{n-1}$
の平均値

$$\bar{X} = \frac{X_0 + X_1 + \dots + X_{n-1}}{n}$$

$$= \frac{\sum_{i=0}^{n-1} X_i}{n}$$

3

for文

式2 (論理式) が真である間、命令を繰り返し実行する。

書式

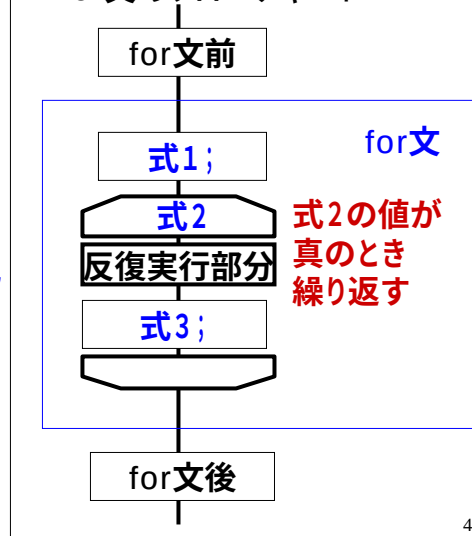
```
for(式1; 式2; 式3)
{
    反復実行部分
}
```

式1: ループカウンタの初期化

式2: 反復を続ける条件

式3: ループカウンタの
インクリメント

for文のフローチャート



4

for文

典型的な使い方

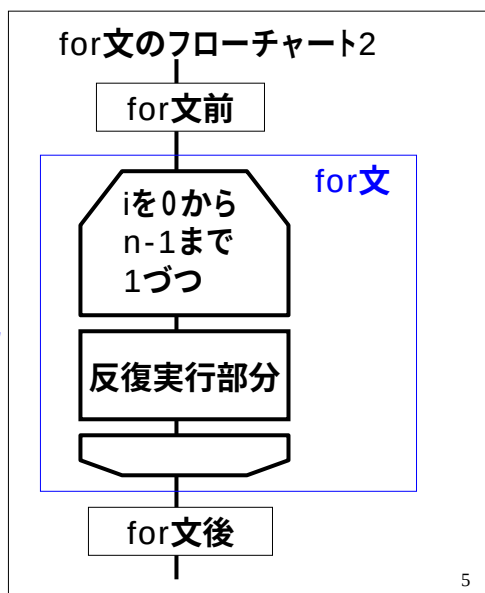
```
for(i=0;i<n;i++)
{
    反復実行部分
}
```

$i=1$: ループカウンタの初期化

$i<n$: 反復を続ける条件

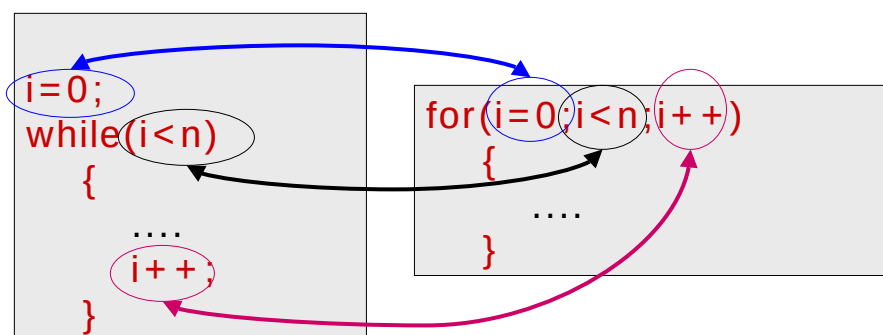
$i++$: ループカウンタの
インクリメント

for文のフローチャート2



5

while 文との比較

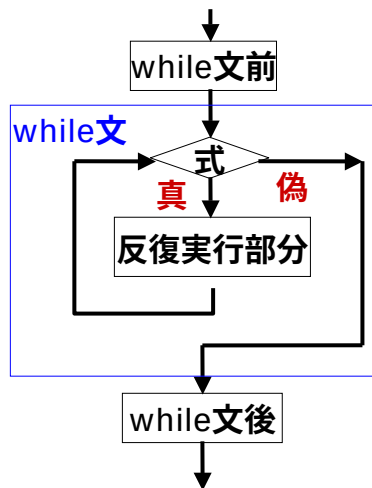


for文では、回数指定の繰り返しの制御記述を、
一個所にまとめている。

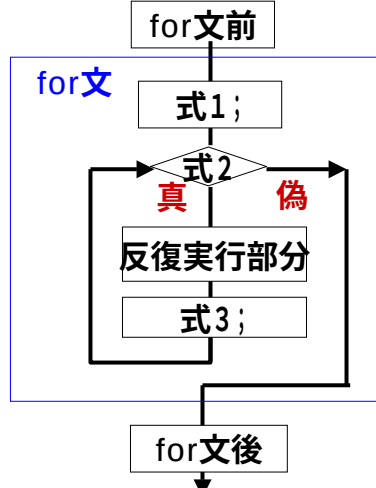
6

while文とfor文のフローチャート

while文のフローチャート



for文のフローチャート



7

while文とfor文の書き換え

```

式1;
while(式2)
{
    反復実行部分
    式3;
}
  
```



```

for(式1;式2;式3)
{
    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; /*反復回数用*/
    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

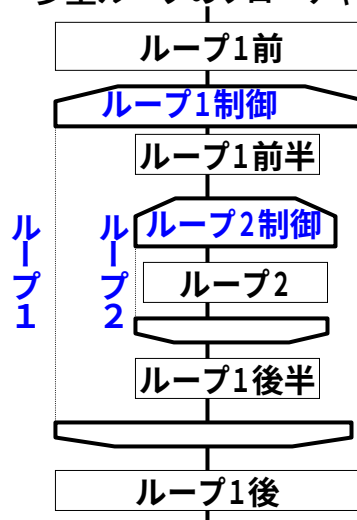
ループ構造の反復実行部分に、小さいループ構造が入っている制御構造。

典型的な例

```

for(式1;式2;式3)
{
    ループ1前半
    for(式4;式5;式6)
    {
        ループ2
    }
    ループ1後半
}

```

多重ループのフローチャート

10

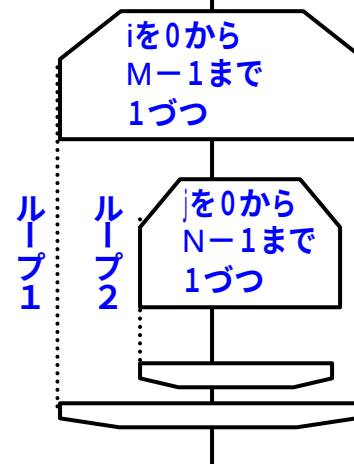
多重ループとループカウンタ

典型的な例

```
/*外側の反復回数*/
#define M 10
/*内側の反復回数*/
#define N 10
int i; /*外側のカウンタ*/
int j; /*内側のカウンタ*/
for(i=0;i<M;i++)
{
    for(j=0;j<N;j++)
    {
        ループ2
    }
}
```

多重ループでは、ループ事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

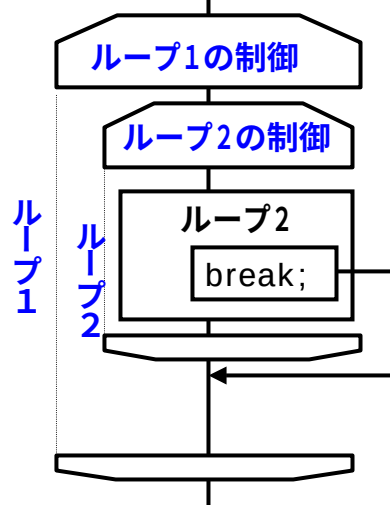
多重ループとbreak文

典型的な例

```
for(式1;式2;式3)
{
    for(式4;式5;式6)
    {
        (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");

    /* 次に行く */
}
```

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

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: B0zB0xx
    ソースファイル: average.c
    実行ファイル: average
    説明: n個の実数から平均値を求めるプログラム。
    入力: 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: 標準出力にデータ中の最大値を出力。
*/
/* 次のページに続く */

```

15

```

/* 続き */
#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; /* 平均値 */
    double sum; /* 総和 */
    double data[DATANUM]; /* データを入れる配列 */
    /* 入力処理 */
    /* 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 */
}

```

```
/*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*/
```

17

```
/*続き*/  
/* 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]);  
}  
  
/* 次に行く */
```

18

```
/*初期設定*/
sum=0.0;

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

/*平均の計算 */
ave=sum/(double)n;

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

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。
$
```

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回関数と記憶クラス



1

今回の目標

- C言語における関数を理解する。
- 関数における仮引数の役割について理解する。
- 関数の戻り値について理解する。
- 関数の副作用について理解する。
- 変数の適用範囲 (スコープ) について理解する。

☆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

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

3

関数の定義

書式:

```

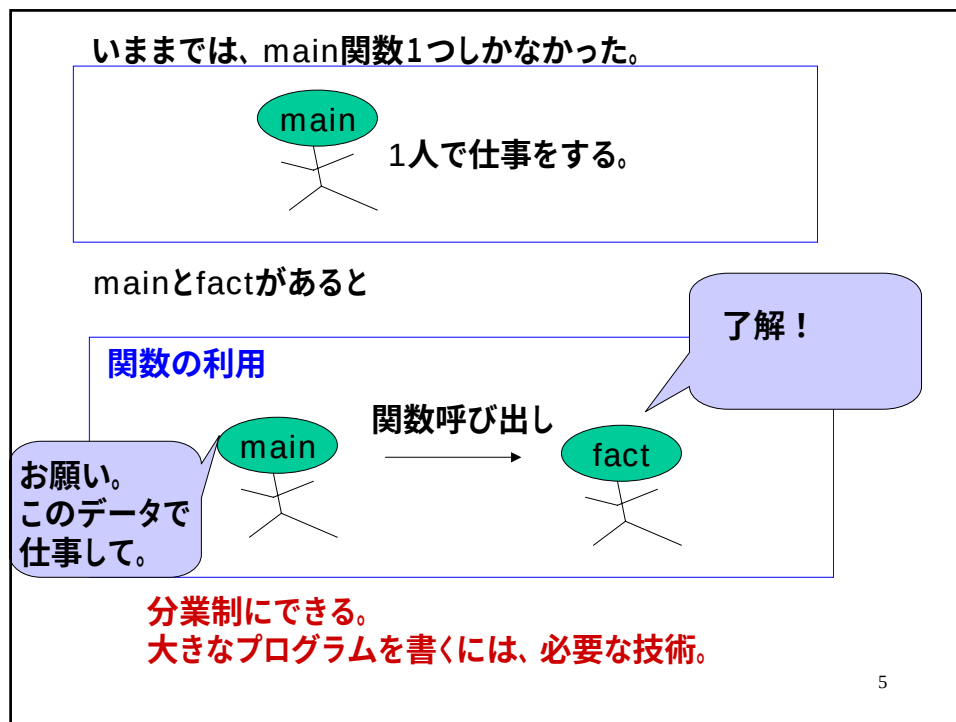
戻り値の型 関数名 (仮引数の宣言付きリスト)
{
    関数の本体
    return 式;
}

```

return文の式と同じ型

戻り値という。
もちろん、
定数や変数であっても
良い。

4



処理の分割と関数の利用

```
int main()
{
  /*組み合わせ数を計算 */
  bunshi=fact(n);
  bunbo1=fact(m);
  bunbo2=fact(n-m);
  com=bunshi/(bunbo1*bunbo2);
  return 0;
}
```

階乗を求める専門家 (関数) があると、便利。

```
int fact(int k)
{
  階乗の処理

  return kの階乗;
}
```

依頼データを元に、結果を返す。
(階乗の計算を行なう。)

関数例

mainというのも関数の一つ。
Cではプログラムは関数の集まりで作られる。

```
int main()  
{  
    関数の本体  
    return 0;  
}
```

戻り値の型や関数への仮引数のリストは省略可能だが、
括弧の省略はできない。

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

7

関数例2

階乗を求める関数の定義

戻り値の型 (facの型)

関数名 (自分で命名できる。
スタイル規則参照。)

仮引数リスト
型 変数名

```
int fact(int n)  
{  
    階乗を求める処理の記述  
    return fac;  
}
```

戻り値

注意: メイン関数以外はプロトタイプ宣言を行うこと。

8

戻り値の型とreturn文

書式:

```
return 式;
または、
return ;
```

関数の実行中にreturn文に出会うと、その式の値を呼び出した関数に返してその関数を終了する。

```
int fact(int n)
{
    * * * *
    int fac;
    * * * *
    * * * *
    return fac;
}
```

9

プロトタイプ宣言と関数の定義

書式

```
/* プロトタイプ宣言 */
型1 関数名1 (型a 仮引数a);          /*関数1のプロトタイプ宣言*/

/* メイン関数*/
int main()
{
    * * * *
    return 0;
}

/* 関数1の定義 (関数1の本体) */
型1 関数名1 (型a 仮引数a)
{
    * * * *
    return 型1の式;
}
```

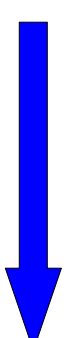
プロトタイプ宣言と関数定義において、セミコロンの有無に注意すること。

注意: メイン関数以外は、プロトタイプ宣言をメイン関数前に記述する。

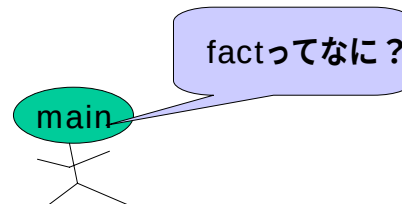
10

プロトタイプ宣言の役割

プログラムは、
上から下の実行されるので、
プロトタイプ宣言が無いと。



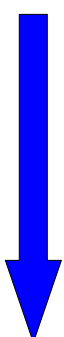
```
/**/  
int main()  
{  
  
    fact(m);  
    return 0;  
}  
  
int fact(int n)  
{  
    return fac;  
}
```



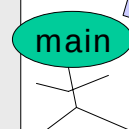
11

プロトタイプ宣言の役割2

プロトタイプ宣言があると。



```
int fact (int n);  
int main()  
{  
  
    fact(m);  
    return 0;  
}  
  
int fact(int n)  
{  
    return  fac;  
}
```



factは
なにか整数データを与えると、
(なにか処理して)
整数データを返してくる関数だな。
だから、
fact関数を使うときは、
整数データを与えているかだけチェックして
あとは、
fact関数さんに任せて、
整数データが戻ってくるまでまてば
いいんだな。

プロトタイプ宣言例

書式だけ抽出

```
int fact(int n);  
int main()  
{  
    fact(m);  
    return 0;  
}  
  
int fact(int n)  
{  
    int fac;  
  
    return fac;  
}
```

プロトタイプ宣言

関数の呼び出し

関数定義
(関数の本体)

13

実引数と仮引数

```
int fact(int n);  
int main()  
{  
    fact(m);  
    return 0;  
}  
  
int fact(int n)  
{  
    int fac;  
  
    return fac;  
}
```

呼び出す側の式 (値) を
実引数 (じつひきすう) と呼び、
呼び出される側の変数を
仮引数 (かりひきすう) と呼ぶ。

このmの値は実引数

この変数nは仮引数

注意: 実引数に変数の場合でも、
実引数と仮引数の名前
は異なってもかまわない。¹⁴

関数へ値の渡し方

呼び出す方では、

`関数名 (式);`

や

`変数 = 関数名 (式);`

などで関数を呼び出す。

呼び出される方では、

仮引数に実引数の値が”代入”される。

注意: 実引数は変数でも
定数でも式でもよい。

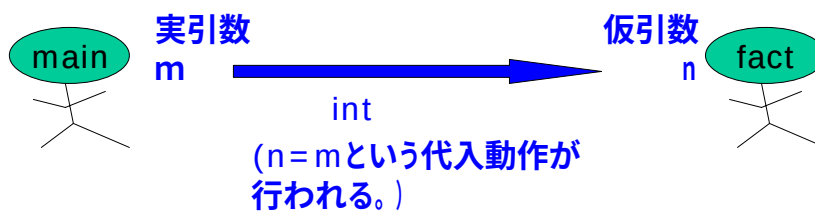
```
int main()
{
    fact(m);
}
int fact(int n)
{
    int fac;
    return fac;
}
```

mainのmの値が、
factのnに代入される。

15

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

```
int fact(int n)
{
    return fac;
}
```



16

呼び出し側への戻り値の渡し方

呼び出す方で、単に **関数名 (式);** とすると、
 せっかくの戻り値が利用できない。

変数 = 関数名 (式); とすると、戻り値が変数に代入される。

```
int main()
{
    int a;
    a = fact(m);
}

int fact(int n)
{
    int fac;
    return fac;
}
```

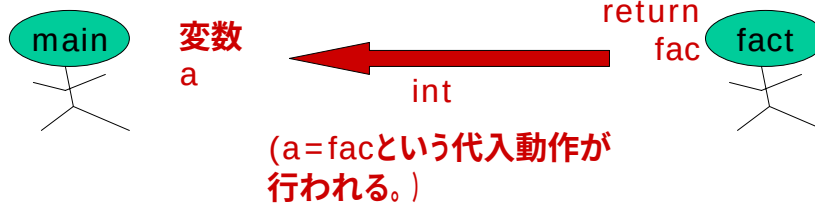
呼び出す側では、
 "関数名 (式)" 全体を
 一つの式あるいは一つの変数
 のように考えてもよい。

factのfacの値が、
 mainのaに代入される。

17

```
int main()
{
    int a;
    a = fact(m);
    return 0;
}
```

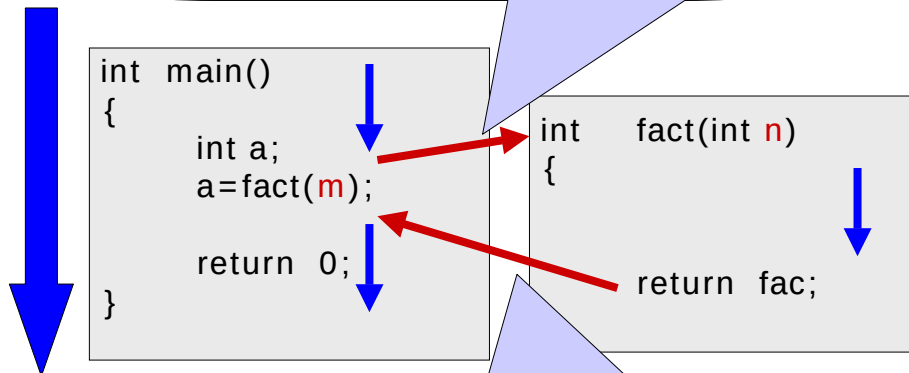
```
int fact(int n)
{
    return fac;
}
```



18

関数が複数あるときの制御の流れ1

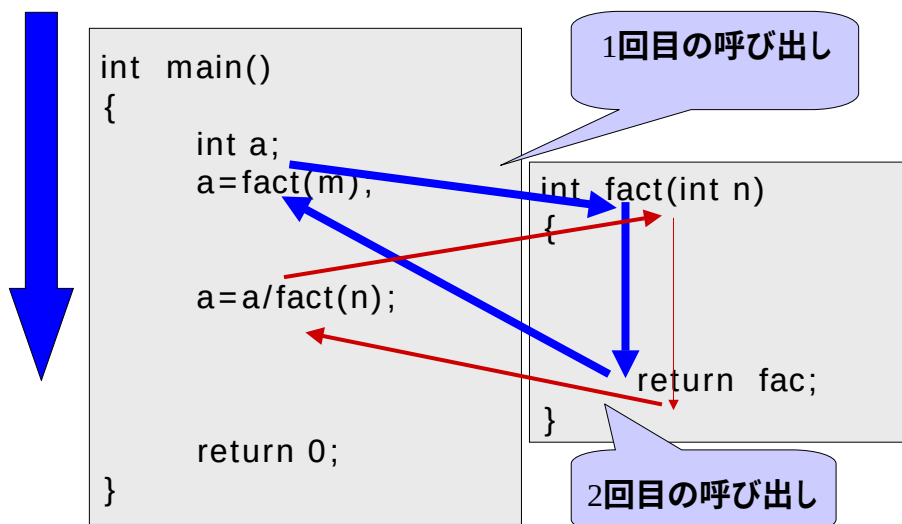
制御が関数factに移ると共に、実引数(mainのm)の値が仮引数(factのn)に代入される。



制御がmain関数に移る共に、factの式facの値がmainの変数aに代入される。

19

関数が複数あるときの制御の流れ2

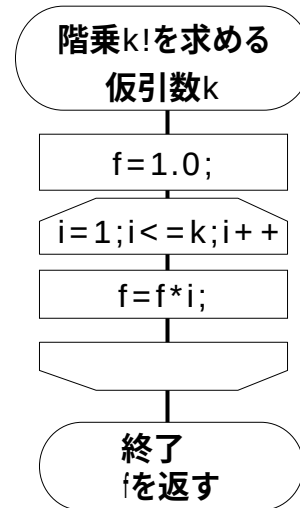
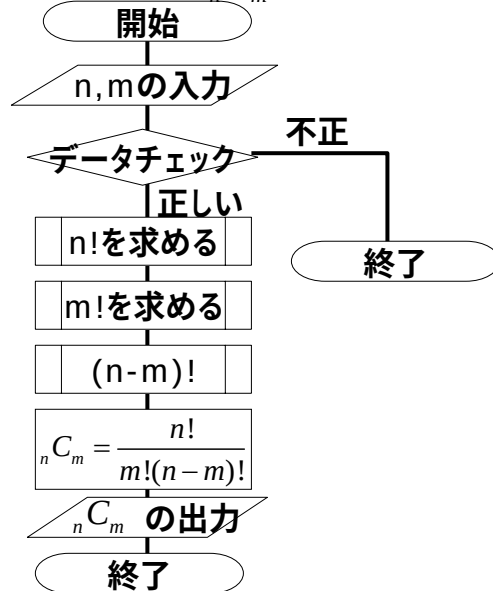


同じような処理を複数回行いたいときにも、関数を用いると便利。

20

関数を表わすフローチャート

組み合わせ数 ${}_nC_m$ を求める



21

練習1

```

/*test_function.c 関数実験1 コメント省略*/
#include <stdio.h>
int switch_sign(int);
int main()
{
    int a;
    int b;
    printf("整数を入力して下さい。 a = ?");
    scanf("%d",&a);
    printf("関数呼び出し前 a=%d : b= %d ¥n",a,b);
    b=switch_sign(a);
    printf("関数呼び出し後 a=%d : b= %d ¥n",a,b);
    printf("%d = switch_sign( %d) ¥n",b,a);
    return 0;
}/* つづく*/
  
```

22

```
int switch_sign(int c)
{
    printf("関数switch_sign実行中¥n");
    printf("c= %d¥n",c);
    return -c;
}
```

23

voidという型

```
return;
```

という文を持つ関数には戻り値がない。
このように、戻り値がないことをvoidという型であらわす。

```
void fanc1()
{
    return ;
}
```

あるいは、仮引数がないことを
明示的にvoidという型で表す。

```
void fanc2 (void)
{
    return ;
}
```

24

練習2

```

/*test_void.c 関数実験2 コメント省略*/
#include<stdio.h>
void print_com(void);
int main()
{
    print_com( );
    return 0;
}

void print_com(void)
{
    printf("print_com内で実行¥n");
    return;
}

```

いつもの処理やって

終わったよ。

複数の関数を持つプログラムの書き方 書式

```

/*      プロトタイプ宣言      */
型1 関数名1 (型1a 仮引数1a, 型1b 仮引数1b, …) ;
型2 関数名2 (型2a 仮引数2a, 型2b 仮引数2b, …) ;
int main()
{
}

型1 関数名1 (型1a 仮引数1a, 型1b 仮引数1b, …)
{
}

型2 関数名2 (型2a 仮引数2a, 型2b 仮引数2b, …)
{
}

```

26

練習3

```
/*test_function3.c 関数実験3 コメント省略*/
/*ヘッダファイルの取り込み*/
#include <stdio.h>

/*プロトタイプ宣言*/
void print_com(void); /* コメントを表示する関数 */
int plus(int,int); /* 和を求める関数 */
int main()
{
    int a;
    int b;
    int c;

    /*次に続く */
```

27

```
/*続き*/
    printf("整数を入力して下さい。a= ?");
    scanf("%d",&a);
    printf("整数を入力して下さい。b= ?");
    scanf("%d",&b);

    /*引数と戻り値が無い関数呼び出し*/
    print_com( );

    /*引数を基に与えて戻り値を受け取る関数
    呼び出し。*/
    c=plus(a,b);

    printf("%d = plus( %d,%d) ¥n",c,b,a);
    return 0;
}
```

28

```
/*続き*/  
/*コメントを表示する関数  
   仮引数:なし (void)  
   戻り値:なし (void)*/  
void print_com(void)  
{  
    printf("計算開始\n");  
    return;  
}  
/*print_com終了*/
```

この関数のように、
値のやりとりがなくても、
なにか動作することがあります。
これを副作用といいます。

```
/*続き*/  
/*2つの整数の和を計算する関数  
   仮引数x:被演算項1 (整数)  
   仮引数y:被演算項2 (整数)  
   戻り値:"被演算項1+被演算項2"の値を返す。*/  
int plus(int x, int y)  
{  
    /*変数宣言*/  
    int z;  
  
    /*計算*/  
    z=x+y;  
    return z;  
}  
/*plus終了*/  
/*全プログラム (test_fanction3.c)終了*/
```

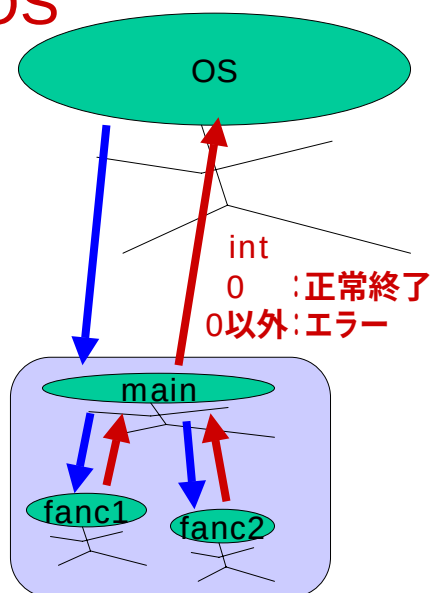
こんな風に、
関数にコメントを付けること。
(スタイル規則参照。)

30

関数mainの型とOS

```
/* aaaaa.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。



31

変数のスコープ1 (有効範囲1)

関数定義の一般的な書式:

```
型1 関数名1 (型1a 仮引数1a, 型1b 仮引数1b, ...)  
{  
    /*変数宣言*/  
    型x    変数x  
}
```

引数が複数ある場合は、
引数リスト中で
各引数をカンマ「,」
で区切る。

仮引数とその関数内で宣言した変数は、
宣言した関数の内部だけで有効である。

したがって、異なる2つの関数で同じ変数名を用いても、
それぞれの関数内で別々の変数としてあつかわれる。

32


```

int    main()
{
    /*変数宣言*/
    mainの変数
}

型1    関数1 (型1a 仮引数1a,型1b 仮引数1b)
{
    /*変数宣言*/
    関数1の変数
}

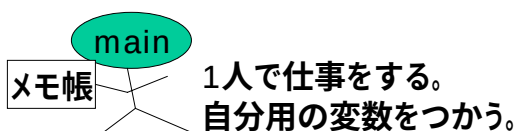
型2    関数2 (型2a 仮引数2a,型2b 仮引数2b)
{
    /*変数宣言*/
    関数2の変数
}

```

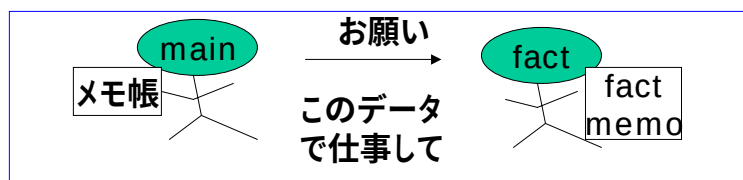
33

イメージ

いままでは、main関数1つしかなかった。



mainとfactがあると



34

練習4

```

/*test_scope.c 関数実験4 コメント省略*/
#include<stdio.h>
void myvar(void);
int main()
{
    int a;

    printf("( In main) Input a= ? ");
    scanf("%d",&a);
    printf("(In main) a= %d ¥n",a);
    myvar();
    printf("(In main) a= %d ¥n",a);
    return 0;
}
/* 次が続く */

```

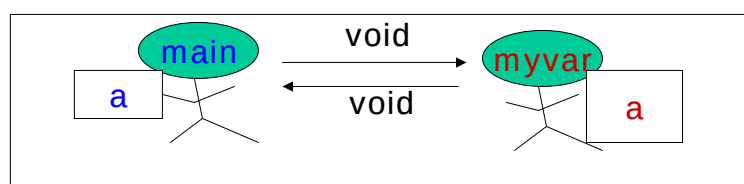
```

/* 続き */
void myvar(void)
{
    int a;

    printf("( In myvar) Input a= ? ");
    scanf("%d",&a);
    printf("(In myvar) a= %d ¥n",a);

    return;
}

```



36

グローバル変数とローカル変数

実は、関数の外でも、変数の宣言ができます。
その変数をグローバル変数と呼びます。

一般的な形

```
/*グローバル変数宣言*/
型A 変数A;
型B 変数B;

int main()
{
    /*mainのローカル変数宣言*/
}
```

本演習では、
グローバル変数は、
1文字だけ英大文字
で残り小文字にしましょう。
(スタイル規則参照)

```
int Global1;
```

37

グローバル変数のスコープ

```
/*グローバル変数宣言*/
グローバル変数

int main()
{
    /*変数宣言*/
    mainの変数
}

型1 関数1 (型1a 仮引数1a, 型1b 仮引数1b)
{
    /*変数宣言*/
    関数1の変数
}
```

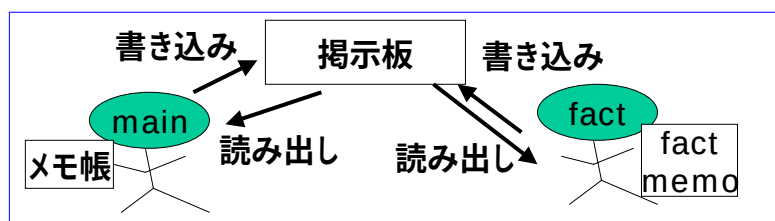
38

ローカル変数とグローバル変数

ローカル変数は、自分用のメモ帳



グローバル変数は、どの関数でも読み書きできる掲示板



39

グローバル変数とローカル変数が同じ名前的时候は？

ある関数でグローバル変数と同じ名前の変数を宣言すると、その関数内ではその変数名はローカル変数としてあつかわれる。したがって、グローバル変数の変更はおこなわれない。

(変数名が同じもの同士では、そのスコープが狭いものが優先される。関数が違えば、同じ変数名でも大丈夫。)

注意：

本演習のスタイルでは、
ローカル変数とグローバル変数は必ず異なる。

ローカル変数：すべて小文字

グローバル変数：1文字目大文字

マクロ名：すべて大文字

40

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:B0zB0xx
   ソースファイル:combi1.c
   実行ファイル:combi1
   説明:組み合わせ数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数nCmを出力。
*/
#include <stdio.h>

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

```
/* 続き */
/* main関数 */
int main()
{
    /*ローカル変数宣言*/
    int n; /*nCmのn*/
    int m; /*nCmのm*/
    int com; /*組み合わせ数nCm*/

    /* 次のページに続く */
}
```

```
/*    続き    */
/*    入力処理 */
printf("組み合わせ数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。*/
/*    次ページへ続く    */
```

```
/*    続き    */

/*    組み合わせ数を計算 */
com=fact(n)/(fact(m)*fact(n-m));

/*出力処理*/
printf("%d  C  %d = %5d¥n",n,m,com);

return 0;
}
/*main関数終了*/

/*    次へ続く    */
```

```
/*    続き    */
/*階乗を求める関数
   仮引数n: n!のn
           (0以上15未満の値とする。)
   戻り値:n!を返す。*/
int fact(int n)
{
    /*    ローカル変数宣言    */
    int i;    /*ループカウンタ*/
    int fac;  /*階乗n!*/

    fac=1;    /*0!=1であるので1を代入*/

    /*    次に行く    */
}
```

45

```
/*    続き    */
/*計算処理*/
for(i=1;i<=n;i++)
{
    /*階乗の計算*/
    fac=fac*i;
}

return fac; /*facの値n!を返す*/
}
/*factの終了*/
/*全てのプログラム(combi1.c)の終了*/
```

46

実行例

```
$make  
gcc combi1.c -o combi1  
$ ./combi1  
組み合わせ数 $nCm$  を計算します。  
Input n=? 4  
Input m=? 3  
4C3 = 4  
$
```

```
$. /combi1  
組み合わせ数 $nCm$  を計算します。  
Input n=? 4  
Input m=? 5  
不正な入力です。  
$
```

47

第10回 関数と再帰



1

今回の目標

- 再帰的な考え方に慣れる
 - C言語における再帰関数を理解する。
- ☆階乗を求める再帰的な関数を作成し、その関数を利用するプログラムを作成する。

2

階乗 $n!$ の2つの数学的表現

(1) 繰り返しによる表現

$$n! = 1 \times 2 \times \cdots \times i \times \cdots \times n \quad (n \geq 1 \text{ のとき。})$$

$$= \prod_{i=1}^n i \quad (\text{なお、} 0! = 1)$$

(2) 漸化式による表現

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

3

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 (仮引数の宣言付きリスト)
{
    関数名 (実引数リスト);
    return 式;
}
```

同じ関数名

注意: 再帰関数は、漸化式で表わされている数学的な関数などを直接プログラミングすることができる。

4

再帰関数例

```
int fact(int n)
{
    int fac;

    if(n == 0)
    {
        fac = 1;
    }
    else
    {
        fac = n * fact(n-1);
    }

    return fac;
}
```

再帰呼びだし
といいます。

再帰の典型的な書き方

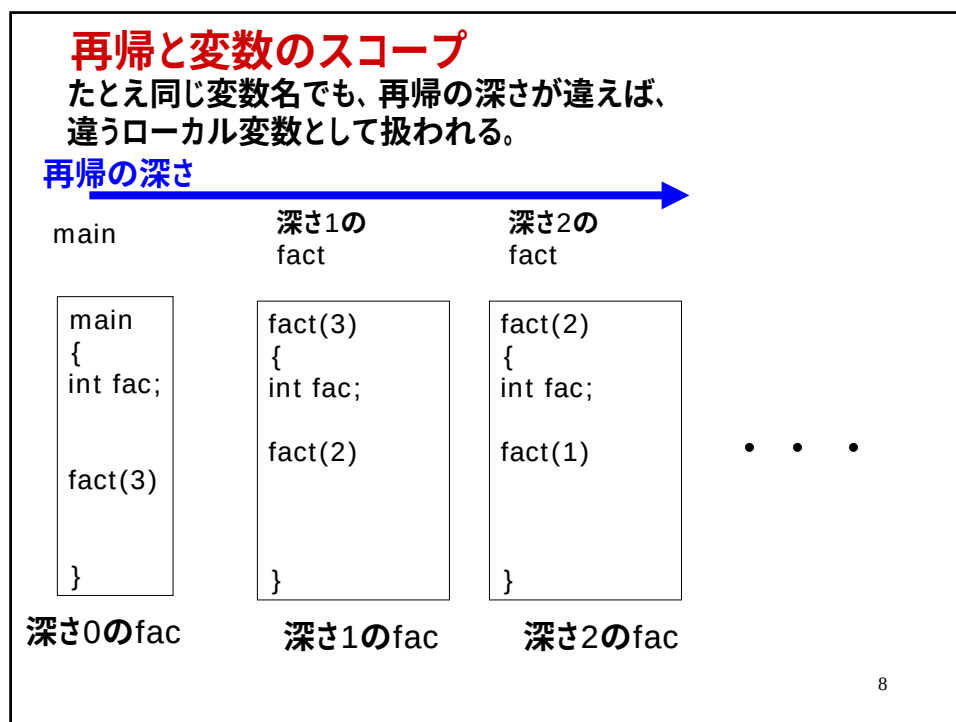
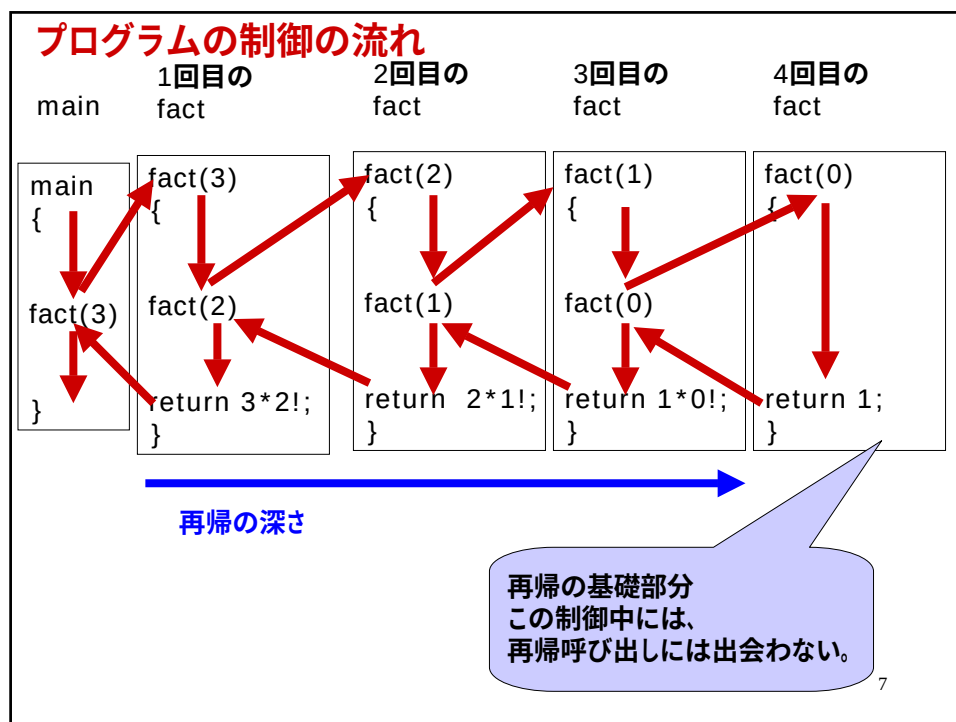
書式だけを抽出

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

必ず再帰の基礎
(出口)をつくる事。

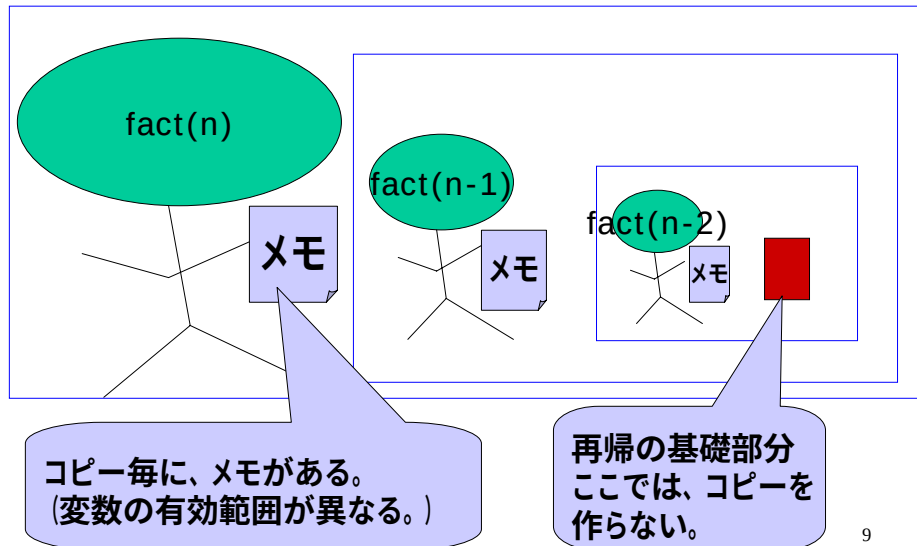
再帰呼び出しをするとき
には、
必ず出口に近づくように
すること。

6



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。



9

練習1

```
/* print_depth.c 再帰関数実験1 コメント等一部省略 */
#include <stdio.h>
#define MAX 10 /* nの最大値、見栄えを制御 */
int fact(int n); /* 階乗n!を求める再帰的な関数
                  再帰の深さも表示する */
void print_dot(int k); /* 仮引数分ドットを表示する関数 */

int main()
{
    int n; /* n! のn */
    int fac; /* 階乗 */
    printf("n = ? ");
    scanf("%d", &n);
    fac = fact(n);
    printf("%d\n", fac);
    printf("%d ! = %5d %d\n", n, fac);
    return 0;
} /* 続く */
```

10

```
int fact(int n)
{
    int fac;

    print_dot(MAX-n);          /*再帰の深さのドット表示*/
    printf("Called by n= %d ¥n",n);/*仮引数の値を表示*/
    if(n==0)
    { /*再帰の基礎部分*/
        fac=1;
        print_dot(MAX-n);      /*再帰の深さのドット表示*/
        printf("REACHED BASIS  0!=1 ¥n");
    }
    else
    { /*再帰部分*/
        fac=n*fact(n-1);
        print_dot(MAX-n);      /*再帰の深さのドット表示*/
        printf("%d != %d * ( %d! ) ¥n",n,n,n-1);
    }
    return fac;
}
/*続く*/
```

```
/*仮引数分のドットを表示する関数*/
void print_dot(int k)
{
    int i;
    for(i=0;i<k;i++)
    {
        printf("..");
    }
    return;
}
/*おしまい。*/
```

終わらない再帰関数1

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。
基礎が無い再帰関数

```
/*終わらない関数1*/  
int fact(int n)  
{  
    int fac;  
  
    fac=n*fact(n-1);  
    return fac;  
}
```

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。

出口の無い迷路のようなもの。
プログラムが停止しないときには、
^C (コントロール+'c') で
停止させましょう。

13

終わらない再帰関数2

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。
出口に近づかない再帰関数

```
/*終わらない関数2*/  
int fact(int n)  
{  
    int fac;  
  
    if(n==0)  
    {  
        fac=1;  
    }  
    else  
    {  
        fac=n*fact(n);  
    }  
    return fac;  
}
```

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。

無限増殖関数の出来上がり。

プログラムが停止しないときには、
^C (コントロール+'c') で
停止させましょう。

14

終わらない再帰関数3

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。

出口から遠ざかる再帰関数

```
/* 終わらない関数3 */
int fact(int n)
{
    int fac;

    if(n==0)
    {
        fac=1;
    }
    else
    {
        fac=n*fact(n+1);
    }
    return fac;
}
```

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。

無限増殖関数が出来上がる。

プログラムが停止しないときには、
^C (コントロール+'c') で
停止させる。

15

再帰と繰り返し

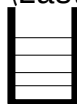
再帰的な関数は、繰り返しを用いて書き直せることがある。
再帰を用いてプログラミングするか、
繰り返しを用いてプログラミングするかは、
よく考えて決めること。

```
int fact(int n)
{
    int i;
    int fac;
    fac=1;
    for(i=1;i<=n;i++)
    {
        fac = fac*i;
    }
    return fac;
}
```

16

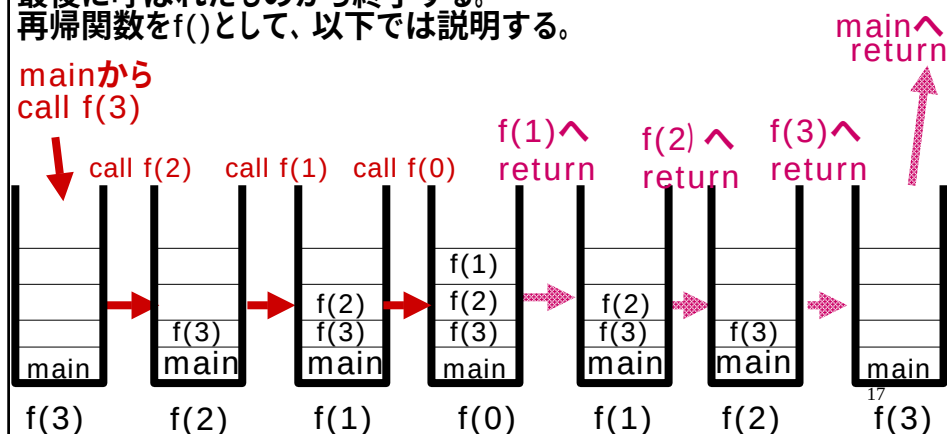
再帰とスタック

スタックとは、後入れ先出し (Last In First Out, LIFO) のデータ構造

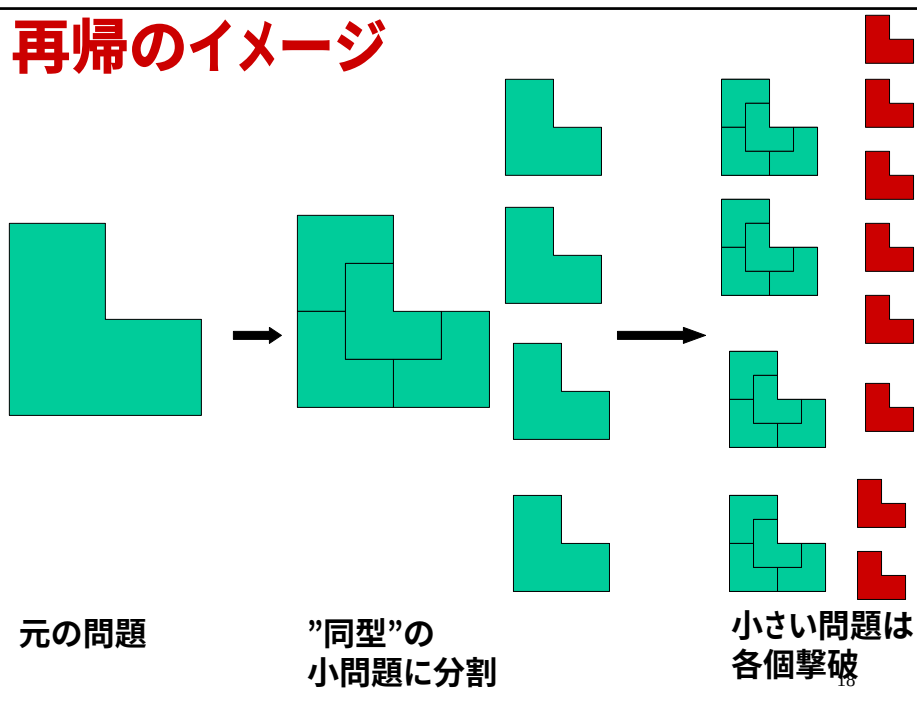


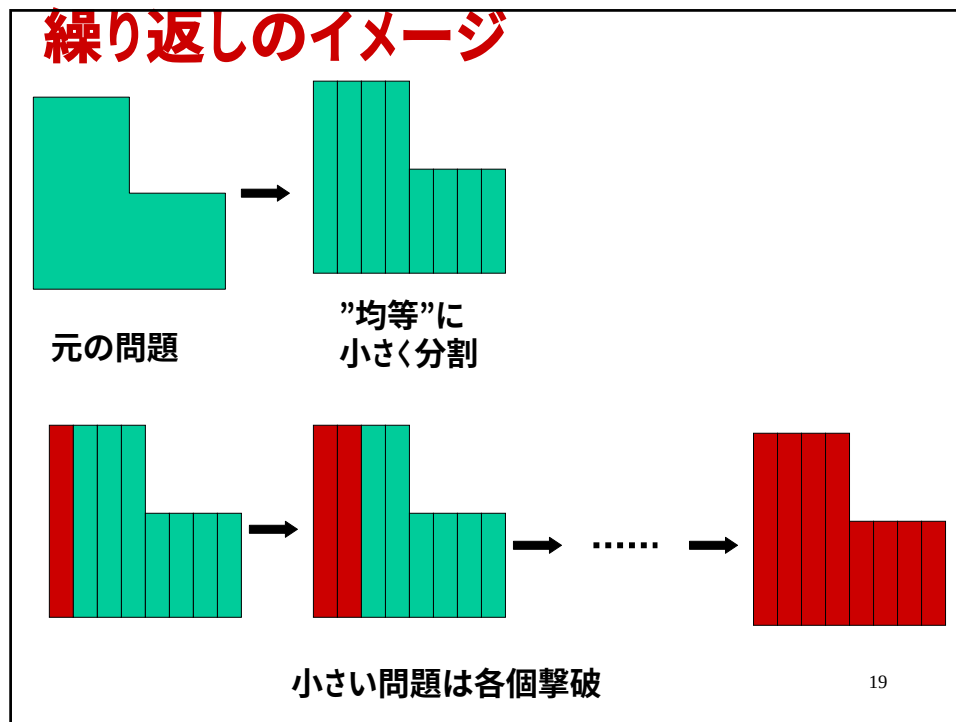
再帰関数はその呼ばれた順序にスタックにつまれ、最後に呼ばれたものから終了する。
再帰関数をf()として、以下では説明する。

mainから
call f(3)



再帰のイメージ





再帰的に階乗を求めるプログラム (p.123)

```

/*
    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:fact_rec.c
    実行ファイル:fact_rec
    説明:再帰的にn!を求める関数を用いて、
          n!を計算する。
    入力:標準入力から0から15までの整数nを入力。
    出力:標準出力にn!を出力。
*/
#include<stdio.h>
/*プロトタイプ宣言*/
int fact(int n);      /*階乗n!を求める再帰的な関数*/
/* 次のページに続く */

```

```
int    main()
{      /*ローカル変数宣言*/
    int    n;      /*n!のn*/
    int    fac;    /*階乗n!*/
    /*入力処理*/
    printf("階乗n!を計算します。¥n");
    printf("n=? ¥n");
    scanf("%d",&n);

    /* 入力値チェック */
    if(n<0||15<n)
    {
        printf("nは0から15までの整数にする。¥n");
        return -1;
    }
    /*これ以降では0から15までの整数*/
    /*続く*/
```

```
        /*    続き    */

        /*階乗n!の計算*/
        fac=fact(n);

        /*出力処理*/
        printf("%d != %10d ¥n",n,fac);

        return 0;
    }

    /*続く*/
```

```
/*階乗を求める再帰的関数
   仮引数n: n!のn,(nは0から15)
   戻り値:n! */
int fact(int n)
{
    /*ローカル変数宣言*/
    int fac;    /*階乗n!*/

    /*次に続く*/
```

23

```
/*漸化式に基づく、階乗の定義式*/
if(n==0)
{
    /*再帰の基礎、0!=1*/
    fac=1;
}
else
{
    /*再帰部分、n!=n*(n-1)!*/
    fac=n * fact(n-1);/*再帰呼び出し*/
}
return fac;
}
```

24

実行例

```
$make
gcc fact_rec.c -o      fact_rec
$ ./fact_rec
階乗n!を計算します。
n=?
5
5! =      120
$
```

```
$ ./fact_rec
階乗n!を計算します。
n=?
-1
nは0から15までの整数にする。
$
```

25

第11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。
- アドレス演算子、参照演算子の効果を理解する。
- 参照による関数呼び出しを理解する。
- 配列とポインタの関係を理解する。

☆他の関数内の2つの変数の値を交換する関数を作成し、その効果を確認するプログラムを作成する。

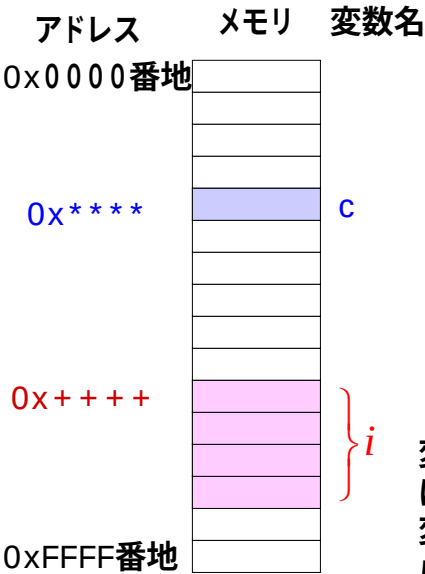
2

ポインタ

ポインタとは、
変数のアドレスを入れる変数である。

3

アドレス



コンピュータのメモリには、
すべてアドレスがある。

C言語を用いたプログラムでは、
プログラマがアドレスを管理できる。

```
char c;
```

1バイト

```
int i;
```

4バイト

変数宣言すると、その変数のためにメモリが割り当てられる。
変数名は、メモリの一區画につけられた名前である。

4

アドレス演算子 &

C言語には、変数に割り当てられたメモリの、先頭アドレスを陽に調べる演算子がある。

書式

& 変数名

&は、変数のアドレスを求めるための単項演算子。

例

```
int age;  
scanf("%d",&age);
```

```
double height;  
scanf("%lf",&height);
```

実は、scanf(の変換仕様)では、変数のアドレスを指定すると、そのアドレスが割り当てられている変数(メモリ)に標準入力から値を読み込む。

5

アドレスとscanf文

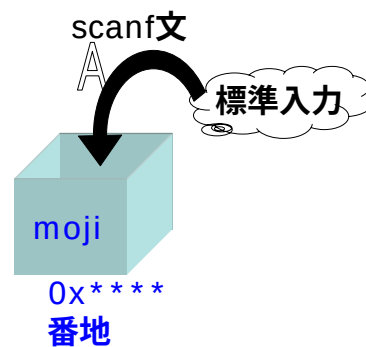
書式

```
scanf("%c",文字をいれる変数のアドレス)  
scanf("%d",整数をいれる変数のアドレス)  
scanf("%lf",実数をい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。



6

アドレスとprintf文

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
(型の区別無し)

例

変数のアドレスを表示させるには、

```
char moji;  
printf("%p",&moji);
```

16進数で表示される。

&がついているので
アドレス

変数の中身 (値) を表示させるには、

```
char moji;  
printf("%c",moji);
```

文字として表示される。

&がついていないので
中身 (値)

7

イメージ

0x****番地

A

a

0x++++番地

B

b

0x####番地

C

c

アドレス

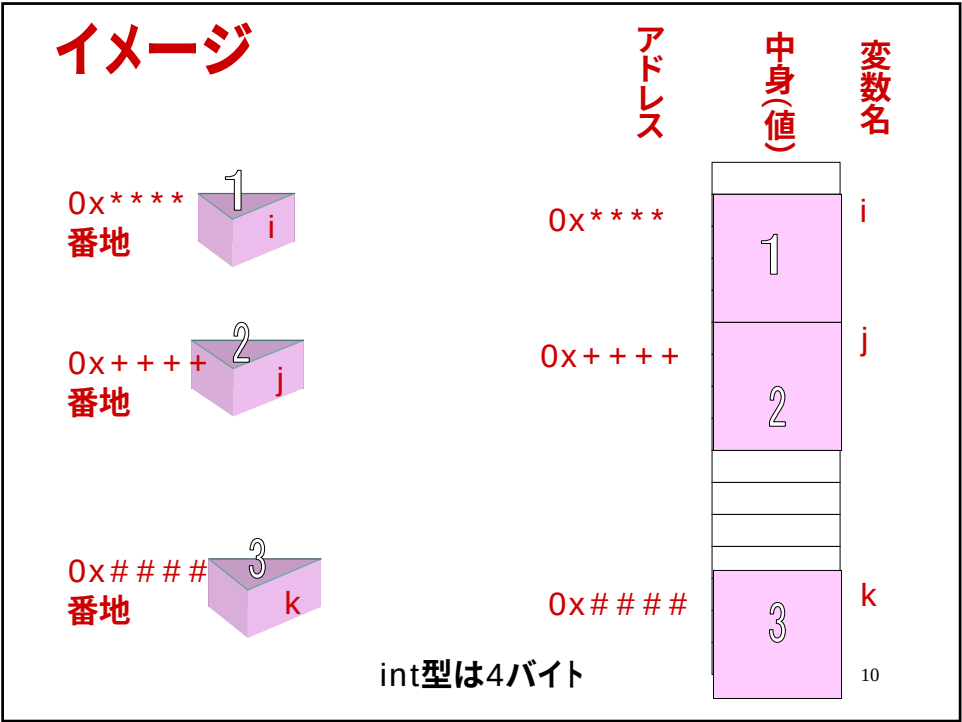
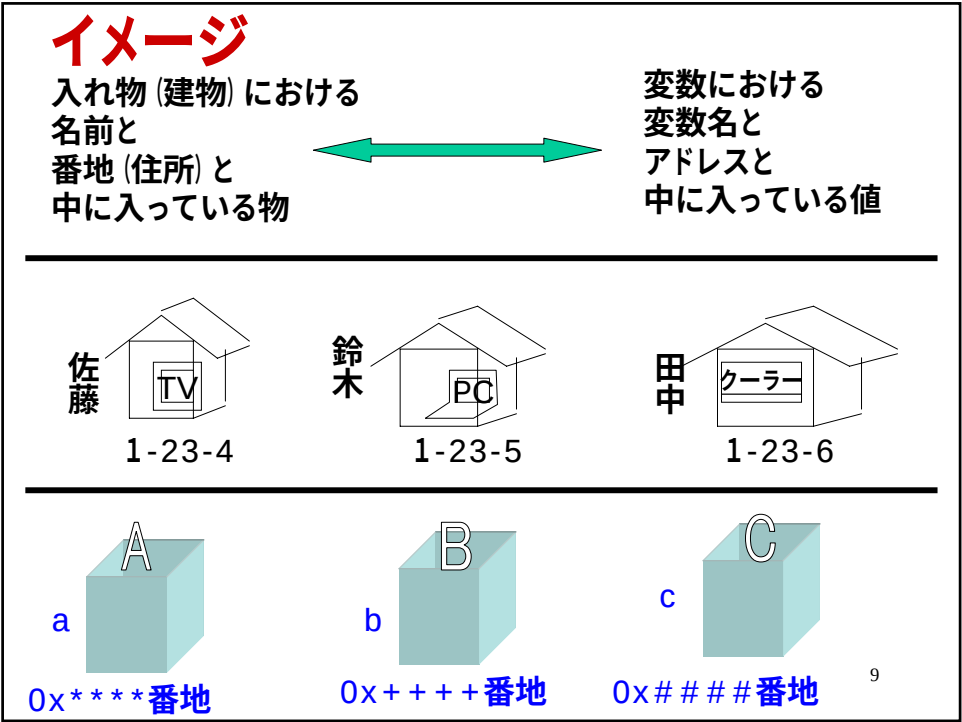
中身 値

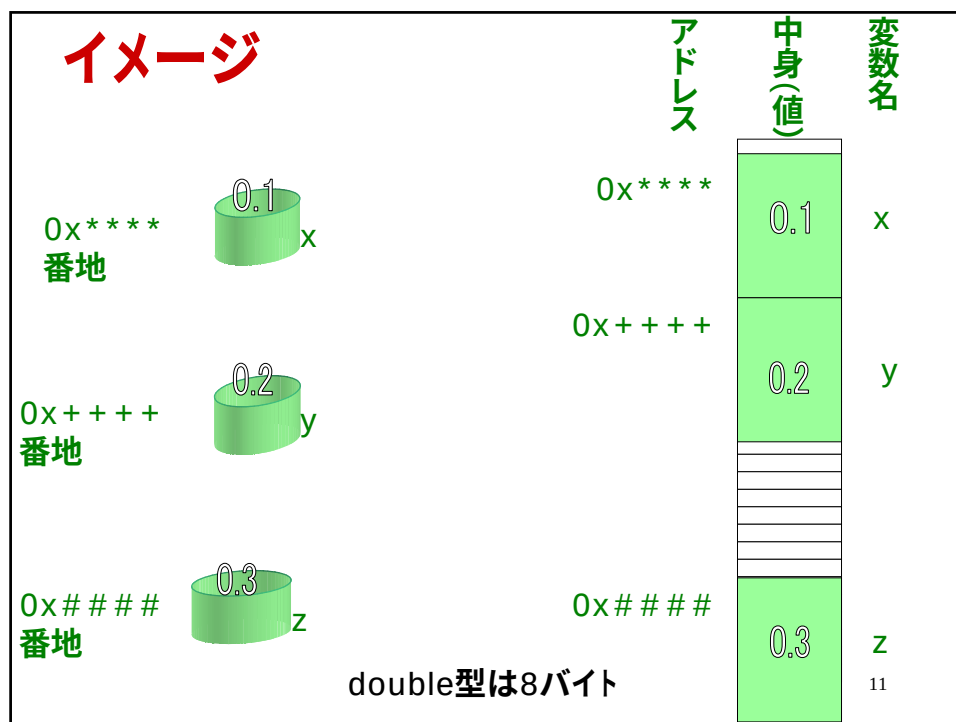
変数名

0x****	A	a
0x++++	B	b
0x####	C	c

char型は1バイト

8



**練習1**

```
/* print_address.c アドレス表示実験 */
#include <stdio.h>
int main()
{ /*変数宣言*/
    char a;
    char b;
    int i;
    int j;
    double x;
    double y;
    /*代入*/
    a='A';
    b='B';
    i=1;
    j=2;
    x=0.1;
    y=0.2;
    /* 次へ続く */
}
```

```

printf("char 型の変数のアドレス¥n");
printf("a: %p b: %p ¥n",&a,&b);
printf("char型の変数の中身¥n");
printf("a: %c b: %c ¥n",a,b);
printf("¥n");

printf("int型の変数のアドレス¥n");
printf("i: %p j: %p ¥n",&i,&j);
printf("int型の変数の中身¥n");
printf("i: %d j: %d ¥n",i,j);
printf("¥n");

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

```

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

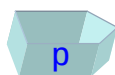
データ型 *ポインタの名前;

例

char *p;

int *q;

double *r;



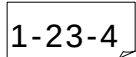
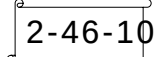
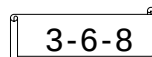
文字型の変数の
アドレス専用

整数型の変数の
アドレス専用

実数型の変数の
アドレス専用

ポインタ=変数のアドレスを入れるための入れ物
(ただし、用途別)

14

イメージ			変数 (の型) アドレス (ある型の変数を指す) ポインタ
(種類別の) 建物 住所 (種類別の) アドレス帳	対応		
 1-23-4  1-23-5  1-23-6			民家専用の アドレス帳 
 2-46-8  2-46-9  2-46-10			工場専用の アドレス帳 
 3-6-9  3-6-8  3-6-7			店専用の アドレス帳 

15

間接演算子 *

(ポインタとポインタが指す変数)

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。

例

```
int    i;
int    *p; /* ポインタ */

p = (&i);
/* pにはi のアドレスが入る */
```

ポインタpがある変数yのアドレスを蓄えているとき、
ポインタpは変数yを指すという。
あるいは、pは変数yへのポインタであるという。

16

ポインタと変数の別名

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。

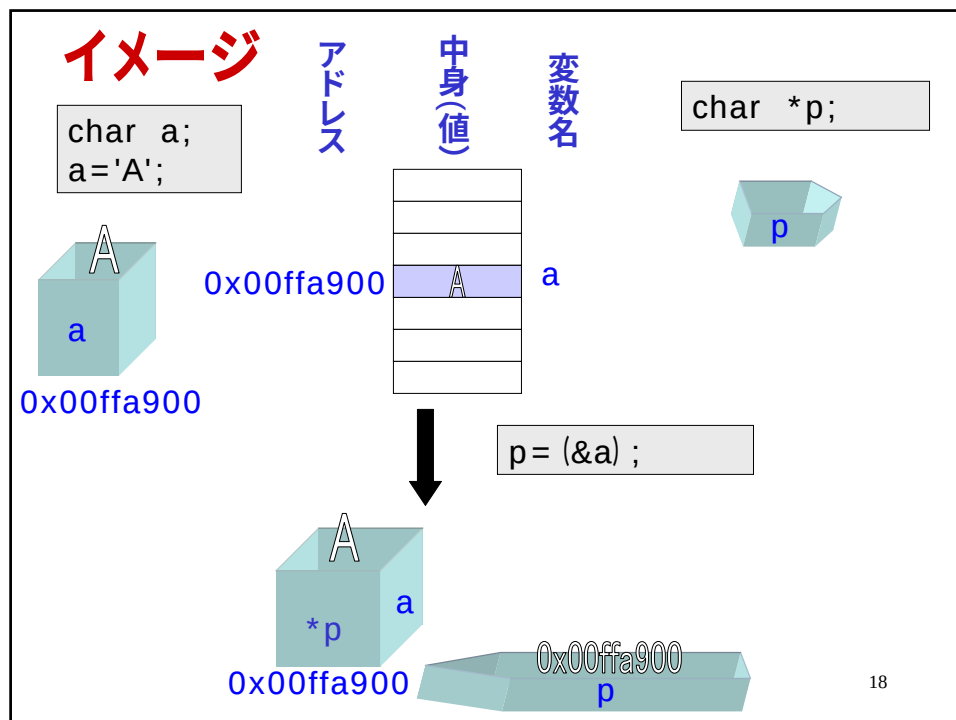
```
char a;
char b;
char *p;

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。 */
```

17



18

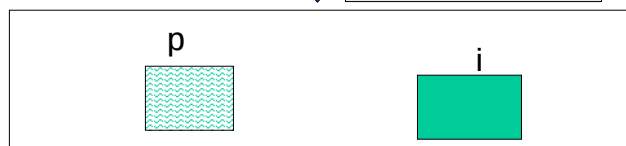
イメージ

プログラムを図で説明するときには、
アドレスを数字で表さずに、
矢印でポインタをあらわすことがある。

変数宣言 (変数の用意)



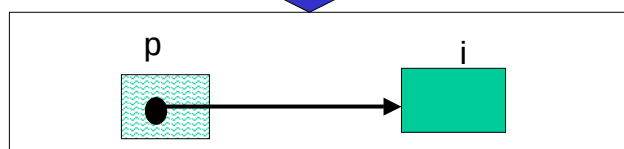
```
int i;  
int *p;
```



アドレス代入



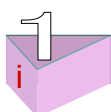
```
p = (&i);
```



19

イメージ

```
int i;  
i = 1;
```



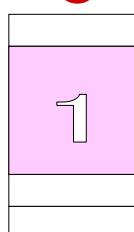
0x00ffba8

アドレス

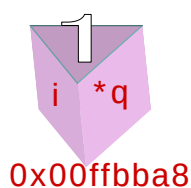
中身(値)

変数名

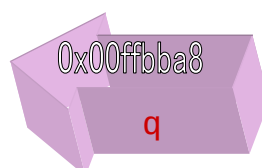
0x00ffba8



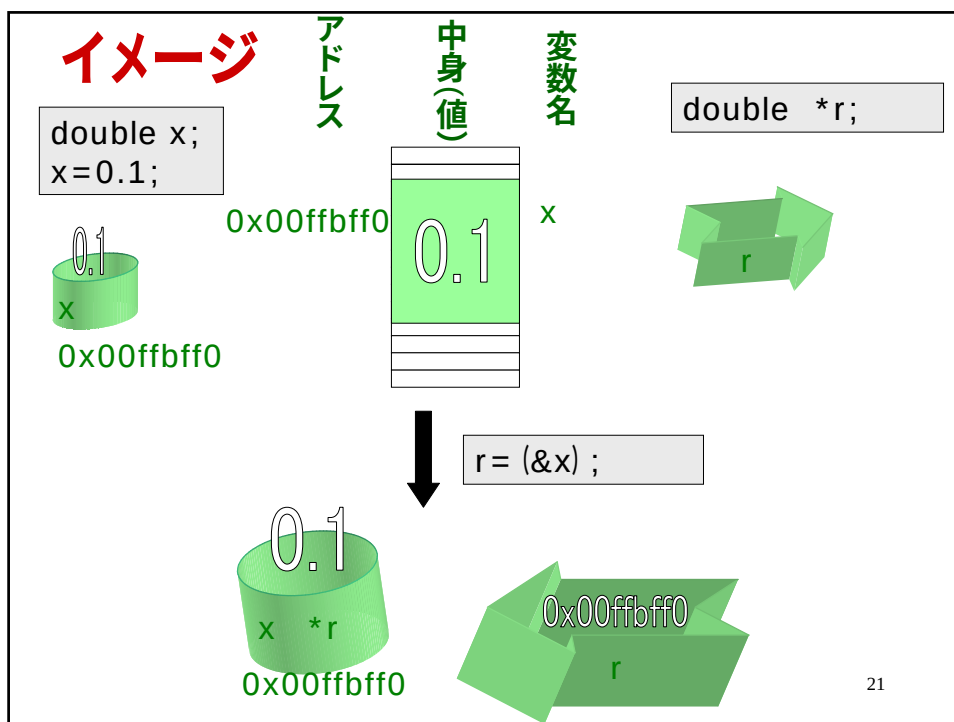
```
q = (&i);
```



0x00ffba8



20



NULL

C言語では、どの変数も指さない特別なアドレスがあり、**NULL**として表す。(スタイル規則参照)

```
int    *p; /*int型を指すポインタ*/
double *q; /*double型を指すポインタ*/
```

```
/*ポインタへNULLを代入*/
p=NULL;
q=NULL;
```

このように、
どんな型の変数を指す
ポインタへもNULLを代入
できます。

22

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;

    /*代入*/
    i=1;
    j=2;

    /*    次へ続く    */
}
```

23

```
/*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*/

/*続き*/
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\n",*p);

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\n",*q);
/*    次へ続く    */
```

24

```

/*続き*/
/*ポインタによる演算*/
(*q)=(*q)+(*p);
printf("( *q)=( *q)+( *p);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¥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;
}

```

25

配列とポインタ

C言語では、配列名は先頭の要素のアドレスを指す。

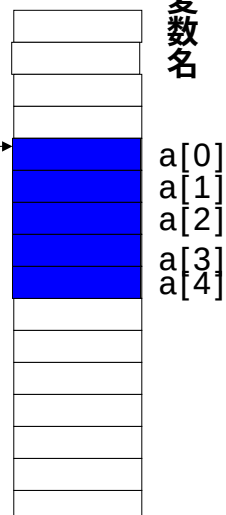
例えば、

```
#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。

配列名

a



26

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

配列名

a

p

ポインタ
(配列の先頭要素の
アドレスを持つ)

とすると、
a[i]とp[i]は同じ変数 (配列要素)
を表す。

a[0]	p[0]
a[1]	p[1]
a[2]	p[2]
a[3]	p[3]
a[4]	p[4]

27

イメージ

```
char a[3];
a[0]='A';
a[1]='B';
a[2]='C';
```

アドレス

変数名

中身(値)

0x00ffaa00	A	a[0]
0x00ffaa01	B	a[1]
0x00ffaa02	C	a[2]

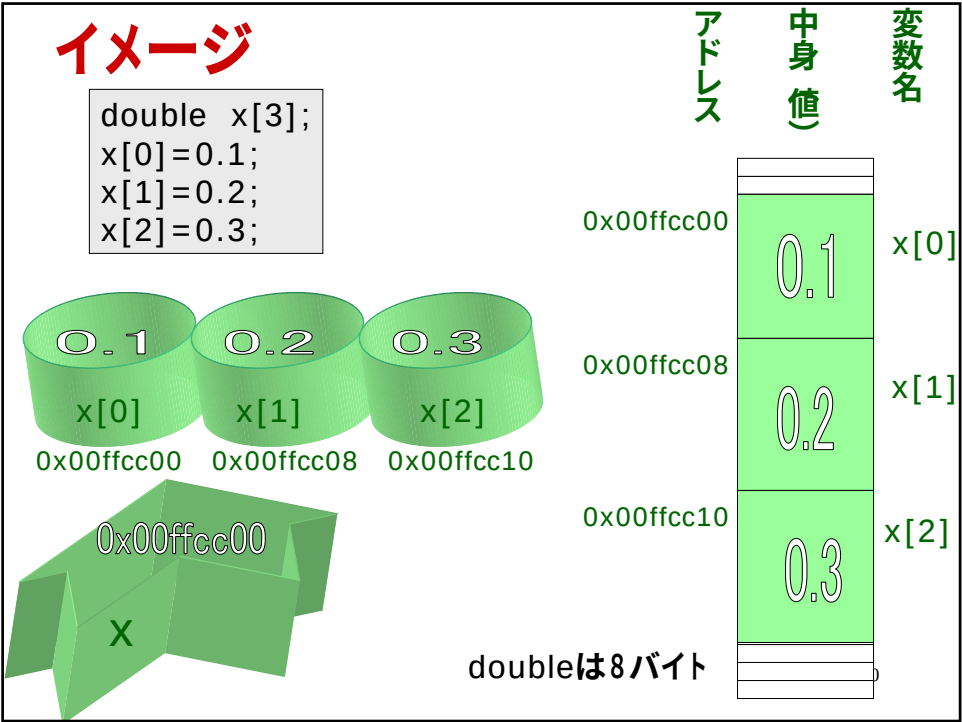
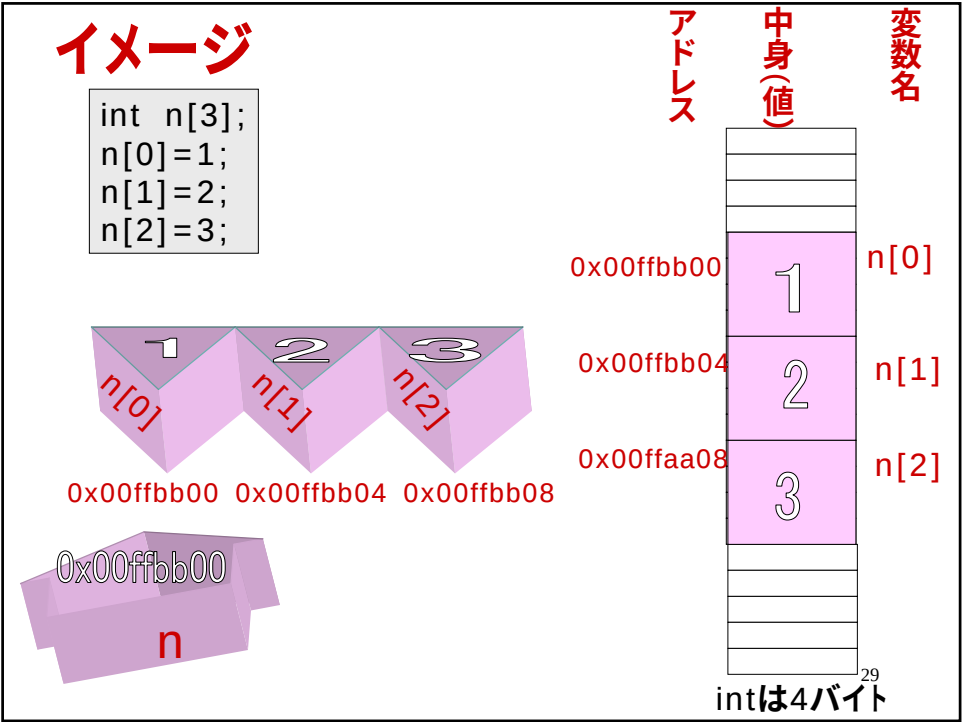
0x00ffaa00 0x00ffaa01 0x00ffaa02

0x00ffaa00

a

28

charは1バイト



練習3

```
/* pointer_array.c ポインタと配列実験 コメント省略 */
#include <stdio.h>
#define MAX 5
int main()
{
    int i;
    int n[MAX]; /*データを入れる配列*/
    int *p; /*上の配列の先頭を指すポインタ*/

    for(i=0;i< MAX;i++)
    {
        n[i]=i+1;
    }

    /* 次が続く */
}
```

31

```
/* 続き */
m_ptr=n;
printf("nの値は %p ¥n",n);
printf("n[0]のアドレスは%p ¥n",&n[0]);
printf("m_ptrの値は%p ¥n",p);
printf("¥n¥n");

printf("&n[i] n[i] p[i]¥n");
for(i=0;i<MAX;i++)
{
    printf("%p %d %d ¥n",
        &n[i],n[i],p[i]);
}
return 0;
}
```

32

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。

++ > & > / > +
-- > *(単項演算子) > *(2項演算子) > -

```
*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。

33

関数とポインタ

2つの関数間の引数の受け渡しに、ポインタは重要な役割を果たす。
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

```
void    関数名 (仮引数の型 * 仮引数名)
{
    return;
}
```

例

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

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



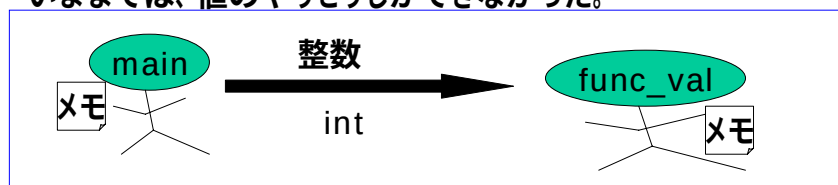
“ほぼ”同一
の効果

```
int func_val(int k)
{
    return k;
}
```

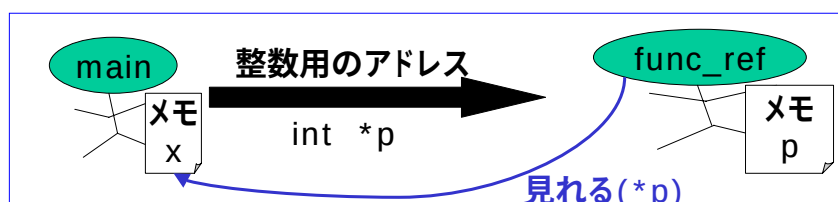
```
int main()
{
    i=func_val(i);
    a=i;
}
```

イメージ

いままでは、値のやりとりしかできなかった。



ポインタの仮引数を用いると、アドレスのやりとりができる。

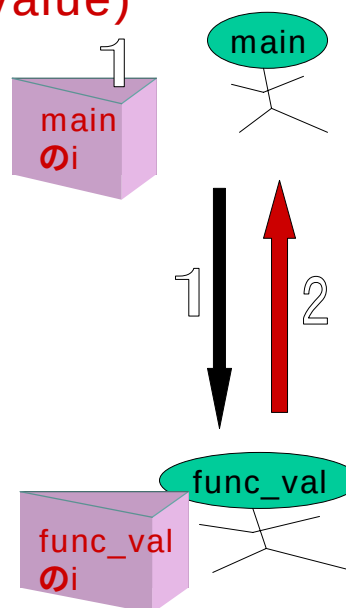


アドレスのやりとりをすると、
他の関数内のローカル変数の内容を変更できる。

35

値による呼び出し(call by value)

```
/*test_cbv.c*/
#include <stdio.h>
int func_val(int);
int main()
{
    int i;
    int j;
    i=1;
    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;
}
```



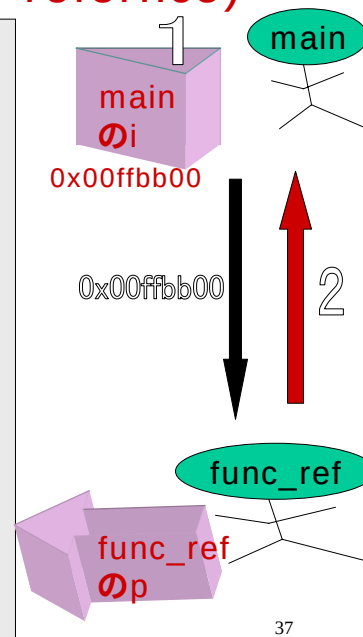
36

参照による呼び出し(call by reference)

```

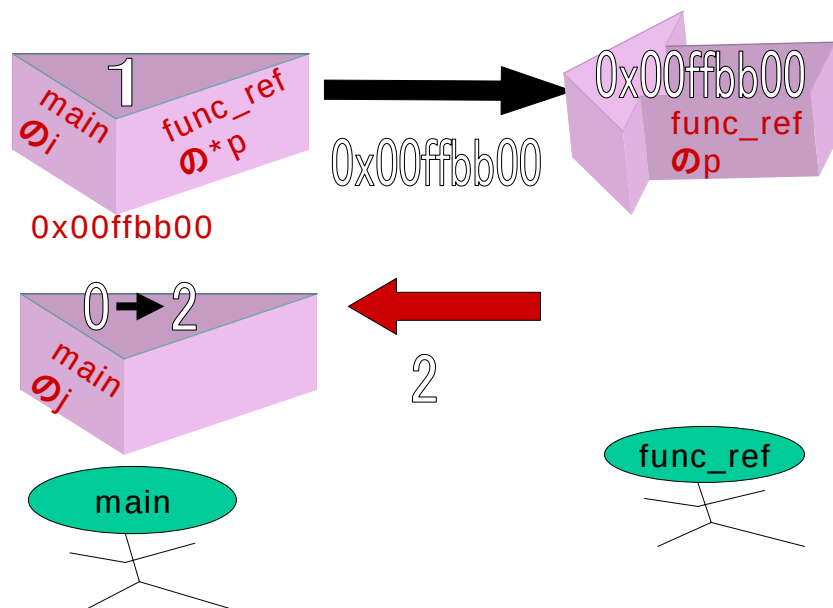
/*test_cbr.c*/
#include <stdio.h>
int func_ref(int *);
int main()
{
    int i;
    int j;
    i=1;
    j=0;
    printf("i= %d j= %d\n",i,j);
    j=func_ref(&i);
    printf("i= %d j= %d\n",i,j);
    return 0;
}
int func_ref(int *p)
{
    (*p)++;
    return (*p);
}

```



37

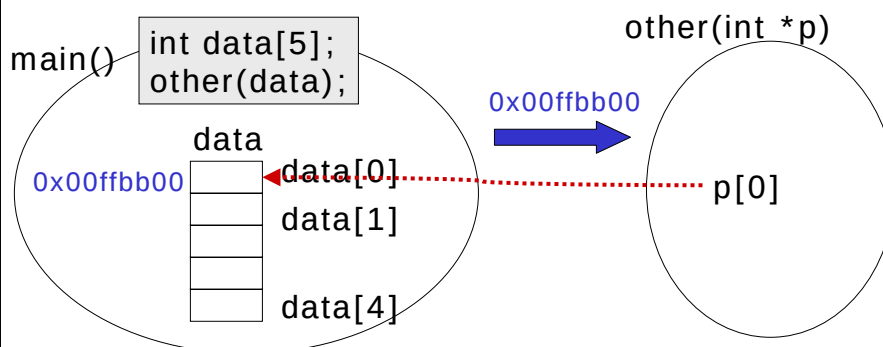
イメージ



38

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]`)を渡すことができる。配列名が配列の先頭アドレスを保持していることに注意する。

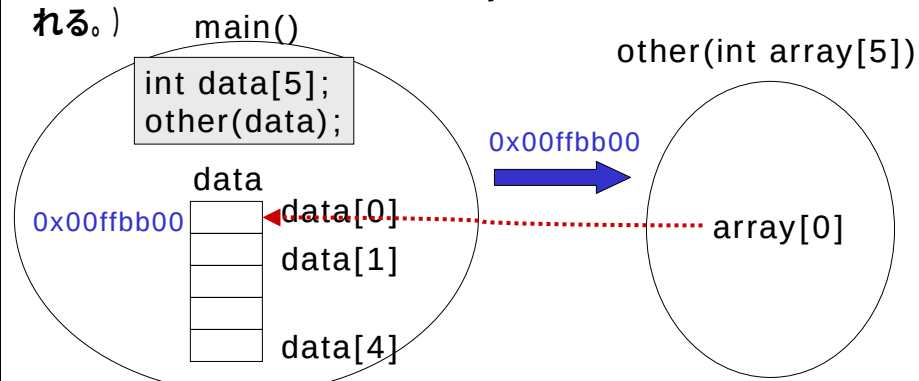


受け取った他の関数の中では、配列の先頭要素のアドレスの入ったポインタを配列名のように使うことができる。

39

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。

40

練習4

```
/*test_sendarray.c*/
#include <stdio.h>
#define MAX 10
void print_array(int n,int *p);
void write_array(int n,int *p);
int main()
{
    int i;
    int n;
    int data[MAX];

    for(i=0;i<MAX;i++)
    {
        data[i]=i;
    }

    printf("n=?");
    scanf("%d",&n);
    /* 次が続く */
```

41

```
/*続き*/
print_array(n,data);

/*内容変更*/
write_array(n,data);

print_array(n,data);
return 0;
}
/*main関数終了*/

/*次が続く*/
```

42

```
/*    続き    */
void print_array(int n,int *p)
{
    int i;
    for(i=0;i<n;i++)
    {
        printf("data[%d] = %d ¥n",i,p[i]);
    }
    printf("¥n");
    return;
}

/*    次に行く    */
```

注意:
呼ばれる方では、配列の最後に気を付ける事。

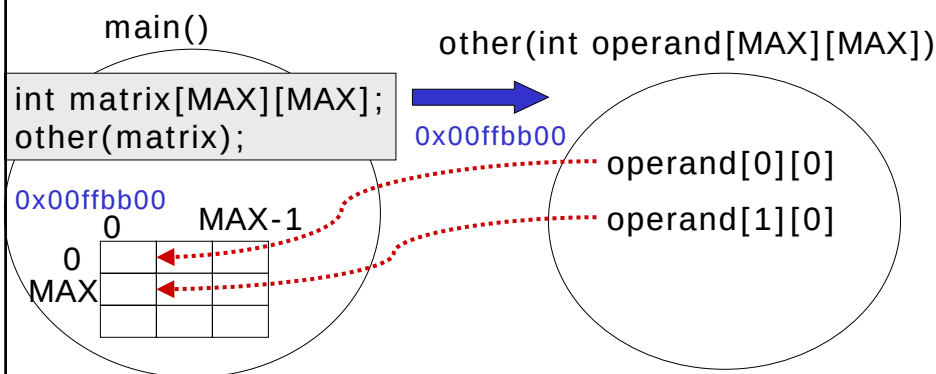
43

```
/*    続き    */
void write_array(int n,int *p)
{
    int i;
    for(i=0;i<n;i++)
    {
        p[i]=(p[i])*2;
    }
    return;
}
/*全てのプログラム終了*/
```

44

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。

45

他の関数内の2つの変数の値を交換する関数

```
/*
    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:pointer_swap.c
    実行ファイル:pointer_swap
    説明:参照呼び出しの効果を調べるための
          プログラム。ポインタを用いて、他の関数内
          の2つの変数の値を交換する関数を作成し
          利用する。
    入力:標準入力から2つの整数を入力。
    出力:標準出力にその2つの整数を交換して出力する。
*/
```

46

```
/* 続き */
/* ヘッダファイルの読み込み */
#include <stdio.h>

/* マクロの定義 */
/* このプログラムでは、マクロは用いない。 */

/* グローバル変数の宣言 */
/* このプログラムでは、グローバル変数はいらない。 */

/* プロトタイプ宣言 */
void swap(int *p, int *q);
/* 参照呼び出しを用いて、
   2つの変数の値を入れ替える関数 */
/* 次のページに続く */
```

47

```
/* 続き */
/* main関数 */
int main()
{
    /* ローカル変数宣言 */
    int data1; /* 入力整数1 */
    int data2; /* 入力整数2 */

    /* 入力処理 */
    printf("data1=?");
    scanf("%d",&data1);
    printf("data2=?");
    scanf("%d",&data2);
}
```

48

```

/*    続き    */

/* 2つの変数の値の交換 */
swap(&data1,&data2); /*関数swapの引数1、引数2
                     とも参照呼出しなので、
                     アドレスを与える。*/

/*データ出力*/
printf("data1= %d¥n",data1);
printf("data2= %d¥n",data2);

return 0; /*正常終了*/
}
/* main関数終了 */

/*    次に行く    */

```

49

```

/*他の関数の2つの変数の値を交換する関数
仮引数 p,q:交換すべき整数型の変数のアドレス
(参照呼出なので、呼び出し側ではアドレスを指定する)
戻り値:なし
*/
void swap(int *p, int *q)
{
    /*ローカル変数宣言*/
    int temp; /*2つの変数の値を交換するために、
              値を一時的に蓄えておく変数。*/

    /*処理内容の通知*/
    printf("交換中 ¥n");

    /*    次に行く    */
}

```

50

```
/*    続き    */

/*    命令記述    */
temp=(*p);
(*p)=(*q);
(*q)=temp;

return;
}
/*swap関数終了*/
/*全プログラム (pointer_swap.c)終了*/
```

51

実行例

```
$/pointer_swap
data1=? 1
data2=? 3
交換中
data1= 3
data2=1
$
```

52

第12回新しい型と構造体



1

今回の目標

- 新しい型の定義法を理解する。
- 構造体を理解する。

☆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 $z_1 = a_1 + b_1i$ と $z_2 = a_2 + b_2i$ の和 $z_3 = a_3 + b_3i$ は、次式で与えられる。

$$\begin{aligned} z_3 &= z_1 + z_2 \\ &= (a_1 + a_2) + (b_1 + b_2)i \end{aligned}$$

3

typedef文 (型の別名の付け方)

C言語では、いろいろなデータ型に自分の好きな名前をつけることができる。

宣言

```
typedef 既に使える型 別名;
```

例

```
typedef int Count;
Count i;
```

int, double, charや
int *, double *, char*や
既に定義した型

この宣言で、
Count型 (int 型)
の変数iが用意される。

セミコロンを
忘れずに。

参考 `int i;`

新しい型名は、
大文字+小文字で命名すること。
(スタイル規則参照)

4

練習1

```
/*newtype.c 新しい型実験 コメント省略*/
#include <stdio.h>

#define SIZE 5

typedef int Count; /*序数を表す型*/
typedef double Real; /*実数を表す型*/

int main()
{
    /*ローカル変数宣言*/
    Count i; /*カウンタ*/
    Real x[SIZE]; /*実数データ*/
    /*次に続く。*/
```

```
/*続き*/
/*ローカル変数の初期化*/
i=0;
for(i=0;i<SIZE;i++)
{
    x[i]=(Real) i;
    /*全て異なる値にするため、インデックスで初期化。
    カウント型から実数型へキャストしている。*/
}

/*出力処理*/
for(i=0;i<SIZE;i++)
{
    printf("x[%d] = %6.2f ¥n",i,x[i]);
}
return 0;
}
```

構造体

(レコード型と呼ぶこともある。)

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。

一まとまりのデータ例

複素数: 実部と虚部

点: x座標、y座標

2次元ベクトル: x成分、y成分

名刺: 所属、名前、連絡先

日付: 年、月、日、曜日

本: 題名、著者、ISBN

7

構造体テンプレートの宣言 (構造体の定義)

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
セミコロンを忘れない。

8

構造体の宣言

(構造体型の変数の用意の仕方)

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;
```

9

typedef文と構造体

(構造体型の変数の用意の仕方2)

構造体型の宣言は、
2つの文字列で一つの型を意味するので紛らわしい。
typedef文を使うとすっきりする。

宣言

```
typedef struct 構造体タグ 別名;
```

これ2つで、一つの型

例

```
typedef struct complex Complex;
Complex z;
```

別名

参考

```
int i;
```

この宣言で、Complex型 (複素数型) の変数zが用意される。

構造体のイメージ



char



int



double

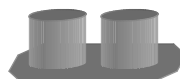
既存の型

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

11

別名の定義

```
typedef struct complex Complex;
```

struct complex型の雛形

2つの文字列で、
一つの型を表すと、
間違いやすい。

Complex型の雛形



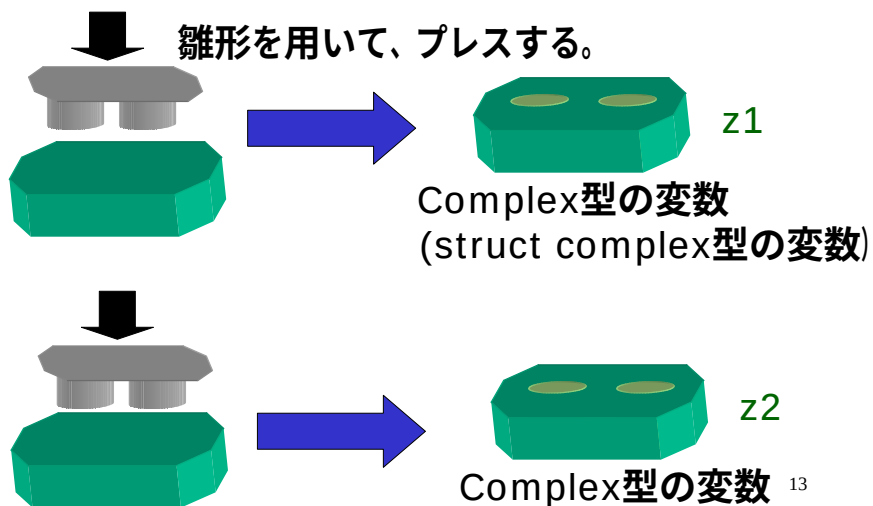
12

構造体型の変数宣言

```
Complex z1;
Complex z2;
```



Complex型の雛形



構造体のイメージ2



char



int



double

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

雛形の作成。



struct card 型の雛形

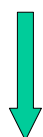
いろいろな型のデータを
一まとまりであつかうときには、
構造体はとくに便利。

14

別名の定義

```
typedef struct card Card;
```

struct card 型の雛形



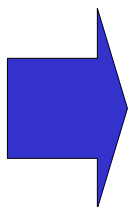
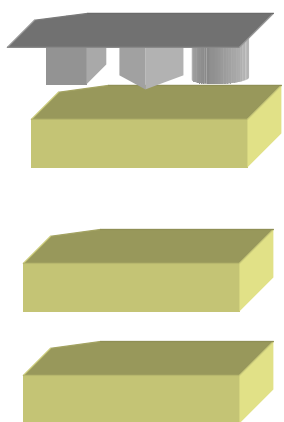
Card型の雛形



15

構造体型の配列宣言

```
#define MAXCARD 3  
Card x[MAXCARD];
```



Card型の変数



x[0]



x[1]



x[2]

16

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

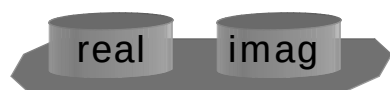
これはchar 型の変数である。

17

参照のイメージ

Complex z1;

Complex型の雛形



Complex型の変数



z1



z1.real



z1.imag

18

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。

$\cdot$ $>$ $++$ $>$ $*$
 $--$ $\&$
`x[0].age++;` は `(x[0].age)++;` の意味

`Complex * p;` のとき、

`*p.real;` は `*(p.real);` の意味になってしまう

両方間違い。(メンバrealは、ポインタではない。)

`(*p).real;`

典型的には、
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。

19

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。

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

間違い例

×

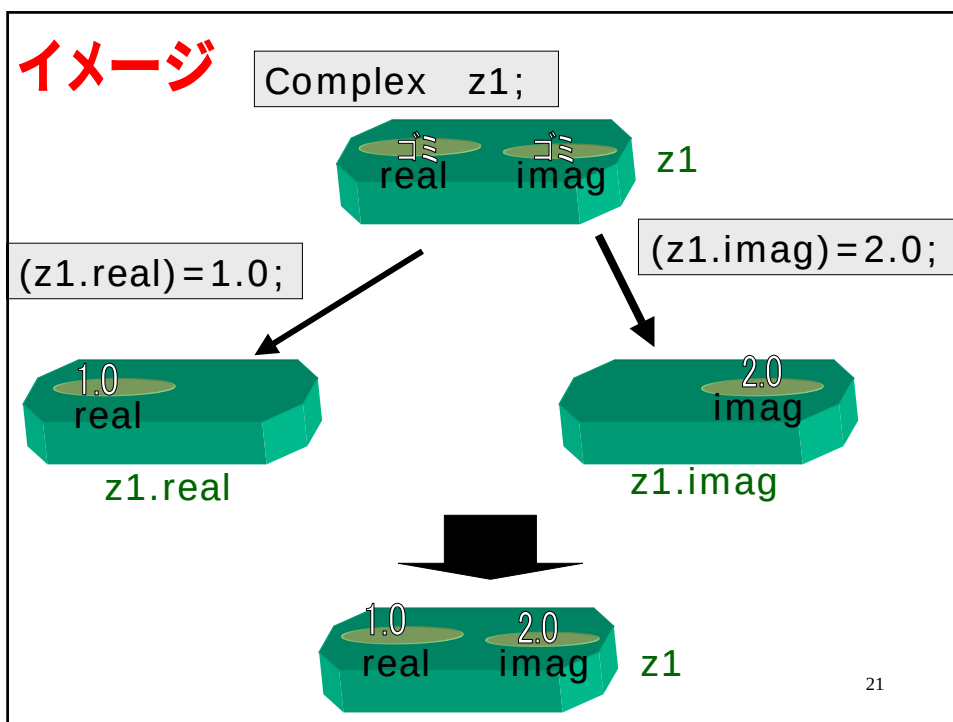
`z1 = 1.0 + 2.0i;`

`z1 = (1.0, 2.0);`

複素数だからって
こんなふうには
かけない。

ベクトル風にも
かけない。

20



構造体と代入演算子2 (構造体への値の入れ方2)

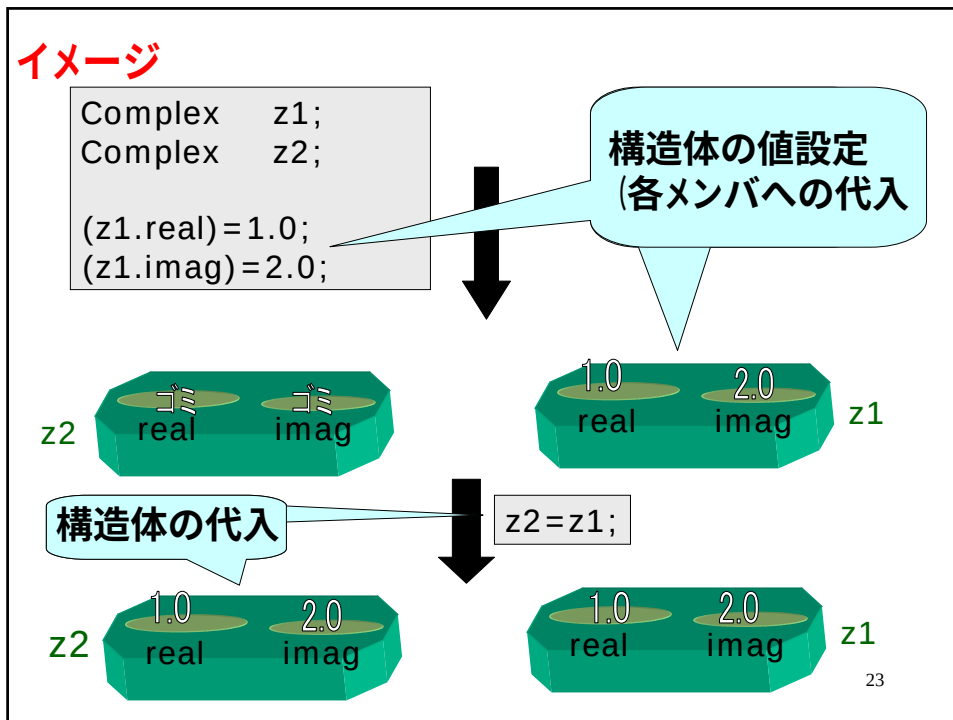
同じ型の構造体同士で代入する。

```
Complex z1;
Complex z2;

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

z2 = z1;
```

22



練習2

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

struct complex
{
    double real;
    double imag;
};

typedef      struct complex  Complex;

/* 次が続く */
```

24

```
int    main()
{
    Complex    z1;
    Complex    z2;

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

25

```
/*    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;
}
```

26

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:B0zB0xx
    ソースファイル:pluscomp.c
    実行ファイル:pluscomp
    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。
*/
/* 次に行く */

```

27

```

/* 続き */
#include <stdio.h>

/* 構造体テンプレート */
struct complex /*複素数を表わす構造体*/
{
    double real; /*実部*/
    double imag; /*虚部*/
};

/* 新しい型名 */
typedef struct complex Complex; /*複素数型*/

/* プロトタイプ宣言*/
Complex plus_complex(Complex,Complex);
/*2つの複素数の和を求める関数*/

/* 次に行く */

```

28

```
/*main関数*/
int main()
{
    /*ローカル変数宣言*/
    Complex z1; /*複素数1*/
    Complex z2; /*複素数2*/
    Complex sum; /*複素数の和を蓄える変数*/
    /* 入力処理*/
    printf("2つの複素数z1,z2を入力して下さい。¥n");
    printf("z1の実部は?");
    scanf("%lf",&(z1.real));
    printf("z1の虚部は?");
    scanf("%lf",&(z1.imag));
    printf("z2の実部は?");
    scanf("%lf",&(z2.real));
    printf("z2の虚部は?");
    scanf("%lf",&(z2.imag));
    /* 次が続く */
}
```

```
/* 続き */
/*計算処理*/
sum=plus_complex(z1,z2);

/*出力処理*/
printf("sum=z1+z2 ¥n");
printf("( %4.2f+ (%6.2f)i )=",
        sum.real,sum.imag);
printf("( %4.2f+ (%4.2f)i )+",
        z1.real,z1.imag);
printf("( %4.2f+ (%4.2f)i)¥n",
        z2.real,z2.imag);
return 0; /*正常終了*/
}
/*main関数終了*/
```

```
/* 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)
*/
Complex plus_complex(Complex z1,Complex z2)
{
    /* ローカル変数宣言 */
    Complex sum; /* 2つの複素数の和を蓄える */
    /* 命令記述 */
    (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plus_complexの終了 */
/* プログラムpluscomp.c の終了 */
```

実行結果

```
$make
gcc pluscomplex.c -o pluscomplex
$./pluscomplex
2つの複素数z1,z2を入力して下さい。
z1の実部は? 1.0
z1の虚部は? 2.0
z2の実部は? 3.0
z2の虚部は? 4.0
sum=z1+z2
(4.00+(6.00)i)=(1.00+(2.00)i)+(3.00+(4.00)i)
$
```