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

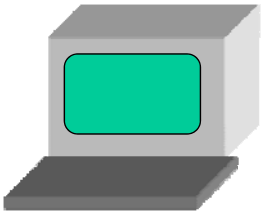
サポートページ：
http://www.ec.h.akita-pu.ac.jp/programming/

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

- 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
- GCC（GNU C コンパイラ）
  C言語で記述されたプログラムを
  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

次回

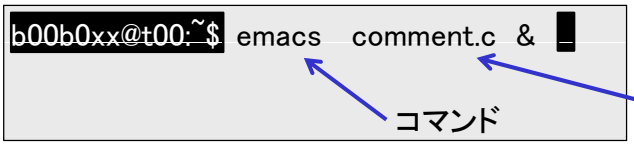
5

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

6

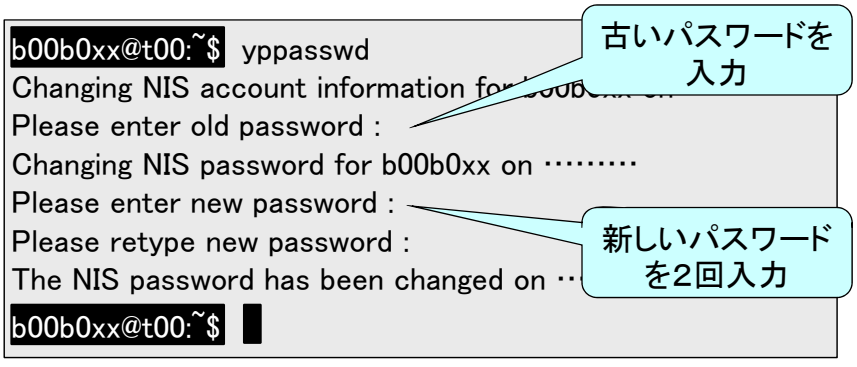
コマンドの実行



- 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

7

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

8

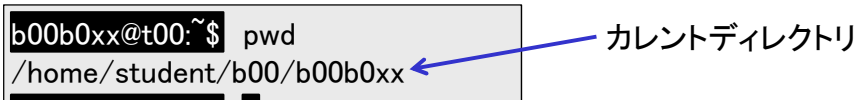
パスワードの付け方

- 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

9

カレントディレクトリ

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



- 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

10

ディレクトリの作成

```
b00b0xx@t00:~$ mkdir sample
b00b0xx@t00:~$
```

カレントディレクトリに sample という名前のディレクトリを作成

- 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

11

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

12

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

13

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

14

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

15

ディレクトリを利用したファイル整理の例

```
b00b0xx@t00:~$ mkdir T01
b00b0xx@t00:~$ cd T01
b00b0xx@t00:~/T01$ emacs comment.c &
b00b0xx@t00:~/T01$ ls
comment.c
b00b0xx@t00:~/T01$
```

「木曜クラスの1回目」という意味で、「T01」という名前のディレクトリを作成した例

Emacs で上書き保存すると自動的にカレントディレクトリに保存される

- 毎回の演習の際に、その日作成したファイル等をすべて保存するための、専用のディレクトリを作成すると良い

16

Emacs の起動

```
b00b0xx@t00:~/T01$ emacs  comment.c  &
b00b0xx@t00:~/T01$ ls
comment.c
b00b0xx@t00:~/T01$
```

Emacs で上書き保存すると自動的にカレントディレクトリに comment.c という名前のファイルが保存される

- コマンドの引数として、保存すべき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

17

課題の提出

```
b00b0xx@t00:~/T01$ ls
comment.c
b00b0xx@t00:~/T01$ submit  T01  1  comment.c
課題番号:  T01
問題番号:  1
comment.c  を提出します。
comment.c  を提出しました（… … …
```

提出すべきファイルがカレントディレクトリにあるか必ず確認

課題番号が T01
問題番号が 1
であった場合の例

- submit コマンド（この演習室専用）を利用
- コマンドの引数として、課題番号、問題番号、提出ファイル名を指定

18

提出の確認

```
b00b0xx@t00:~/T01$ check  T01   1
課題番号:  T01
問題番号:  1
提出されたファイルの履歴を確認します。
-----------------------------------
Tue Apr  1 16:18:20 JST 2008  comment.c  b00b0xx
-----------------------------------
```

- check コマンド(この演習室専用)を利用
- コマンドの引数として、課題番号、問題番号を指定

19

提出内容の確認

```
b00b0xx@t00:~/T01$ check  T01   1    comment.c
課題番号:  T01
問題番号:  1
提出されたファイルの履歴を確認します。
-----------------------------------
Tue Apr  1 16:18:20 JST 2008  comment.c  b00b0xx
-----------------------------------
```

提出したファイル名を指定

提出したファイルの内容が表示される

- コマンドの引数として、課題番号、問題番号、提出ファイル名を指定

20

ヒントの確認

```
b00b0xx@t00:~/T01$ check  T01  1S  README
課題番号:  T01
問題番号:  1S
提出されたファイルの履歴を確認します。
-----------------------------------
Tue Apr  1 16:18:20 JST 2008  comment.c  b00b0xx
-----------------------------------
```

課題のヒントが表示される

問題番号の後に「S」を付ける

- コマンドの引数として、課題番号、問題番号（Sを付ける）、「README」を指定

21

課題の再提出

```
b00b0xx@t00:~/T01$ ls
comment.c
b00b0xx@t00:~/T01$ submit  T01  1R  comment.c
課題番号:  T01
問題番号:  1R
comment.c  を提出します。
comment.c  を提出しました(… … … )
```

提出すべきファイルがカレントディレクトリにあるか必ず確認

問題番号の後に「R」を付ける

- submit コマンド（この演習室専用）を利用
- コマンドの引数として、課題番号、問題番号（Rを付ける）、提出ファイル名を指定

22

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

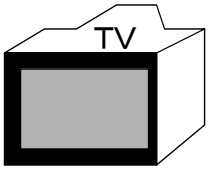
ソフトウェア

プログラム

番組

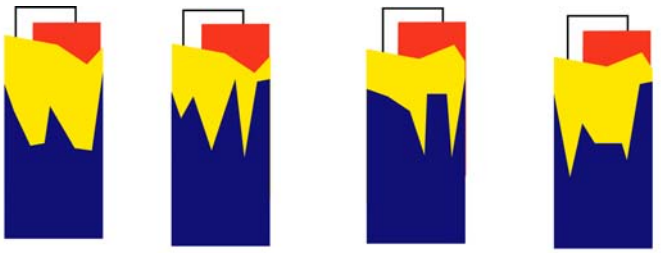
思考、記憶

ハードウェア

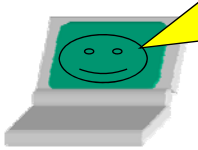


23

ハードウェアとソフトウェアの階層構造

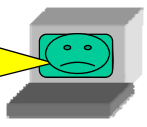


ハードウェア　基本ソフトウェア（OS）　応用ソフトウェア（アプリケーション）　利用者の生成物（ファイル等）

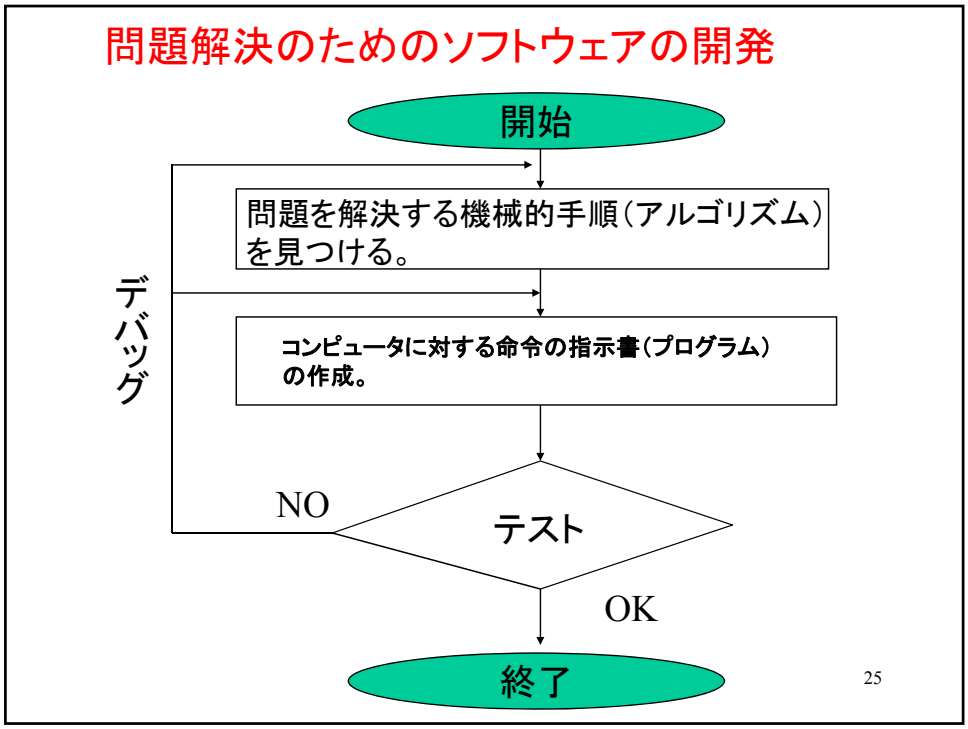


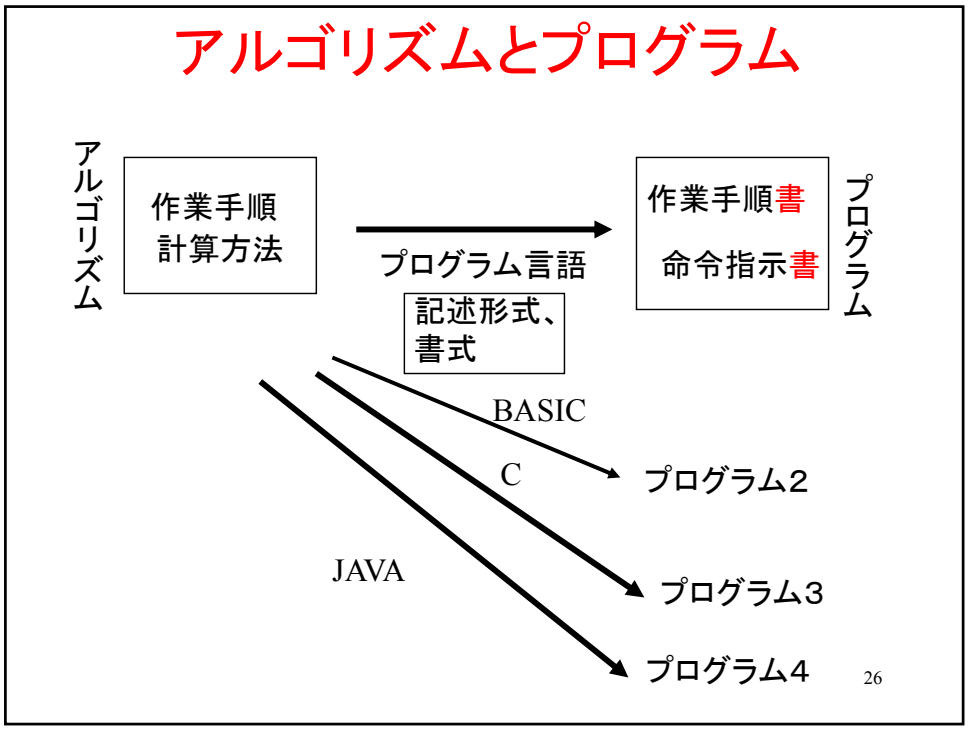
自分用の0と1の並びしかわからないよ。

他のコンピュータは、別の0と1の並びを用いてるかも。



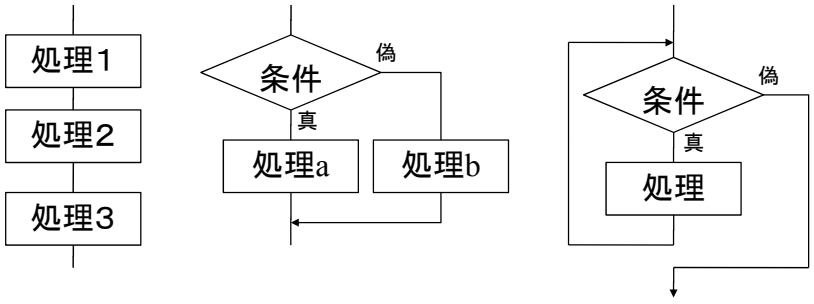
24


問題解決のためのソフトウェアの開発
開始
問題を解決する機械的手順（アルゴリズム）を見つける。
コンピュータに対する命令の指示書（プログラム）の作成。
テスト
NO
OK
デバッグ
終了
25


アルゴリズムとプログラム
アルゴリズム
作業手順
計算方法
プログラム言語
記述形式、書式
作業手順書
命令指示書
プログラム
BASIC
C
JAVA
プログラム2
プログラム3
プログラム4
26

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）

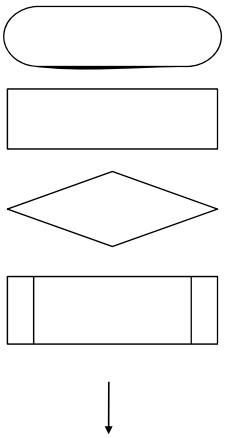


順次　　選択　　反復

27

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）

28

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

29

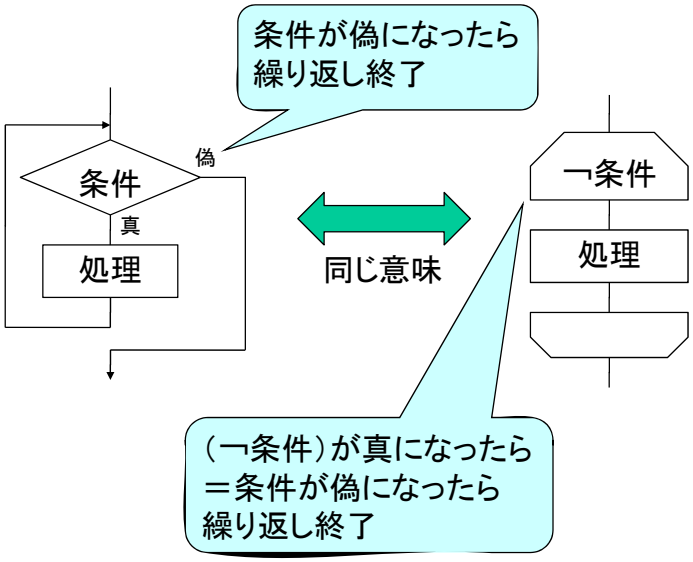
条件判断

- 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

30

反復処理の省略記法



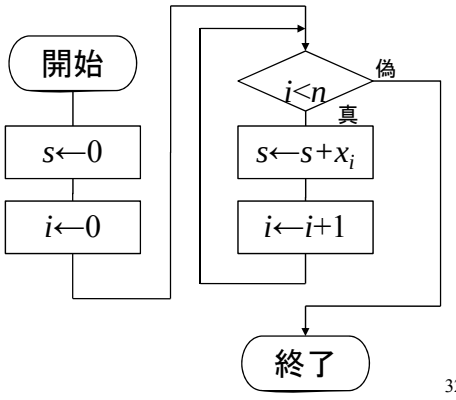
31

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　$s=\sum_{i=0}^{n-1} x_i$



32

正しいアルゴリズム

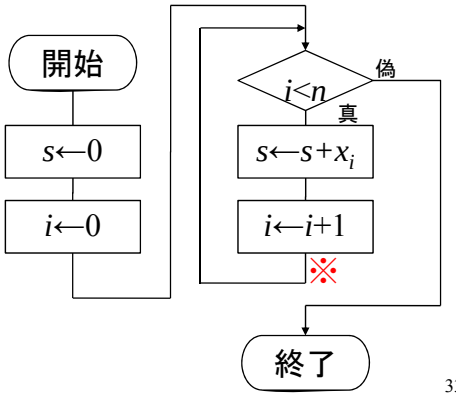
- アルゴリズムが正しいことを数学的に証明することができる

右図※の箇所で

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

が常に真。

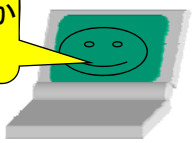
ループ不変条件：
数学的帰納法を使って
証明してみよう



33

プログラミング言語の分類（高級言語と低級言語）

0と1の列しか
わからない



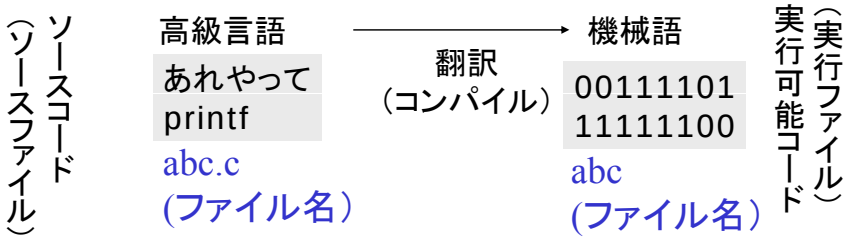
日本語しか
わかんない

コンピュータ側
機械語
アセンブリ言語
低級

Java
C言語
LISP
FORTRAN
人間側
高級

34

コンパイラ

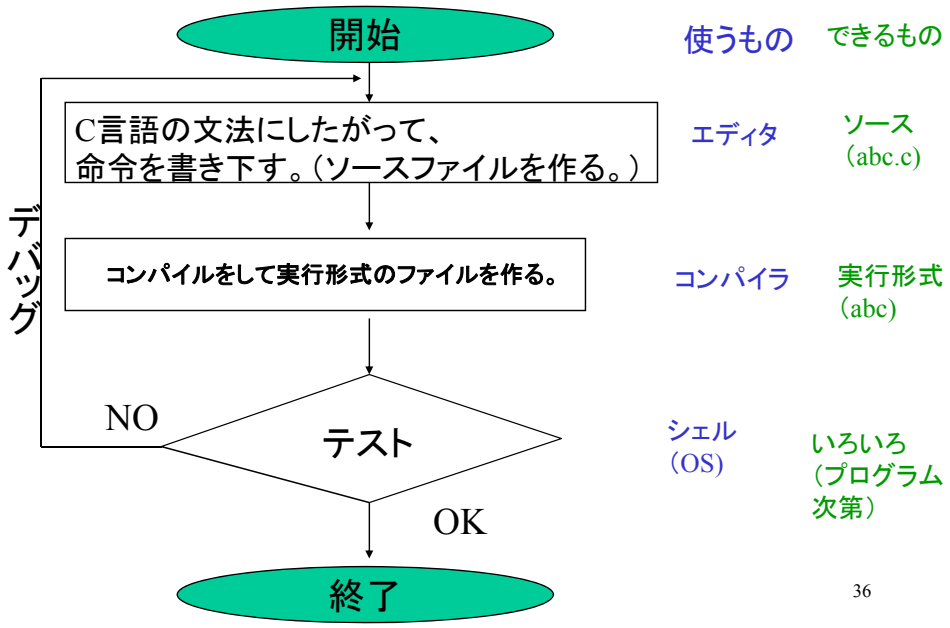


コンパイラ:
高級言語（例えばC言語）から、低級言語（機械後）へ翻訳（コンパイル）するソフトウェア。

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。）

35

C言語でのプログラムの作り方



36

良いソフトウェアの基準

- 速い

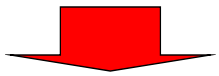
  同じことするなら速い方がいいでしょ。

- 強い

  どんな入力でもきちんと動作してほしいでしょ。

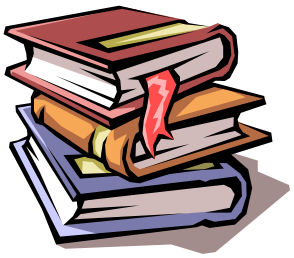
- 分かりやすい

  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)

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
```

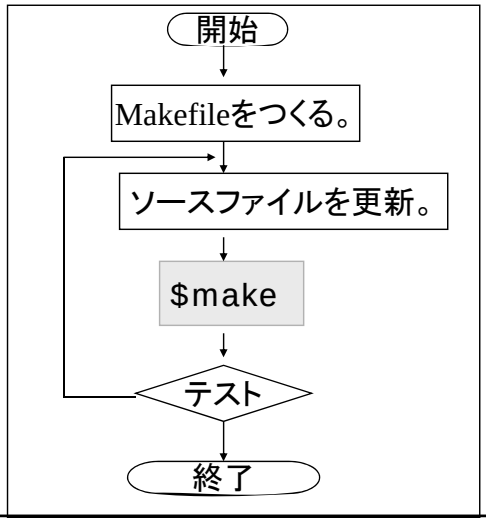
4

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）

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。

```
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 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。

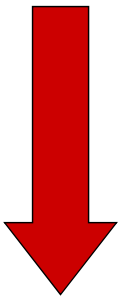
```
int  main()
{
    return  0;
}
```

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）の事だと思うとよい。

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>
```

（教科書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。

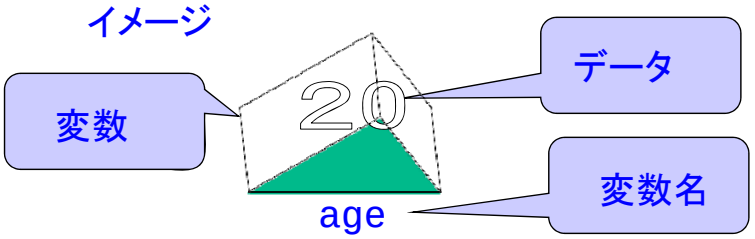
2

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。

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。

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。

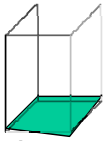
注意：変数名や関数名に予約語を用いてはいけない。
(予約語以外で、命名する。）

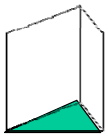
6

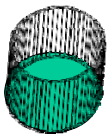
C言語の代表的なデータ型

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

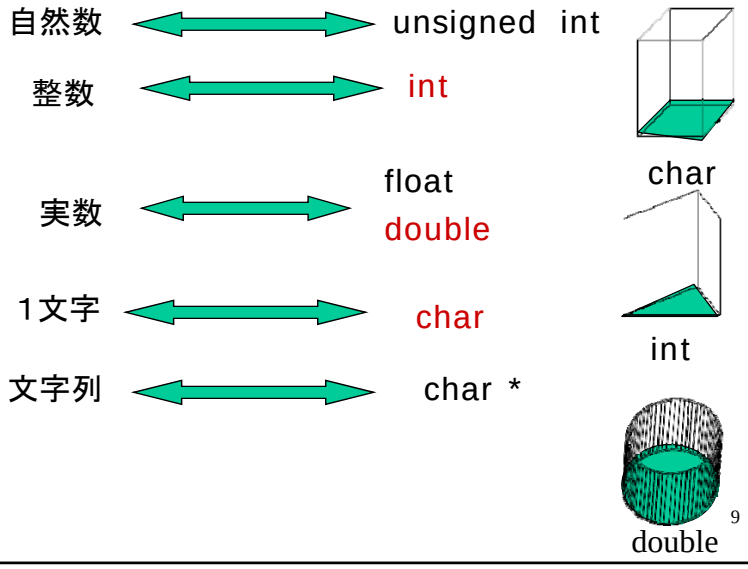
7

C言語のデータ型が扱える範囲

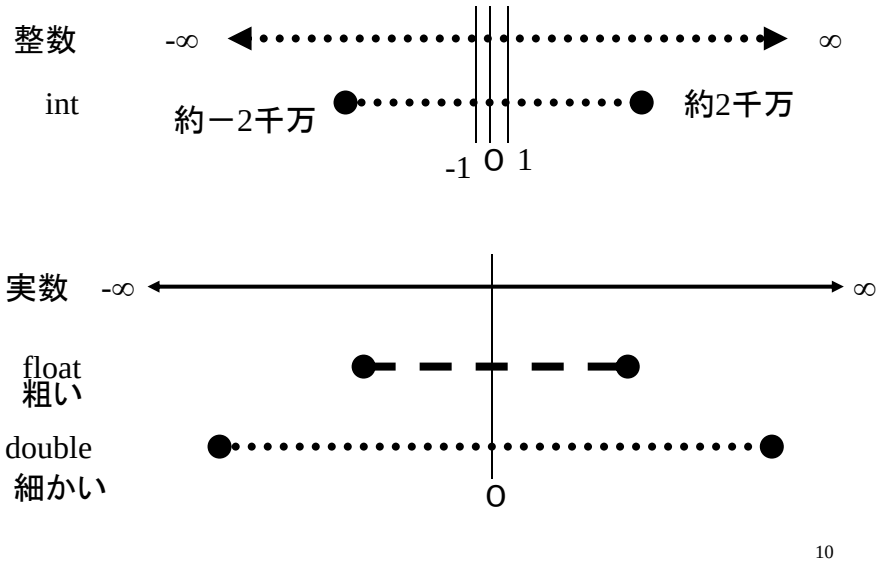
型	バイト長	表現範囲
char	1	0~255
int	4	-214748648~21483647
short int	2	-32768~32767
unsigned int	4	0~4294967295
float	4	$\pm 1.0*10^{-37}$ ~ $\pm 1.0*10^{38}$
double	8	$\pm$ $1.0*10^{-307}$ ~ $\pm 1.0*10^{308}$

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

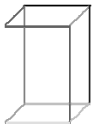
実数(連続量)　double　体重、温度、等

明確に区別すること。

11

文字と文字列

char



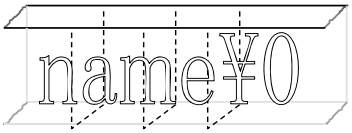
char型には、半角文字1文字だけを保存できる。

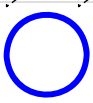
char

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

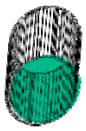
```
int   age;    /*    年齢    */
```

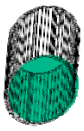
変数宣言は、プログラム内で用いるデータを入れる入れ物(変数)を用意して、その入れ物に名前をつけると考えればよい。

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     */
}
```

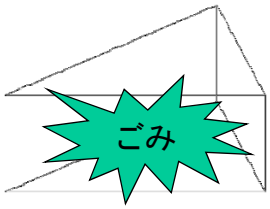
なお、main関数先頭以外での変数宣言については、第9回で説明。

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

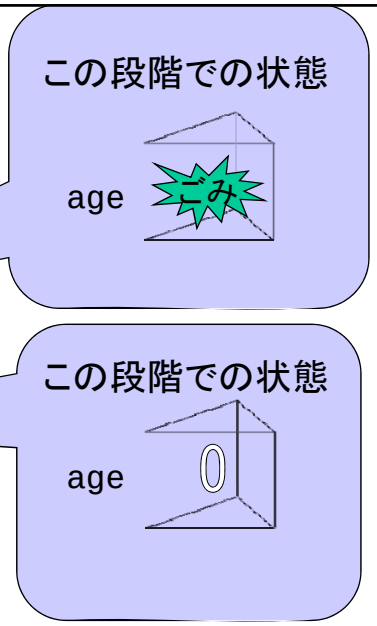


18

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;

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



’＝’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。

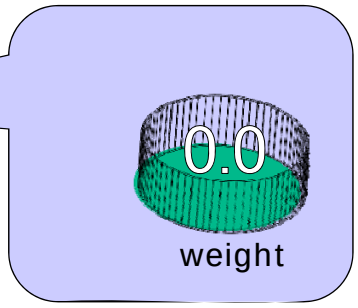
19

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

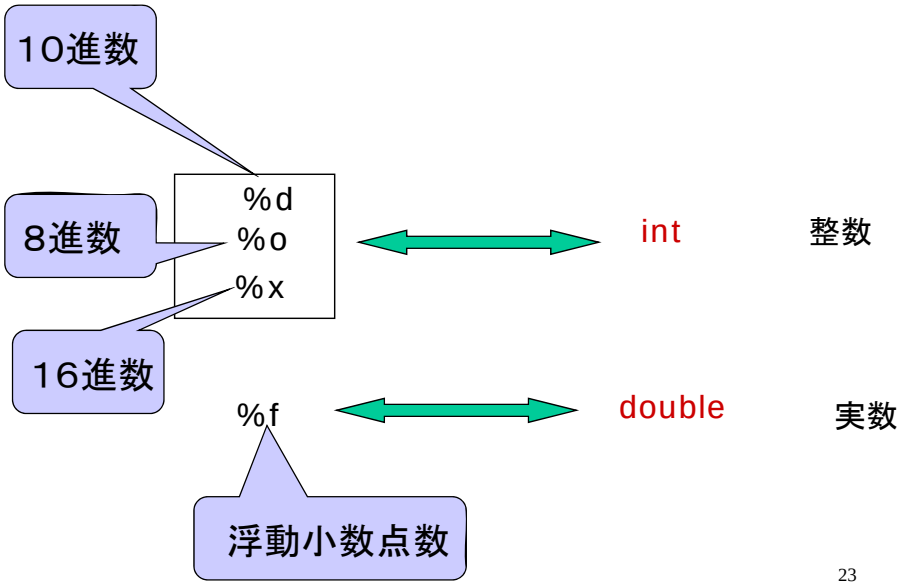
%変換文字　printf文の文字列内の%変換文字(変換仕様)は、後ろの変数に関する出力指示を表わす。

(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文における変換仕様と型の対応



23

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);
```

24

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。

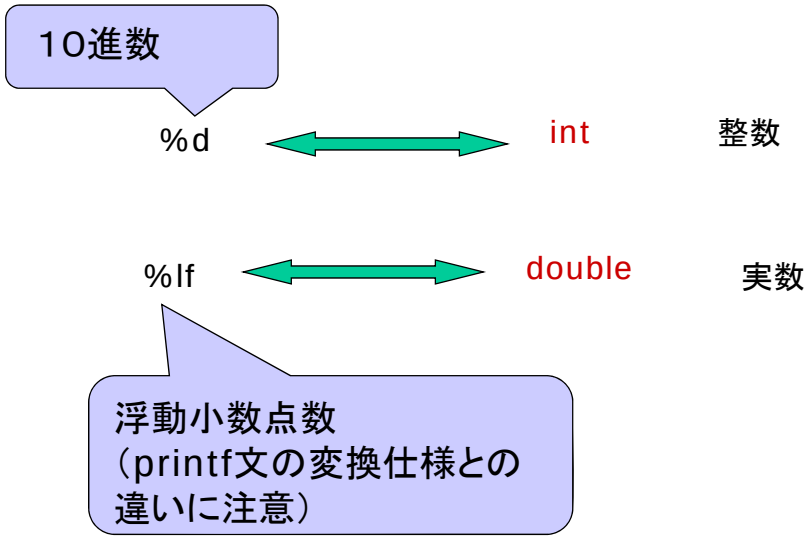
&変数　scanf文の変数名には＆をつけること。
＆は変数のアドレスを表わす。
詳しくは、第１１回ポインタで説明する。

%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



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'の間違い

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
a=2 b=4 c=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
a=1 b=3 c=5
```

36

入出力のあるプログラム例 (教科書p.36参照)

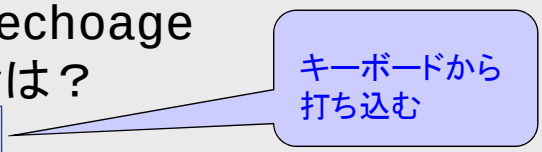
```
/*
        作成日:yyyy/mm/dd
        作成者:本荘　太郎
        学籍番号:B0zB0xx
        ソースファイル:echoage.c
        実行ファイル:echoage
        説明:入力された年齢を表示するプログラム
        入力:標準入力から年齢を入力する。
        出力:標準出力に年齢を出力する。
*/

/*      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;
}
```

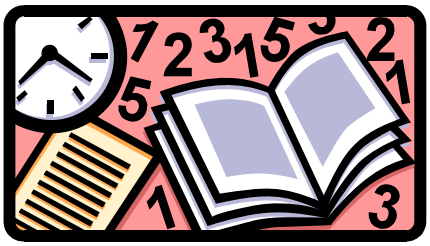
実行結果

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



39

第4回簡単な計算・プリプロセッサ



1

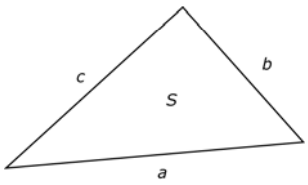
今回の目標

- 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。

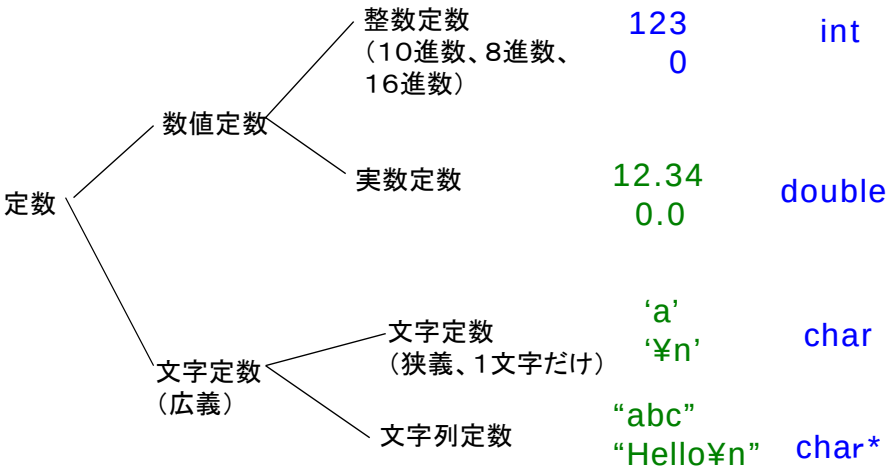
$$d = \frac{a + b + c}{2}$$

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

3

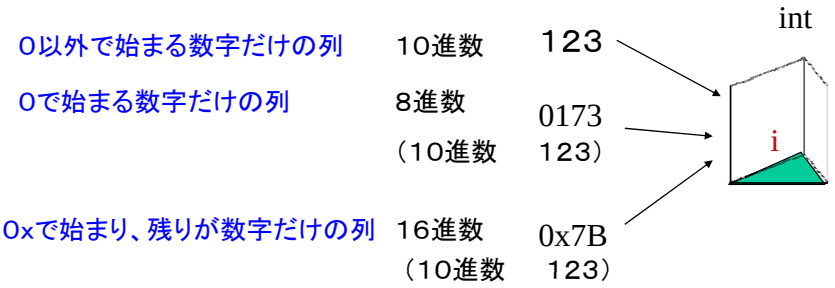
定数の分類と型

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



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  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

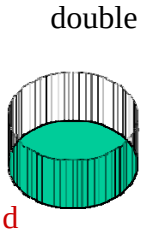
小数点を含む数字列　12.34

e、Eを含む形　1234e-2

（意味　$1234\times10^{-2}=12.34$　）

e、Eと小数点を含む形　0.1234E+2

（意味　$0.1234\times10^{+2}=12.34$　）



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

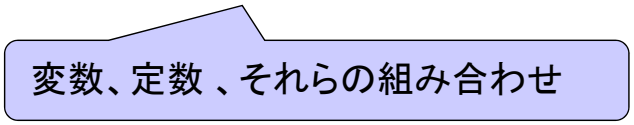
練習2

```
/*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;
}
```

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	$a \times b$
	$a \bullet b$
	ab
	（記号省略可）
		`edge1*edge2`
		`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
	$a \wedge 2$
		`edge1*edge1`
		`pow(edge1,2.0)`
		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。

整数 / 整数 → 整数

```
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。

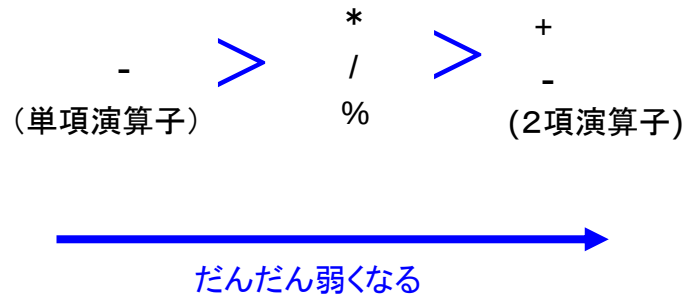
このような例では、a,b,cすべてが整数（int型）でなくてはならない。

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;
```
→ 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

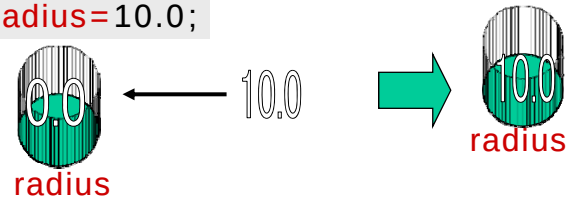
代入演算子

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)

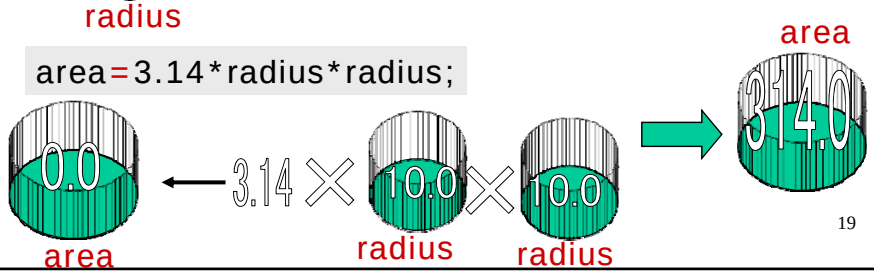
変数＝式

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;
```




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



19

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

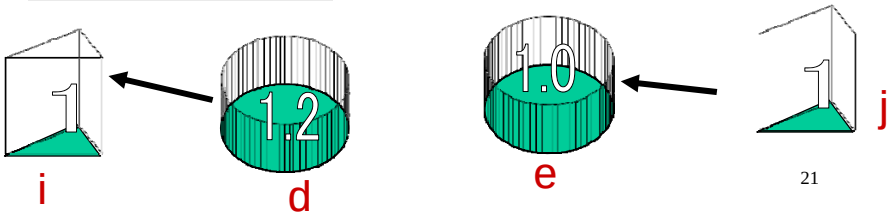
キャスト演算子（型の変換法）

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;
```

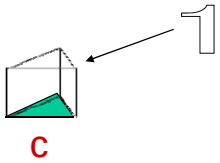
本演習では、
このような自動型変換は用いない事。
（スタイル規則参照）

22

インクリメント演算子とデクリメント演算子

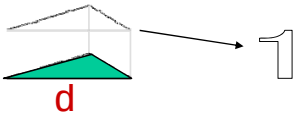
C言語では、1つづつの増減用に、
簡単な形が用意されている。

インクリメント演算子 ++



c++;　　別の書き方。　c=c+1;

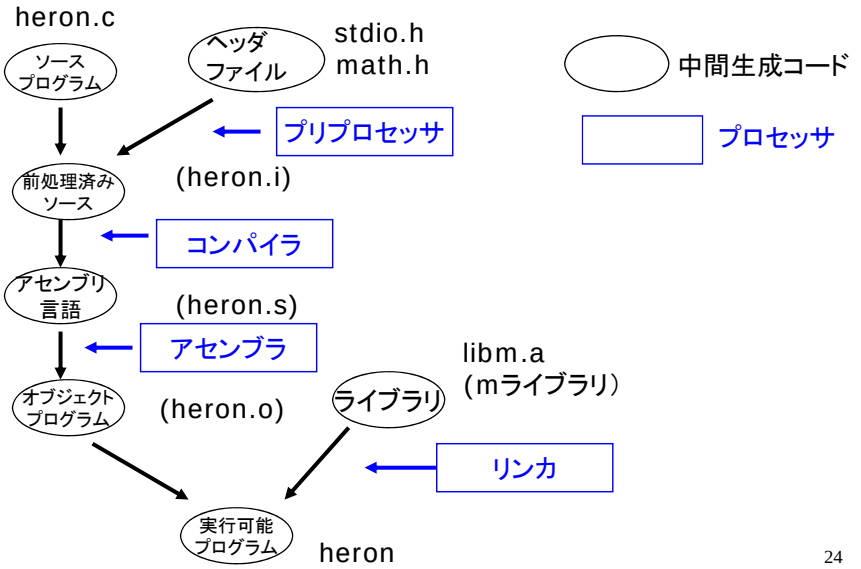
デクリメント演算子 --



d--;　　別の書き方。　d=d-1;

23

gccコマンドの詳細



24

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。

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

#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。

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よい。）

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
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)

32

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
```

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

#difine 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。

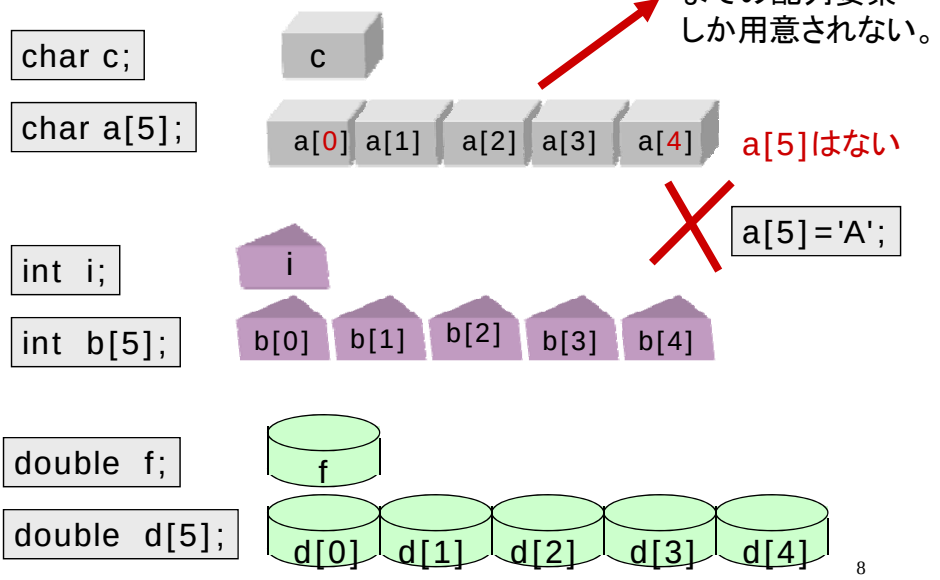
```
int i;
i=3;
min=x[i];
```

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。

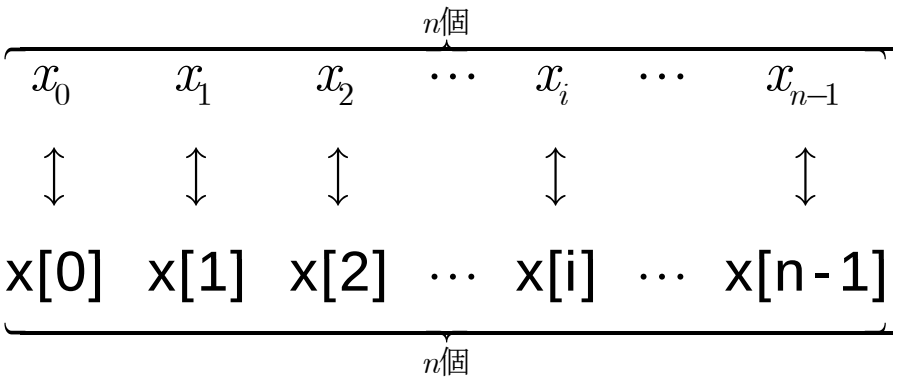
7

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()
{
      int    i;
      double    a[SIZE];
      double    b[SIZE];
      double    c[SIZE];

      /*     次に続く     */
```

```
/* 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*/
/*    次に続く    */
```

```
/*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];
```

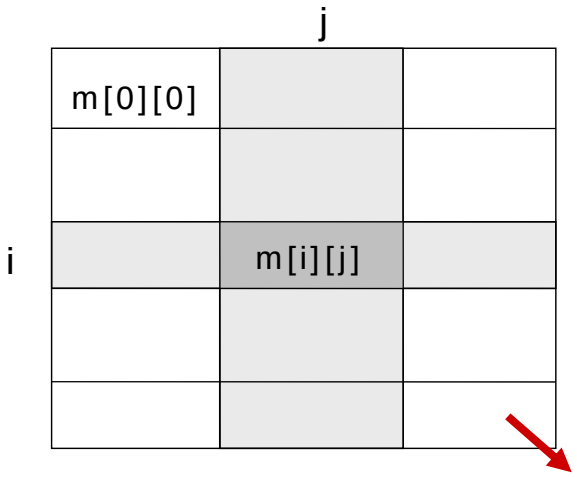
使いかた:

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()
{
      doulbe x[GYO][RETU];    /*行列*/
      double tx[RETU][GYO];   /*転置行列*/

      /*次に続く*/
```

```
      /*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];
      /*次に続く*/
```

```
    /*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];
```

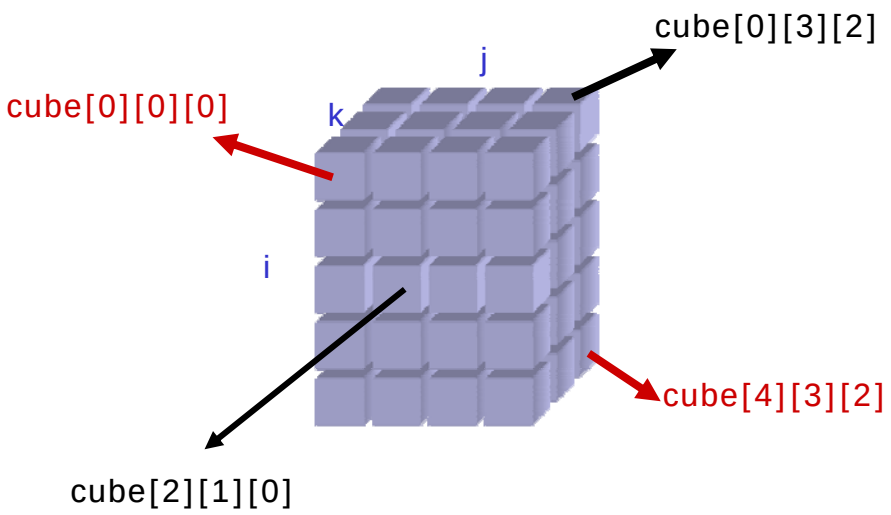
使いかた:

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。
$
```

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(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。

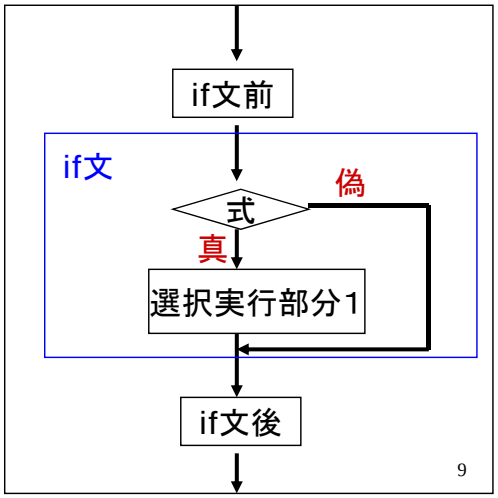
8

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



9

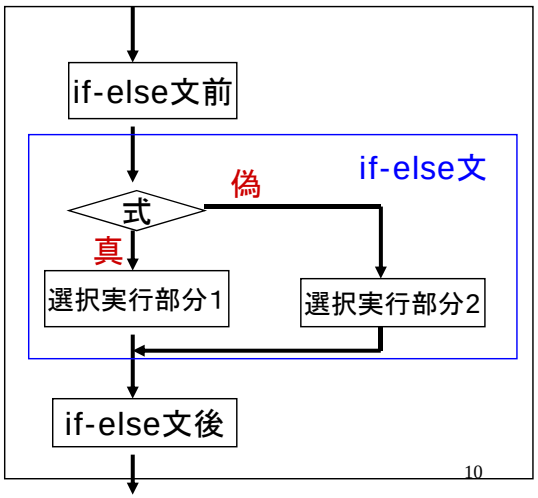
if-else文

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



10

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

間違い　a =< b

a => b

×

正しい　a < = b　a がb以下のとき真

a > = 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以外のときに実行されます。

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

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）

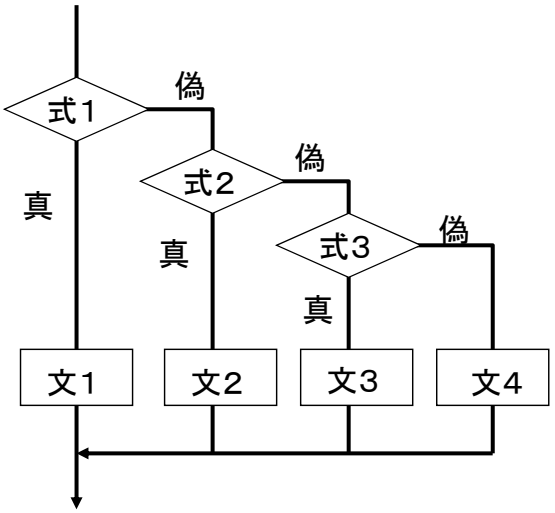
書式

```
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
    }
```

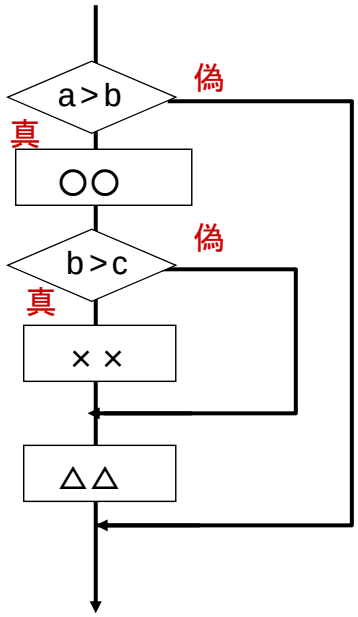
ここに、
またif文を書ける。

```
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*/
        doulbe 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以外にして下さい。¥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

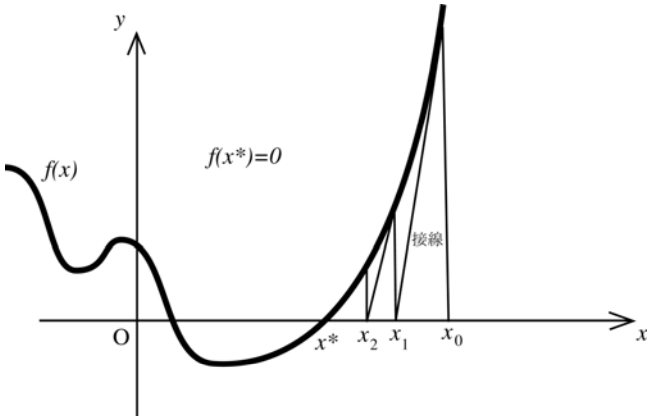
- アルゴリズムの基本となる制御構造（順次、分岐、反復構造）を理解する。
- 繰り返し（反復構造、ループ）を理解する。
- ループからの様々な終了の方法を理解する。
- 繰り返しを用いたアルゴリズムに慣れる。

☆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

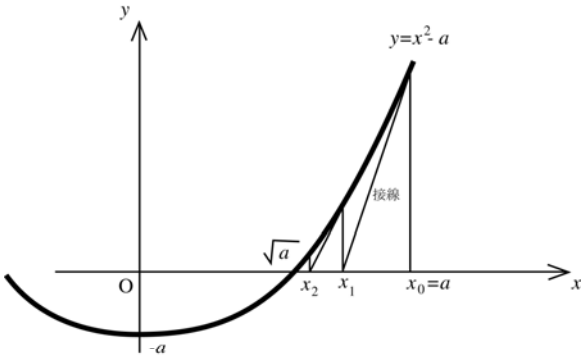


$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} \qquad x_0, x_1, x_2 \ldots, x_\infty \rightarrow x^*$$

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、



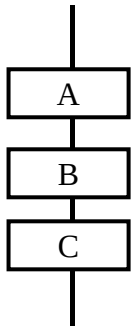
$$x_{i+1} = x_i - \frac{x_i^2 - a}{2x_i} \qquad a = x_0, x_1, x_2 \ldots, x_\infty \rightarrow \sqrt{a}$$

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

4

プログラムの制御構造

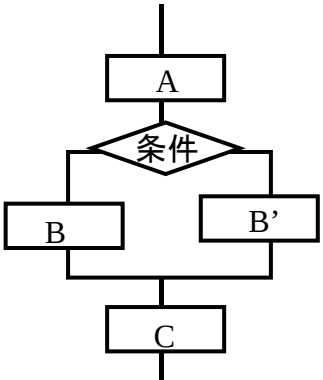
順次構造
（連接構造）



$A \rightarrow B \rightarrow C$

の順序で実行される

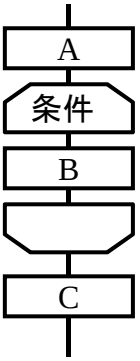
選択構造
（分岐構造）



$A \rightarrow \begin{pmatrix} B \\ B' \end{pmatrix} \rightarrow C$

の順序で実行される

反復構造
（繰り返し構造）



$A \rightarrow \underbrace{B \rightarrow B \rightarrow \cdots B}_{} \rightarrow C$

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。

- 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。

6

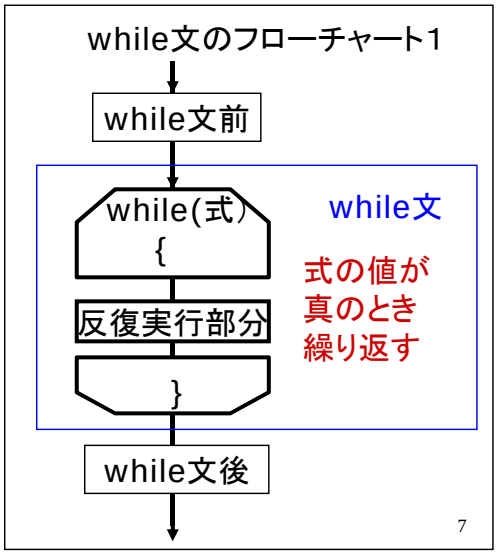
while文

式（論理式）が真である間、命令を繰り返し実行する。

書式

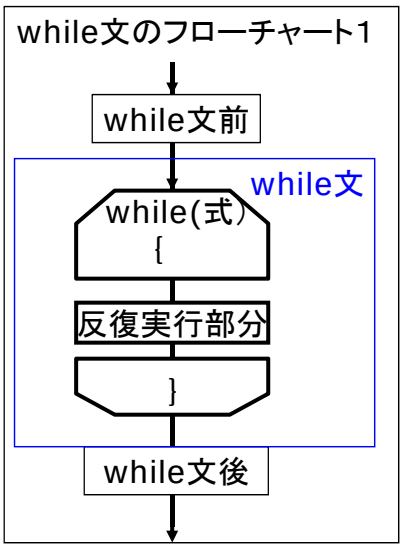
```
while(式)
    {
        反復実行部分
    }
```

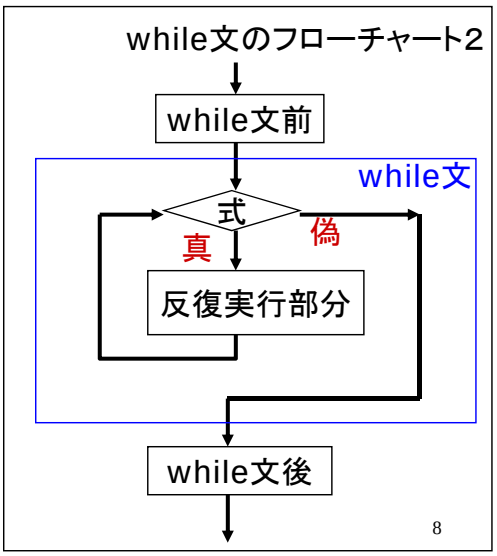
式：反復を続ける条件を表す論理式



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。





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)。

10

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

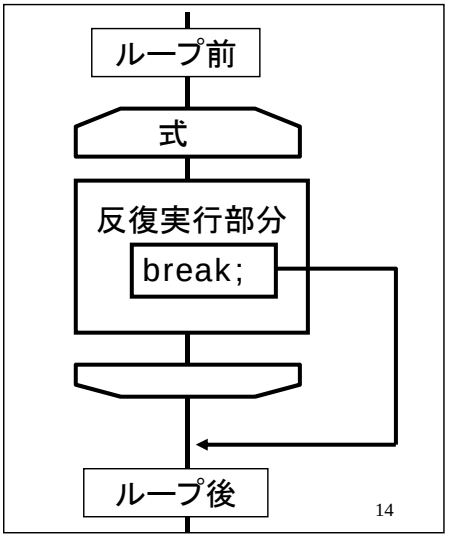
```
/* infty_loop.c　無限ループ実験(コメント省略)　*/
#include<stdio.h>
int main()
{
        printf("無限ループ実験開始 ¥n");
        while(1)
          {
                printf("*¥n");
                printf("**¥n");
                printf("***¥n");
                printf("****¥n");
                printf("*****¥n");
                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;)
   }
```



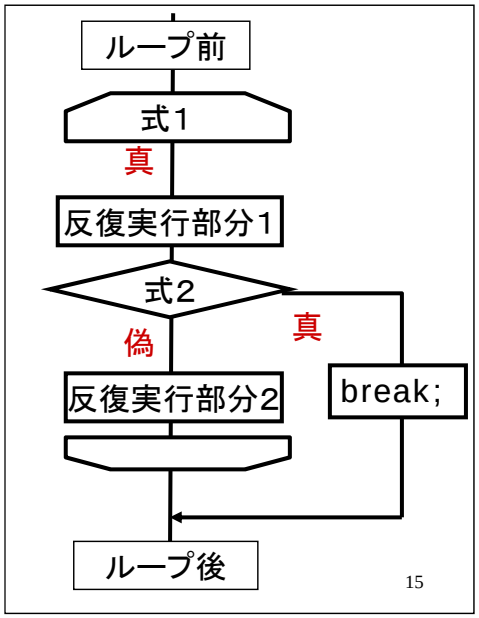
14

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。



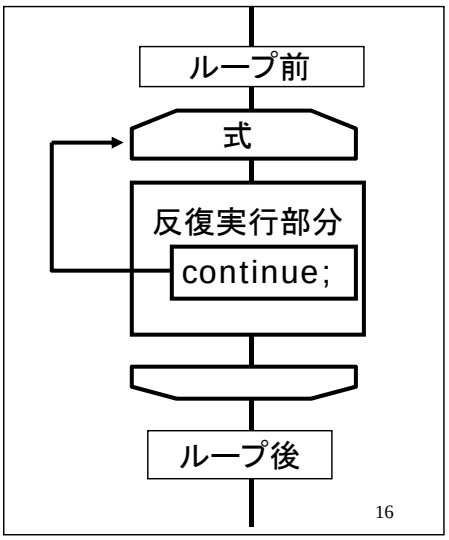
15

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;）
    }
```



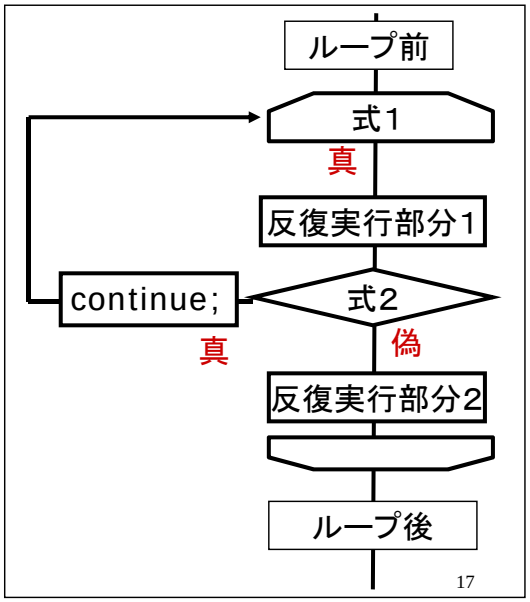
16

continue文の典型的な使い方

典型的な使い方

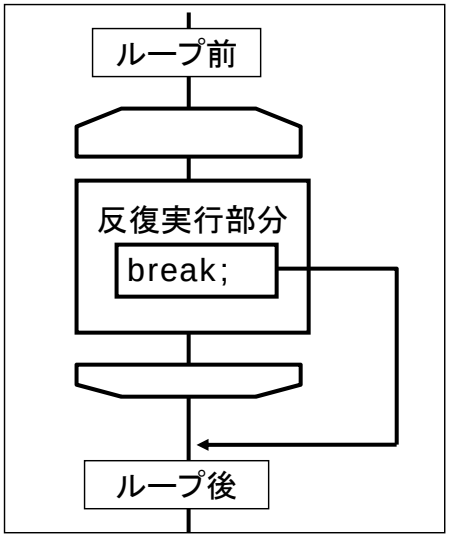
```
while(式1)
    {
        反復実行部分1
        if(式2)
            {
                continue;
            }
        反復実行部分2
    }
```

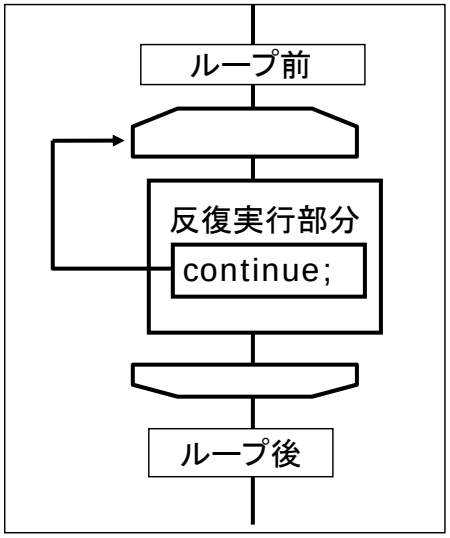
ループ中に反復実行文2を
実行するか選択できる。



17

continue文とbreak文

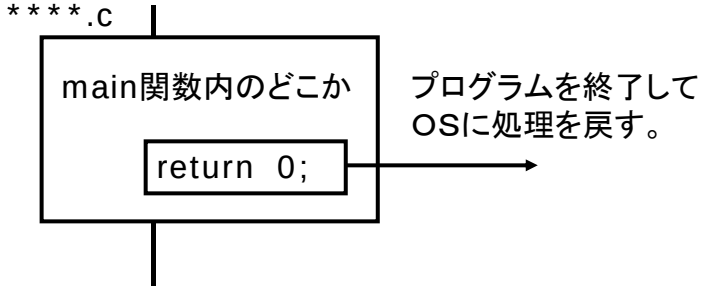




18

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:BOzB0xx
        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。
        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      /*    繰り返しの上限値*/

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より大きい間繰り返す*/
            {
                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;
}
```

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
x4=       1.41421356
  2.00の平方根は          1.41421356です。
$
```

25

実行例2

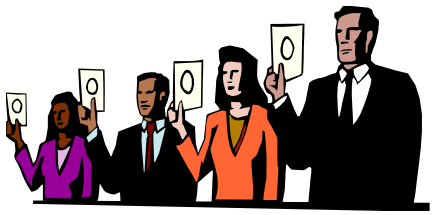
```
$./mysqrt
平方根を求めます。
正の実数を入力して下さい。
0.0
   0.00の平方根は    0.00です。
$
```

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 $x_0, x_1, \cdots, x_{n-1}$

の平均値

$$\overline{x} = \frac{x_0 + x_1 + \cdots + x_{n-1}}{n}$$

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

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。

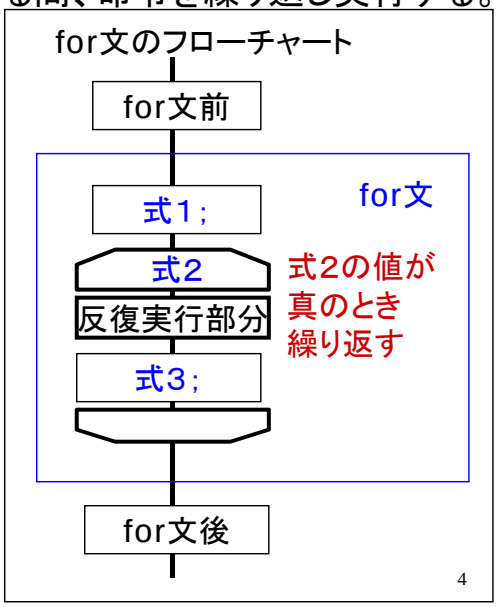
書式

```
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



4

for文

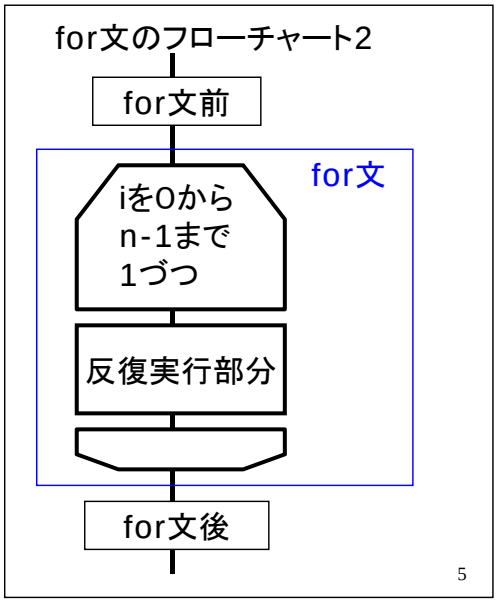
典型的な使い方

```
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

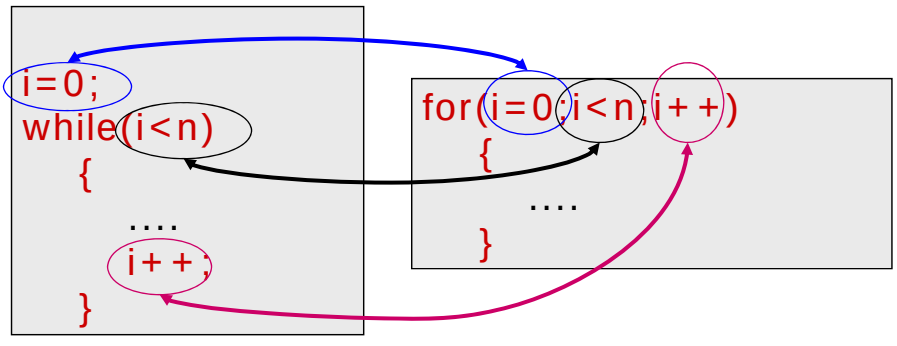
i<n：反復を続ける条件

i++：ループカウンタの
インクリメント



5

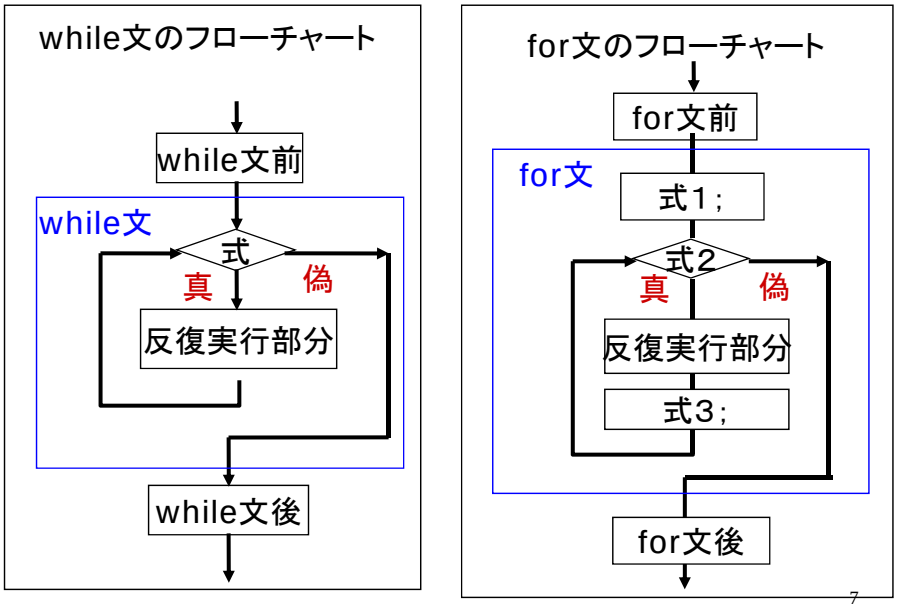
while 文との比較



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(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

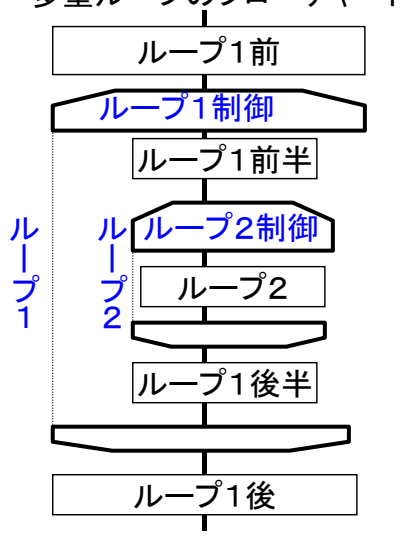
多重ループ

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

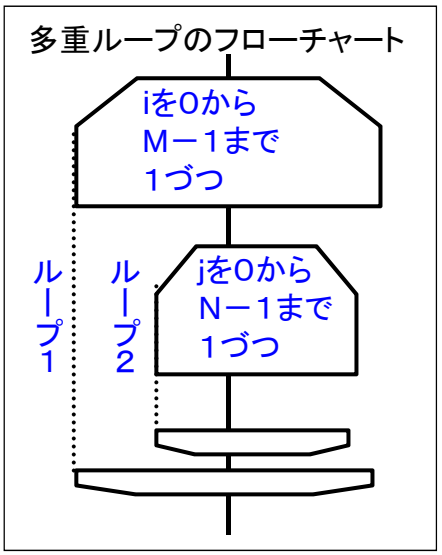


10

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
          }
    }
```



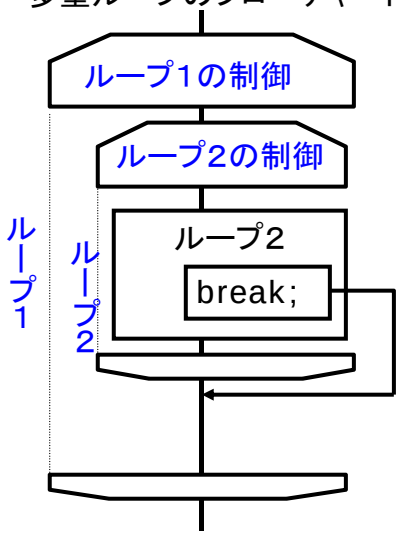
多重ループでは、ループ事に
別のループカウンタを用いる。

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。

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

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*/
```

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]);
   }

/*     次に続く       */
```

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。
$
```

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1
(処理の分解)

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。
- 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

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

returnの後の式(変数や定数)と同じ型

一種の入力、仮引数という。
仮引数は変数の一種。
複数の場合もある。

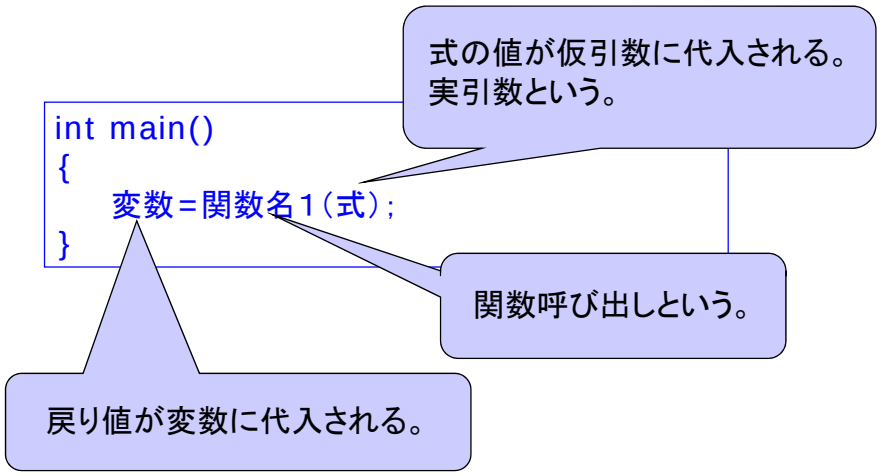
書式:

```
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;
}
```

一種の出力、戻り値という。
もちろん、
定数や変数であっても良い。

4

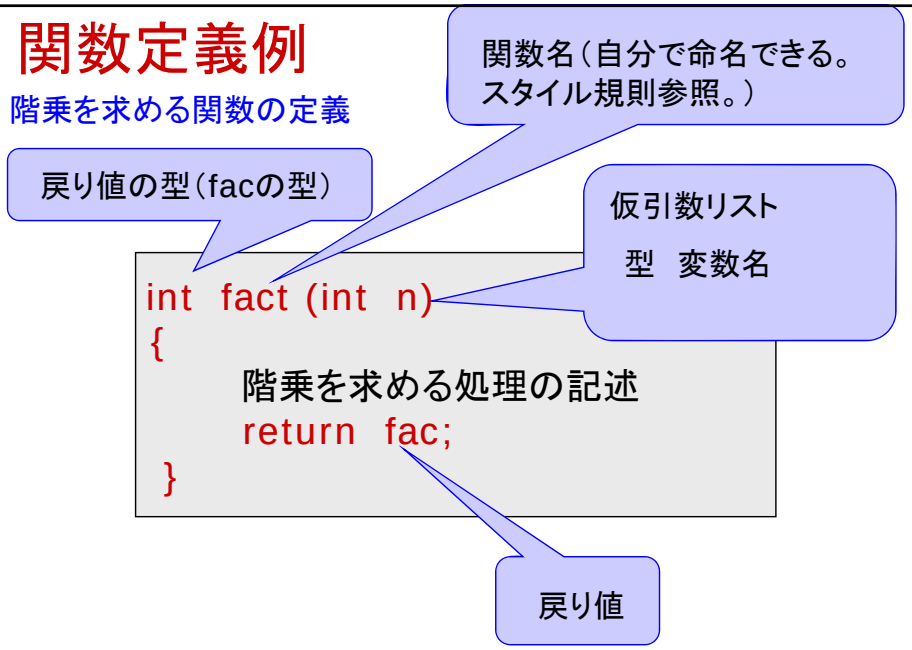
関数の典型的な利用例（関数の呼び出し）



5

関数定義例

階階乗を求める関数の定義



注意：メイン関数以外はプロトタイプ宣言を行うこと。

6

関数利用例

main内での関数呼び出し利用

```
int  main()
{
      int f;
       int n;
       n=5;
       f=fact(n);
       return 0;
```

呼び出す際に、式(変数や定数)を指定する。
この式の値が仮引数に代入される。(仮引数と異なる変数名でも良い。)

戻り値の代入される変数。型は戻り値の型と同じ

7

mainという関数

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。

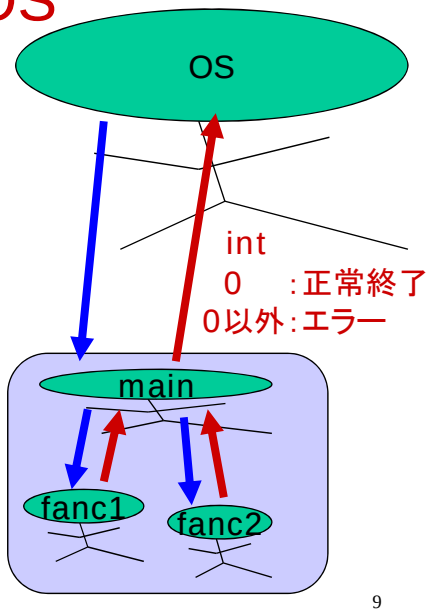
mainは特別な関数名で、一つのプログラムに必ず1つだけなければいけない。プログラムの実行はmain関数の最初から行われる。

8

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。



9

処理の分割と関数の利用1

```
int main()
{  /*関数を用いないで組み合わせ数を計算(詳細省略) */
   for(i=1;i<n;i++){
       bunshi=bunshi*i;
   }
   for(i=1;i<m;i++){
       bunbo1=bunbo1*i;
   }
    for(i=1;i<n-m;i++){
        bunbo2=bunbo2*i;
    }
    com=bunshi/(bunbo1*bunbo2);
   return 0;
}
```

似ている。
プログラムの構造が同一。
(ひとまとまりにできない
か?)

10

処理の分割と関数の利用2

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

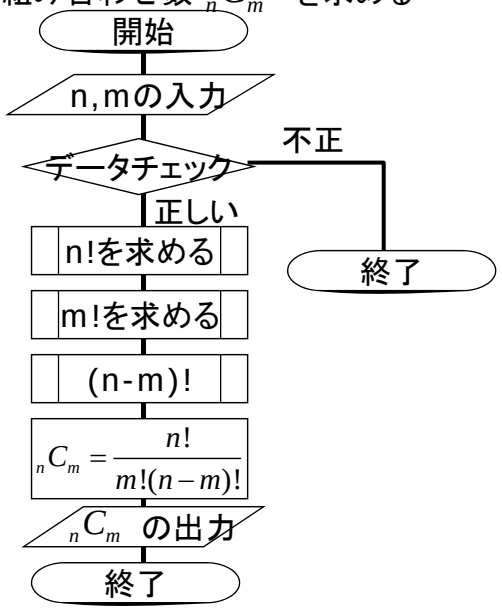
階乗を求める専門家（関数）があると、便利。

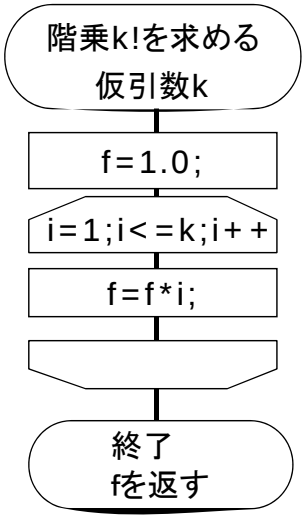
```
int fact(int k)
{
   int fac;
   for(i=1;i<k;i++){
       fac=fac*i;
   }
   return  fac;
}
```

依頼データを元に、結果を返す。（階乗の計算を行なう。）

関数を表わすフローチャート

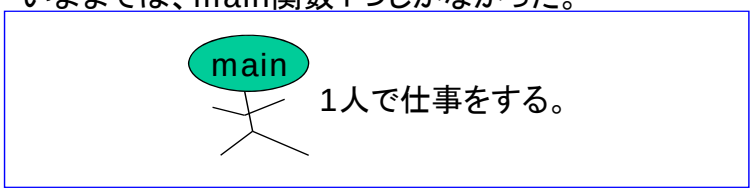
組み合わせ数 ${}_nC_m$ を求める



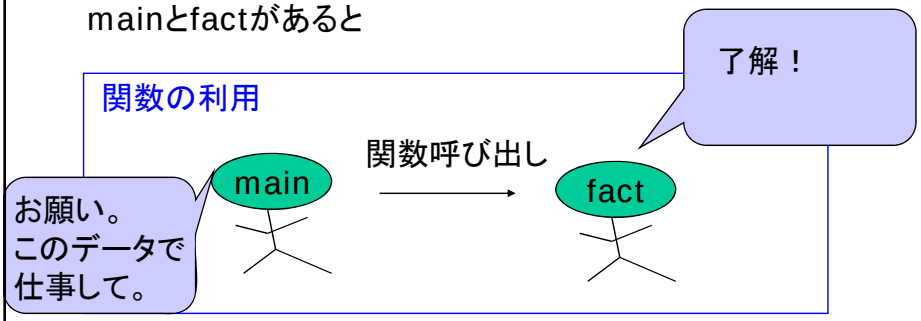


12

いままでは、main関数1つしかなかった。



mainとfact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。

13

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、return文直後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;
}
```

14

ソースにおける関数定義位置とプロトタイプ宣言

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。

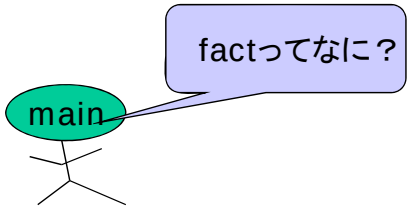
注意:main関数以外は、プロトタイプ宣言をmain関数前に記述する。
関数定義はmain関数後に記述する。

15

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。

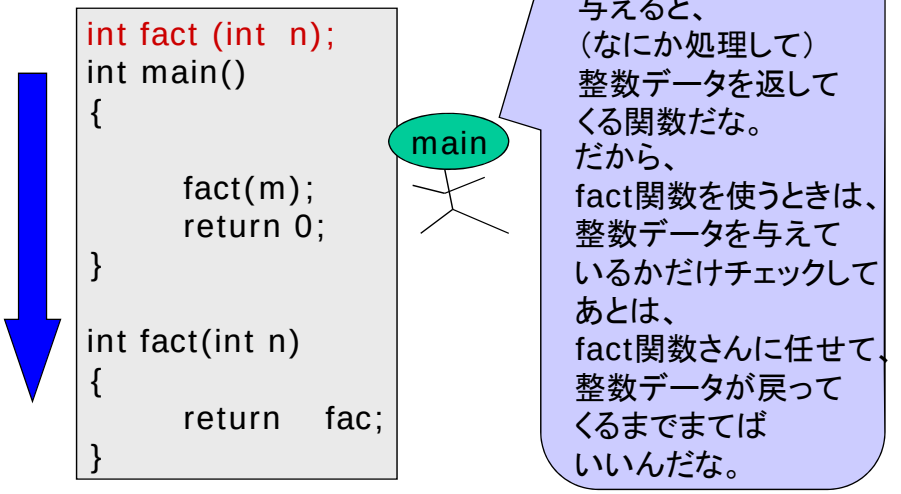
```
/**/
int main()
{

        f=fact(m);
        return    0;
}

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



16

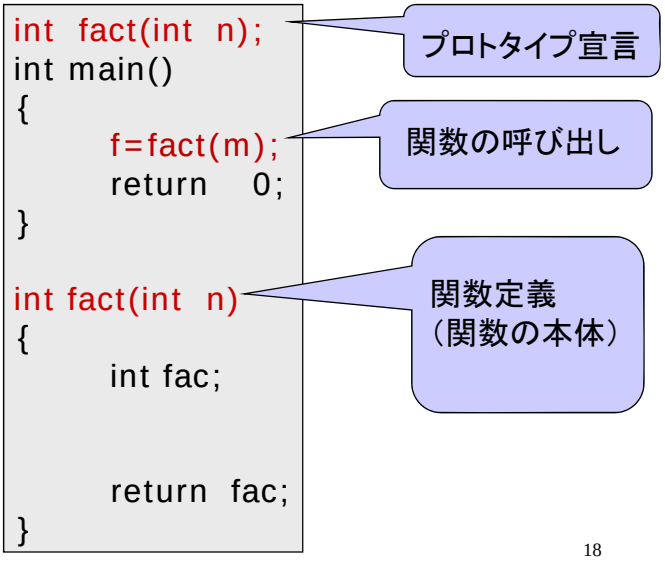
プロトタイプ宣言の役割2

プロトタイプ宣言があると。



複数の関数があるソース例

書式だけ抽出



18

実引数と仮引数

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

int fact(int  n)
{
        int fac;


        return  fac;
}
```

戻り値格納用の変数

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:実引数が変数の場合でも、実引数と仮引数の名前は異なっていてもかまわない。

19

関数へ値の渡し方

呼び出す方では、

変数＝関数名(式);　や　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。

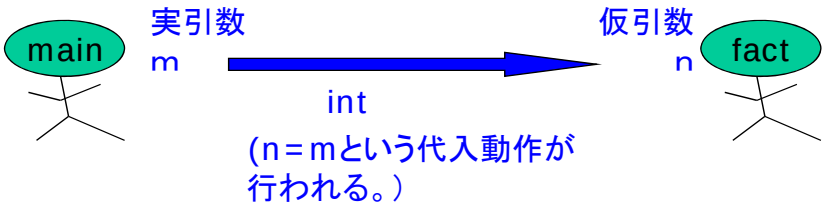
```
int  main()
{
        f=fact(m);
}
int     fact(int n)
{
        int fac;
        return    fac;
}
```

mainのmの値が、factのnに代入される。

20

```
int  main()
{
        f=fact(m);

        return   0;
}
```

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



(n＝mという代入動作が行われる。)

21

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。

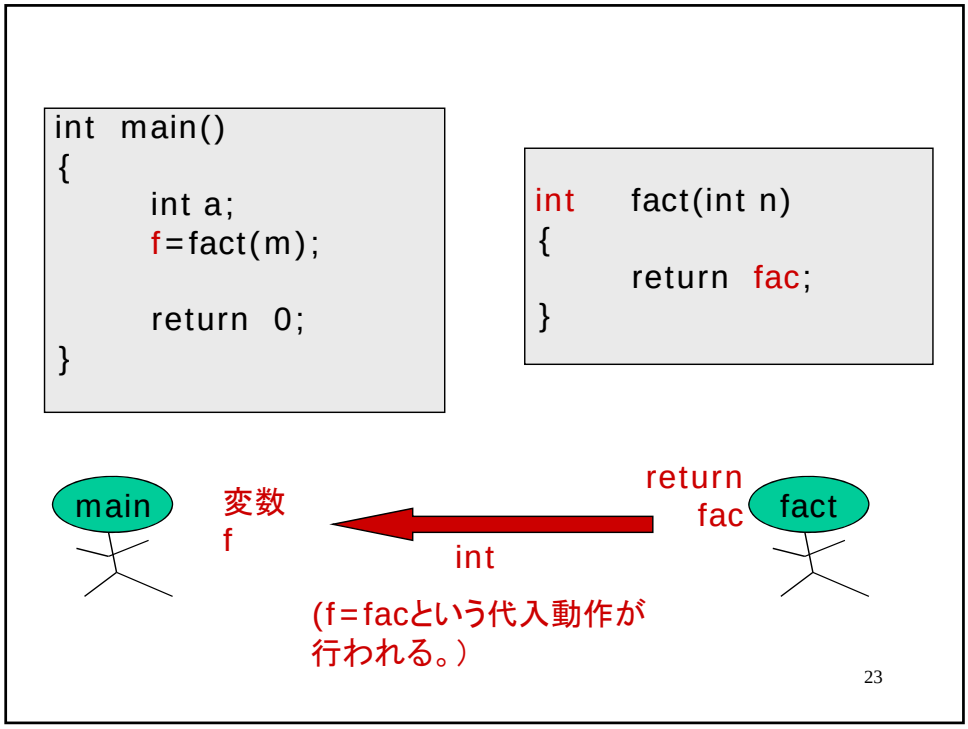
変数＝関数名(式); とすると、戻り値が変数に代入される。

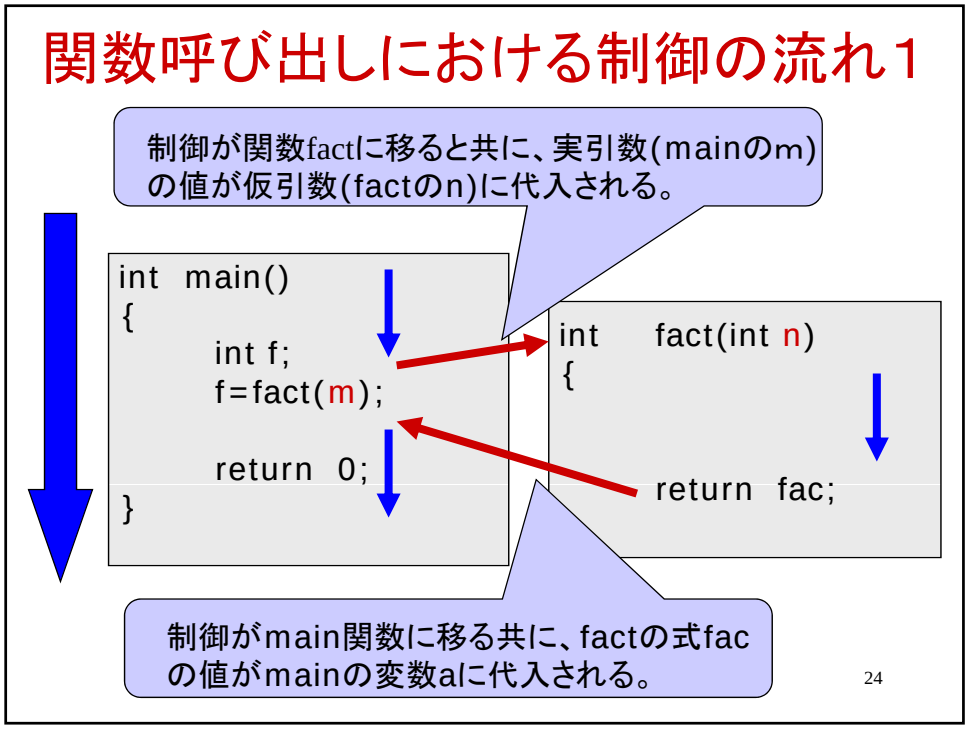
```
int  main()
{
        int f;
        f=fact(m);
}

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。

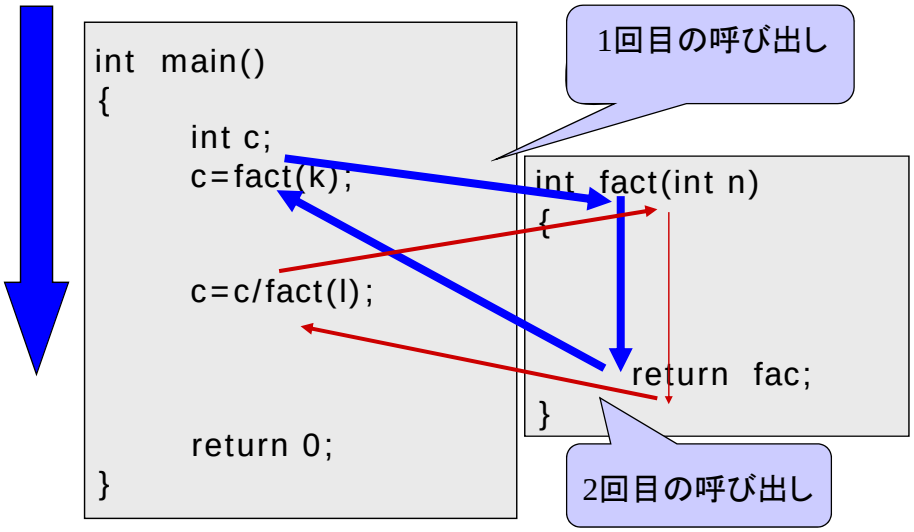
factのfacの値が、
mainのfに代入される。

22


int main()
{
int a;
f=fact(m);
return 0;
}
int fact(int n)
{
return fac;
}
main
変数
f
int
return
fac
fact
(f=facという代入動作が
行われる。)
23


関数呼び出しにおける制御の流れ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。
int main()
{
int f;
f=fact(m);
return 0;
}
int fact(int n)
{
return fac;
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。
24

関数呼び出しにおける制御の流れ2



同じような処理を複数回行いたいとき、
関数を用いると便利。

25

練習1

```
/*test_function.c　関数実験1　コメント省略*/
#include <stdio.h>
int switch_sign(int c);
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*/
```

26

```
/*整数の符合を反転させる関数switch_sign
        仮引数c:被演算項(整数)
        戻り値:"- 被演算項"の値を返す。*/
int  switch_sign(int    c)
{

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

こんな風にコメントを付けること。(スタイル規則参照。)

27

一般的な関数定義
(複数の引数を持つ複数の関数定義)

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,・・・)
{
}
```

プロトタイプ宣言では、セミコロンを忘れずに。

28

練習2

```
/*test_function2.c　関数実験2　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*/
int  add(int x,int y);              /*  和を求める関数 */
int  sub(int x,int y);             /*  差を求める関数 */
int  main()
{
        int a;
        int b;
        int wa;
         int sa;

/*次に続く */
```

29

```
/*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);


        /*和を求める関数関数呼び出し。*/
        wa=add(a,b);
         printf("%d  = add( %d,%d) ¥n",wa,a,b);

        /*差を求める関数関数呼び出し。*/
        sa=sub(a,b);
         printf("%d  = sub( %d,%d) ¥n",sa,a,b);

        return   0;
}
```

30

```
/*続き*/
/*2つの整数の差を計算する関数add
        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     add(int    x, int  y)
{
        /*変数宣言*/
        int   z;

        /*計算*/
        z=x+y;
        return  z;
}
/*add終了*/
/*続く*/
```

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。)

31

```
/*続き*/
/*2つの整数の差を計算する関数sub
        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     add(int    x, int  y)
{
        /*変数宣言*/
        int   z;

        /*計算*/
        z=x+y;
        return  z;
}
/*sub終了*/
/*全プログラム(test_function2.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。)

32

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:combi.c
        実行ファイル:combi
        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      */
```

34

```
    /*    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*/

/*    次に続く      */
```

36

```
        /*      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*/

        /*      次に続く        */
```

37

```
        /*      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!を戻す*/
}
/*関数factの定義終*/
/*全てのプログラム(combi.c)の終了*/
```

38

実行例

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

```
$./combi
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。
$
```

39

第10回関数2
(関数の利用と変数のスコープ)

1

今回の目標

- voidという型を理解する。
- 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。
- 多段にわたる関数呼び出しを理解する。

☆階乗を求める関数を利用して、組み合わせの数を求める関数を作成する

2

順列の数と組み合わせの数

$$ {}_nP_m = \frac{n!}{(n-m)!} = \underbrace{n \times (n-1) \times \cdots \times (n-m+1)}_{m\text{個}} $$

$$ {}_nC_m = \frac{n!}{m! \times (n-m)!} = \frac{\overbrace{n \times (n-1) \times \cdots \times (n-m+1)}^{m\text{個}}}{\underbrace{m \times (m-1) \times \cdots \times 1}_{m\text{個}}} $$

3

引数が無い関数

仮引数(関数への入力)や、
戻り値(関数からの出力)が無いことを、
voidという型であらわす。

仮引数の無い関数例

括弧だけを記述する。

```
int function1()
{
        return  0;
}
```

明示的にvoidと記述する。

```
doulbe  function2(void)
{
        return  1.0;
}
```

4

戻り値の無い関数

戻り値の無い関数例

明示的にvoidと記述する。

```
void function3(int a)
{
        return;
}
```

return の後に式を書かない。

5

関数の副作用

仮引数や戻り値以外の動作を副作用という。
標準入出力への値のやり取り等が代表的な副作用である。
仮引数や戻り値が無い関数でも、副作用によって処理を行える。
(数学的な関数と大幅に異なる機能である。)

副作用例:

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

標準入力から値を
読み込む副作用

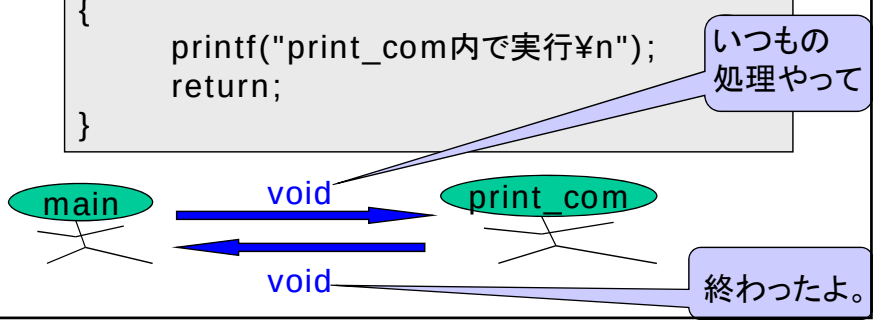
```
void kaigyo(void )
{
        printf("¥n");
        return;
}
```

標準出力へ値を
出力する副作用

6

練習1

```
/*test_void.c  練習1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;
}
```



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。

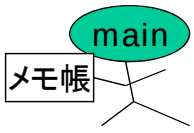
8

```
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
}
```

9

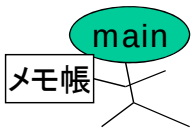
イメージ

いままでは、main関数1つしかなかった。



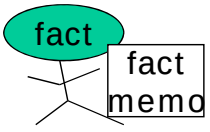
1人で仕事をする。
自分用の変数をつかう。

mainとfactがあると



お願い

このデータ
で仕事して

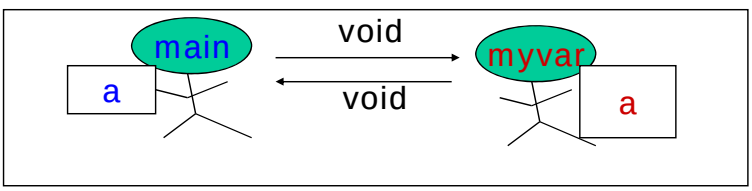


10

練習2

```
/*test_scope.c  練習2  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;
}
/*  次に続く  */
```

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

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

        return;
}
```



12

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;
```

13

グローバル変数のスコープ

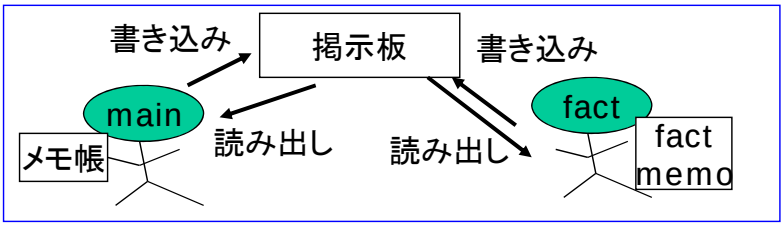
```
/*グローバル変数宣言*/
グローバル変数

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
}
```

14

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版



15

グローバル変数とローカル変数が同じ名前のときは？

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

16

多段にわたる関数呼び出し

main関数以外の関数からでも、関数を呼び出せる。

```
int main()
{
  pe=p(m,n);
  return 0;
}
```

```
int p(int n,int m)
{
  pe=f(n)/f(n-m);
  return pe;
}
```

```
int f(int n)
{
  int fa=1.0;
  for(i=1;i<n;i++)
    {
      fa=fa*i;
    }
  return fa;
}
```

main関数以外からでも、定義した関数を呼び出せる。

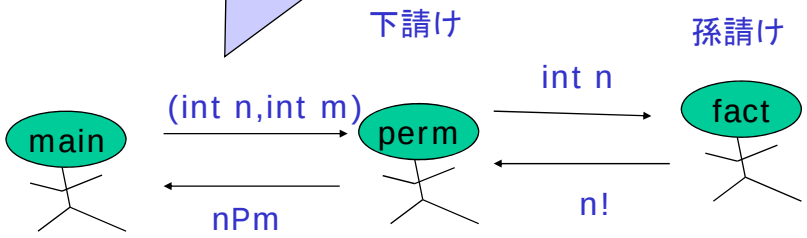
main関数でのreturn文でプログラム全体が終了する。

17

イメージ

仕事を下請け、孫請けに託す。

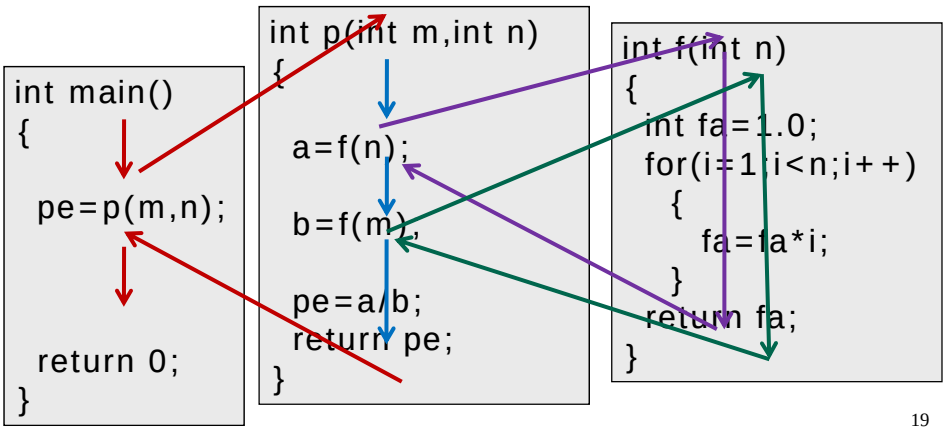
順列の数を求めて。



18

関数呼び出しと処理の流れ処理の流れ

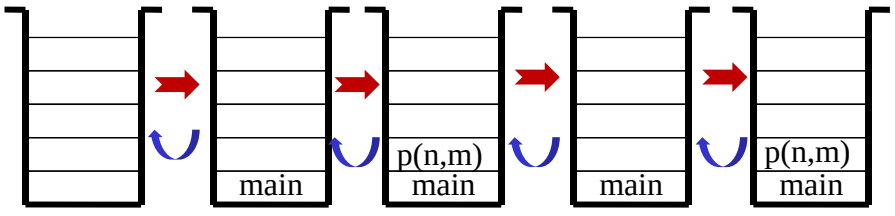
多段に呼び出された場合には、後で呼び出されたものから順に実行される。



19

関数呼び出しとスタック

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



:関数呼び出し

:関数からのreturn

20

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:combination.c
        実行ファイル:combination
        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 n); /*階乗を計算する関数。*/
int     comb(int n,int m);/*組み合わせの数nCmを計算する*/
```

```
/*      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      */
```

22

```
    /*     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=comb(n,m);

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

    return 0;
}
/*main関数終了*/

/*     次に続く       */
```

24

```
        /*      続き    */
/* 組み合わせの数を求める関数
        仮引数n: nCmのn(0以上15未満の値とする。)
        仮引数m:nCmのm (0以上15未満の値とする。)
        戻り値:組み合わせ数nCmを返す。*/
int  comb(int n,int m)
{
        /*      ローカル変数宣言    */
        int     com; /*組み合わせの数*/
        /*計算処理*/
        com=fact(n)/( fact(m)*fact(n-m) );

        return com;
}
/*関数combの定義終*/
/*次に続く*/
```

25

```
        /*      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*/

        /*      次に続く        */
```

26

```
        /*     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!を戻す*/
}
/*関数factの定義終*/
/*全てのプログラム(combination.c)の終了*/
```

27

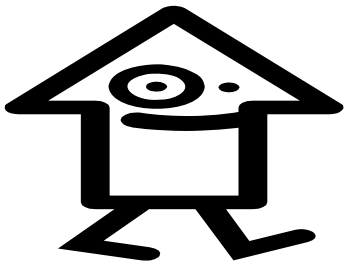
実行例

```
$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。
$
```

28

第１１回ポインタの基礎
（アドレスとポインタ）



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、参照演算子の効果を理解する。
- NULLというアドレスを理解する。

☆複数の関数内で変数のアドレスを表示するプログラムを作成する。

2

ポインタ

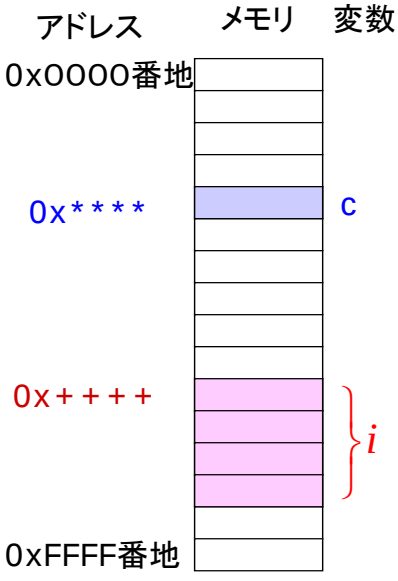
ポインタとは、
変数のアドレスを入れる変数である。

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

3

変数とアドレス

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、メモリの一区画につけ
られた名前である。

4

アドレス演算子　&

&：変数に割り当てられたメモリの先頭アドレスを求める演算子。
前置の単項演算子。

書式

```
&　変数名
```

例

```
int　age;
scanf("%d",&age);
```

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

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

アドレスを指定するscanf文の仕様

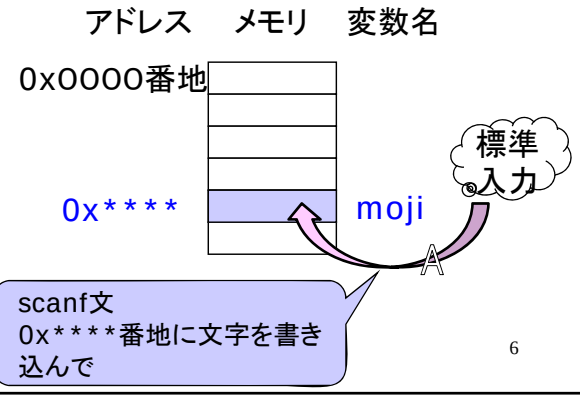
書式

```
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

アドレスを表示するprintf文の仕様

printf文には、アドレスを表示するための変換仕様がある。

例　%p ⟷ アドレス (型の区別無し)

変数のアドレスを表示させる書式

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

&がついているのでアドレス

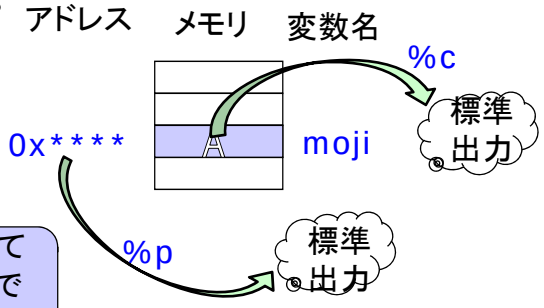
16進数で表示される。

変数の中身(値)を表示させる書式

```
char　moji;
printf("%c",moji);
```

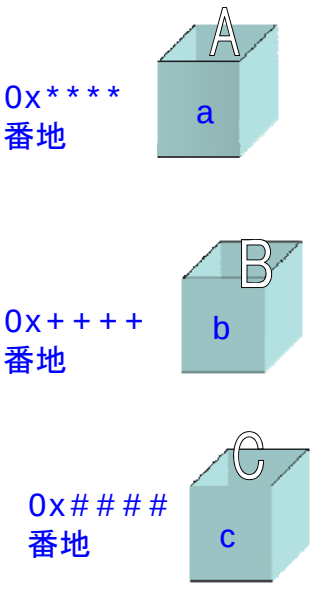
&がついていないので中身(値)

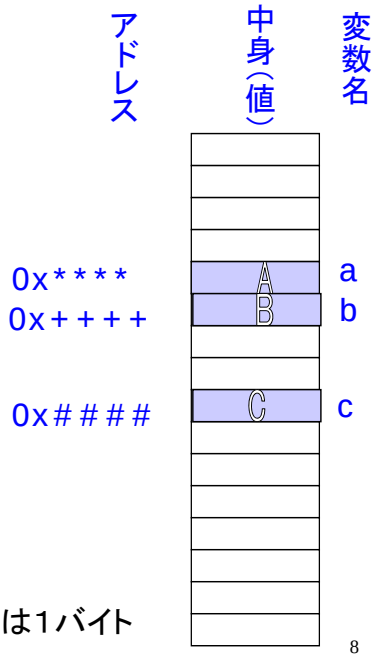
文字として表示される。



7

イメージ





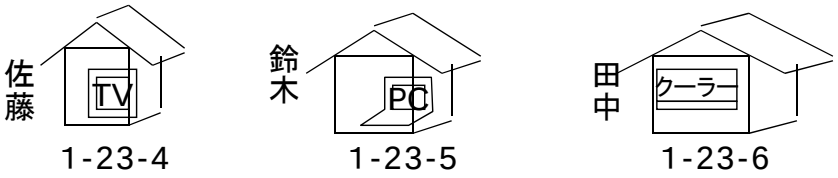
char型は1バイト

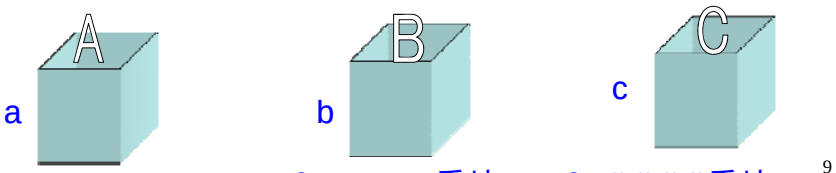
8

イメージ

入れ物(建物)における
名前と
番地(住所)と
中に入っている物

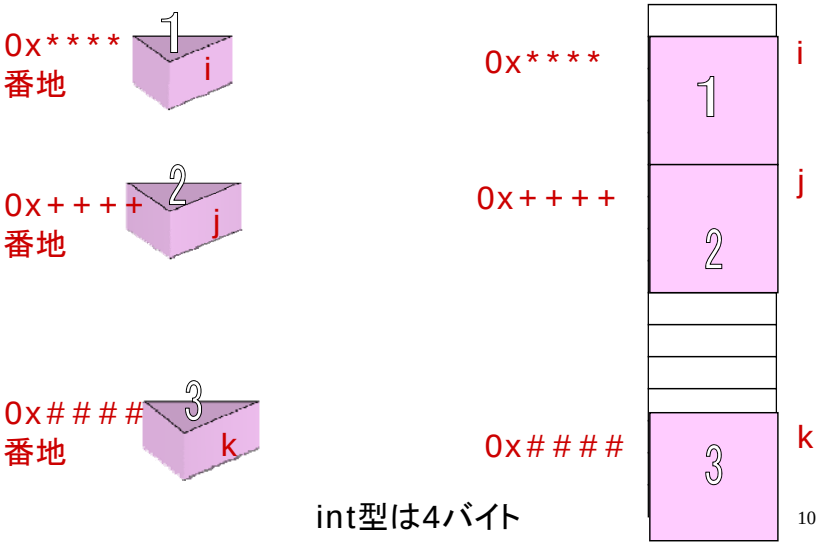
変数における
変数名と
アドレスと
中に入っている値





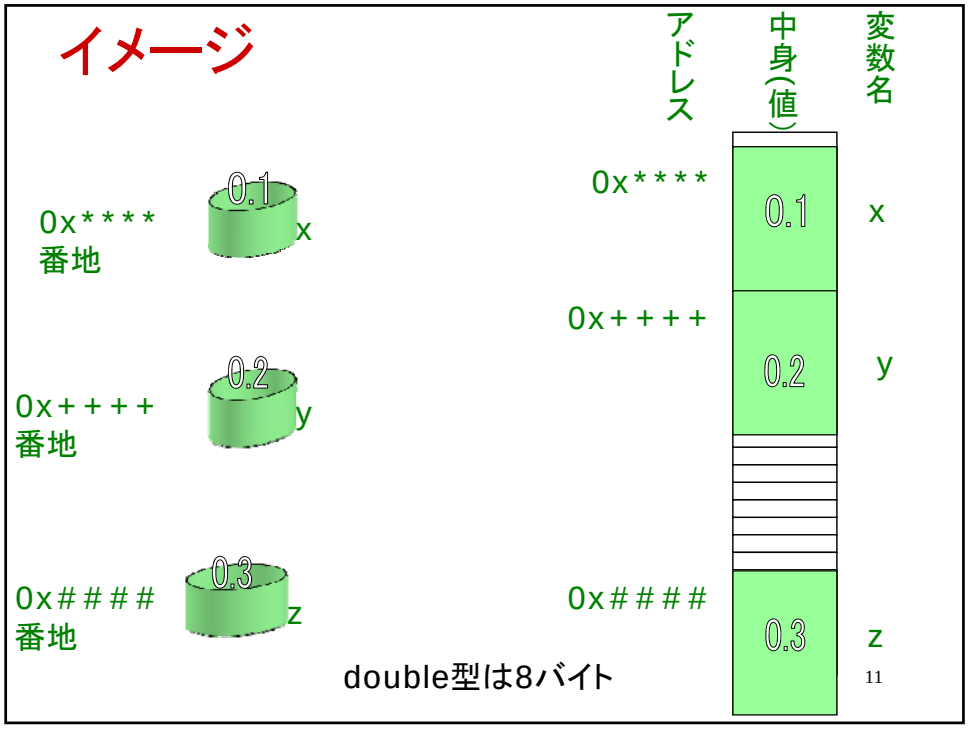
9

イメージ



int型は4バイト

10



練習1

```
/* 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

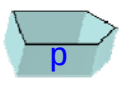
```
データ型　　＊ポインタの名前;
```

例

```
char  *p;
```

```
int  *q;
```

```
double  *r;
```



文字型の変数の
アドレス専用

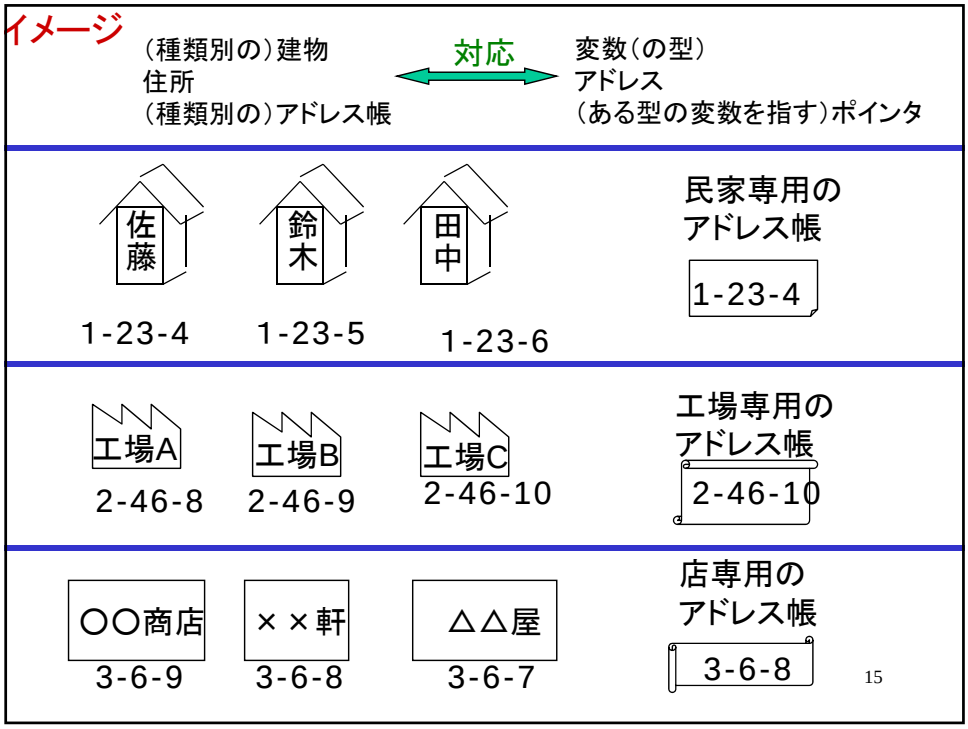
整数型の変数の
アドレス専用

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）

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

14



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。

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

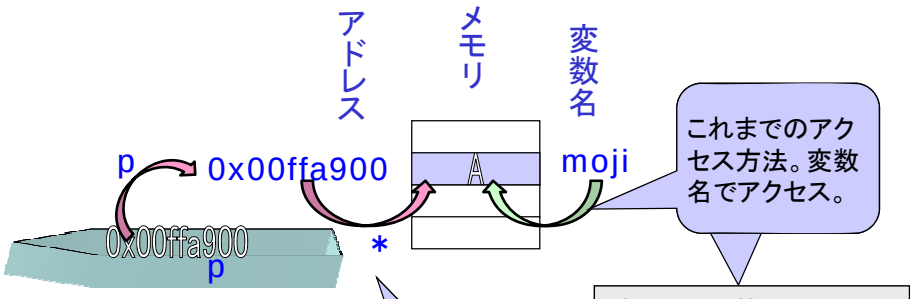
例

```
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

間接演算子を用いたメモリアクセス



これまでのアクセス方法。変数名でアクセス。

```
char moji;

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

```
char moji;
char *p;

p=&moji;

printf("%c",*p);
```

ポインタを用いたアクセス方法。ポインタ名と間接演算子を用いてアクセスする。アドレスの役割に注意する。

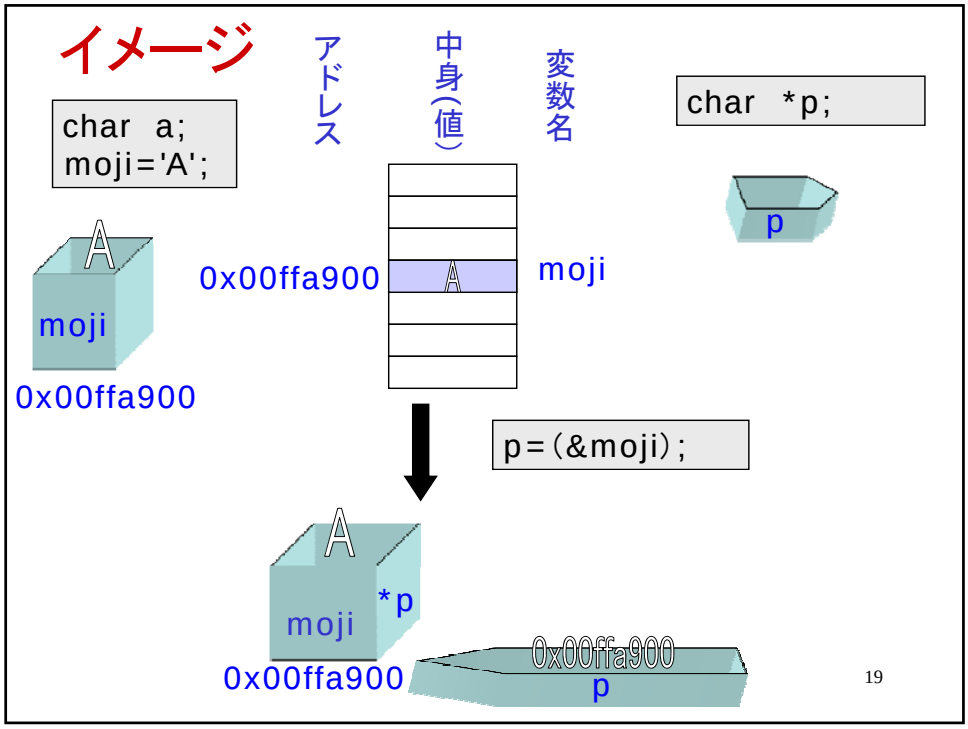
17

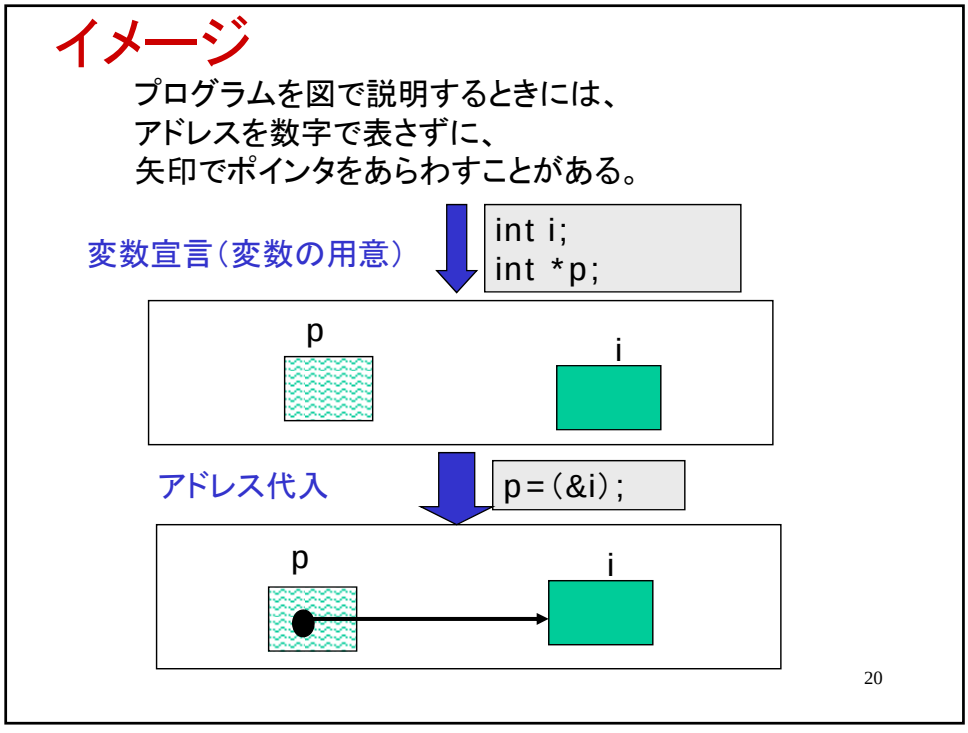
ポインタによる変数の別名

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。*/
```

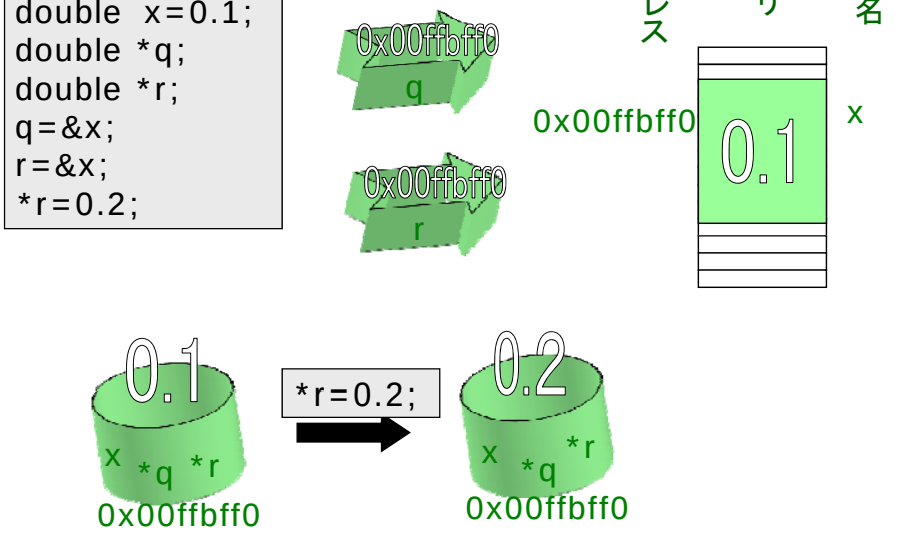
18


イメージ
アドレス
中身(値)
変数名
char a;
moji='A';
char *p;
A
moji
0x00ffa900
0x00ffa900
A
moji
p
p=(&moji);
A
moji
*p
0x00ffa900
0x00ffa900
p
19


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;
p
i
アドレス代入
p=(&i);
p
i
20

複数の別名

```
double  x=0.1;
double *q;
double *r;
q=&x;
r=&x;
*r=0.2;
```



21

NULLというアドレス

どの変数も指さない特別なアドレスを
NULL
として表す。

```
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
できる。

×

```
int     *p;
p=NULL;
/*次は間違い*/
*p=1;
```

NULLを格納するポインタ
へは間接演算子を用いる
ことはできない。

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¥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

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++;
```

は

```
*(p++);
```

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。

26

各種変数のアドレス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:B0zB0xx
        ソースファイル:print_address.c
        実行ファイル:print_address
        説明:変数とアドレスの関係を理解するためのプログラム。
             複数の関数内でアドレスを表示する。
        入力:標準入力から2つの整数値を入力する。
        出力:標準出力に以下を出力する。
             main関数ローカル変数のアドレスと値,
             main関数以外のローカル変数のアドレスと値、
             仮引数のアドレスと値。
           /* 次のページに続く      */
```

27

```
/*      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、グローバル変数は用いない。*/

/* プロトタイプ宣言*/
/*変数のアドレスを表示する関数*/
void function(int data1);
/*      次のページに続く      */
```

28

```
/*     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*/
        int    *p;            /*ポインタ*/

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

25

```
        /*      続き    ポインタの設定*/
        p=(&data2);

        /*関数呼び出し前*/
        /*変数の中身と変数アドレスの表示*/
        printf(“main関数内での表示¥n”);
        printf(“関数function呼び出し前¥n”);
        printf(“&data1=%10p”,&data1);
        printf(“  data1=%2d¥n”,data1);
        printf(“&data2=%10p”,&data2);
        printf(“  data2=%2d¥n”,data2);
        printf(“      &p=%10p”,&p);
        printf(“   p=%10p”,p);
        printf(“ *p=%2d¥n¥n”,*p);

        /*関数呼び出し*/
         function(data1);
/* 次に続く   */
```

```
        /*      続き    */
        /*関数呼び出し後*/
        /*変数の中身と変数アドレスの表示*/
        printf("main関数内での表示¥n");
        printf("関数function呼び出し後¥n");
        printf("&data1=%10p",&data1);
        printf("  data1=%2d¥n",data1);
        printf("&data2=%10p",&data2);
        printf("  data2=%2d¥n",data2);
        printf("      &p=%10p",&p);
        printf("   p=%10p",p);
        printf(" *p=%2d¥n¥n",*p);

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

31

```
/*ローカル変数の値とローカル変数のアドレスを表示する関数
仮引数 data1:main関数から値を受け取る。仮引数のアドレスを
              調べるために用意してある。
戻り値:なし
*/
void  function(int data1)
{
        /*ローカル変数宣言*/
        int data2;   /*整数型の変数。main関数内にある変数
                       と同じ名前にしてある。*/
        int * p;     /*ポインタ。main関数内にあるポインタ
                       と同じ名前にしてある。*/

        /*処理内容の通知*/
        printf("function 実行中¥n");

        /*      次に続く       */
```

```
        /*      続き    ポインタの設定*/
        p=(&data2);

        /*変数の中身と変数アドレスの表示*/
        printf("関数function内での表示¥n");
        printf("&data1=%10p",&data1);
        printf("  data1=%2d¥n",data1);
        printf("&data2=%10p",&data2);
        printf("  data2=%2d¥n",data2);
        printf("      &p=%10p",&p);
        printf("   p=%10p",p);
        printf(" *p=%2d¥n¥n",*p);

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

33

実行例

```
$./print_address < print_address.in
main関数内での表示
関数function呼び出し前
&data1=0xbfba3160  data1= 3
&data2=0xbfba315c  data2= 4
     &p=0xbfba3158 p=0xbfba315c  *p=4

function 実行中
関数function内での表示
&data1=0xbfba3140  data1= 3
&data2=0xbfba3134  data2=-1208557580
     &p=0xbfba3130  p=0xbfba3134  *p =-1208557580

main関数内での表示
関数function呼び出し後
&data1=0xbfba3160  data1= 3
&data2=0xbfba315c  data2= 4
     &p=0xbfba3158 p=0xbfba315c  *p=4

$
```

第12回ポインタの応用
（ポインタと関数、配列）

1

今回の目標

- 関数呼び出しとポインタの関係を理解する。
- 配列とポインタの関係を理解する。
- 関数間での配列を引き渡し方を理解する。

☆他の関数内の2つの変数の値を交換する関数を作成し、その関数を用いて関数内の2次元配列の2つの行を交換する関数を作成する。

2

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

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

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。

例

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

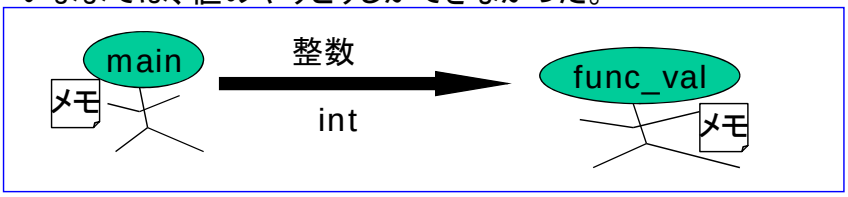
```
int main()
{
   func_ref(&i);
   a=i;
}
```

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。

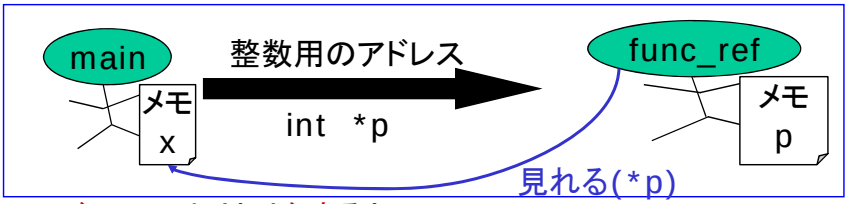
3

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。

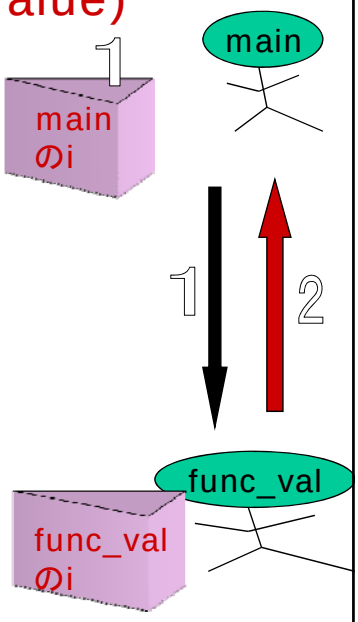


アドレスのやりとりをすると、
他の関数内のローカル変数の内容を変更できる。

4

値による呼び出し(call by value)

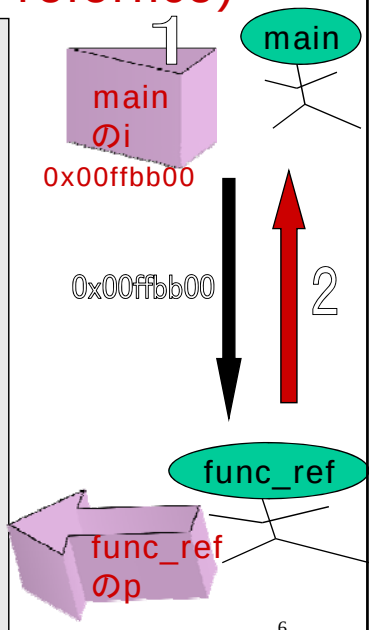
```
/*test_cbv.c 値による呼び出し実験*/
#include <stdio.h>
int  func_val(int);
int  main()
{
        int i=1;
        int j=0;
        printf("i=%d  j=%d¥n",i,j);

        j=func_val(i);

        printf("i=%d  j=%d¥n",i,j);
        return  0;
}
int  func_val(int i)
{
        i++;
        return i;
}
```



5

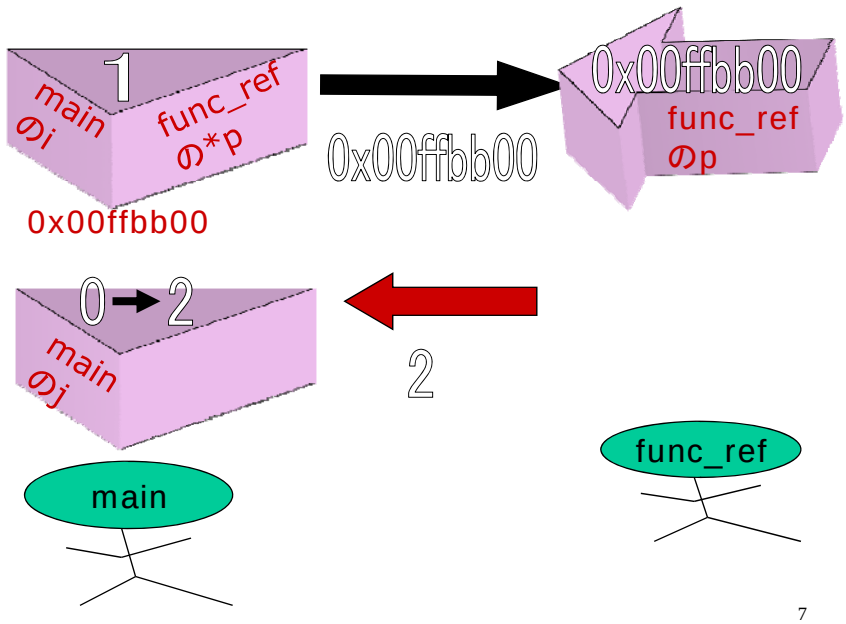
参照による呼び出し(call by refernce)

```
/*test_cbr.c  参照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);
}
```



6

イメージ



7

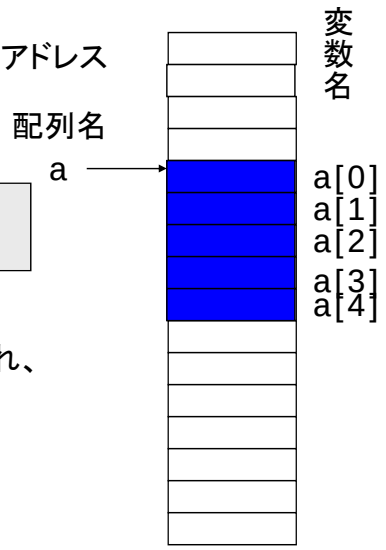
配列とポインタ

C言語では、配列名は先頭の要素のアドレスを指す。

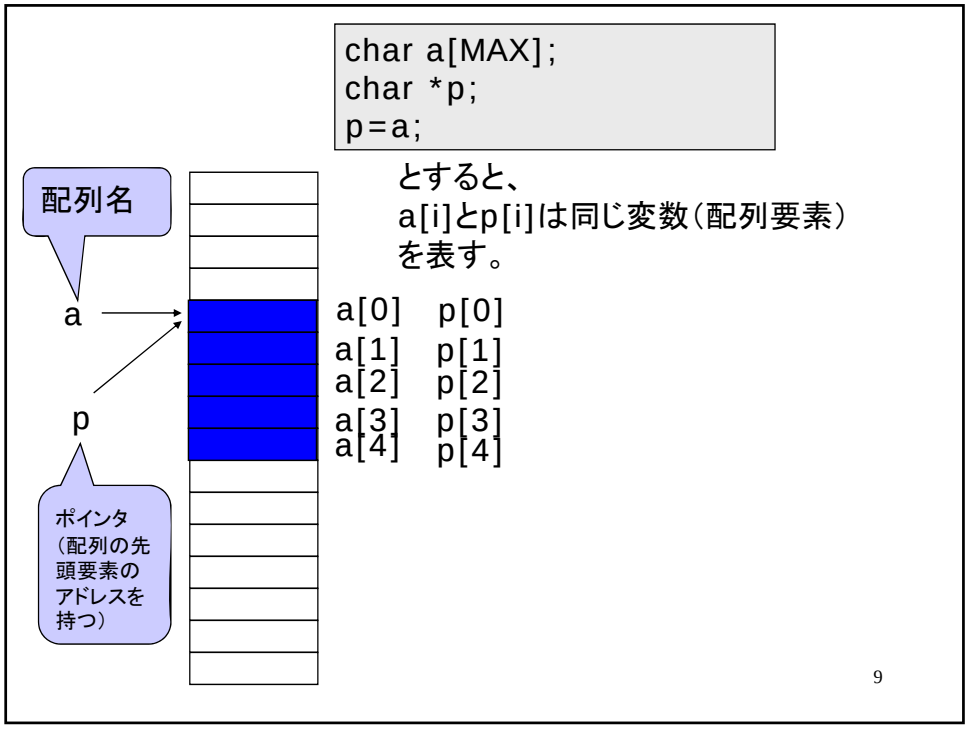
例えば、

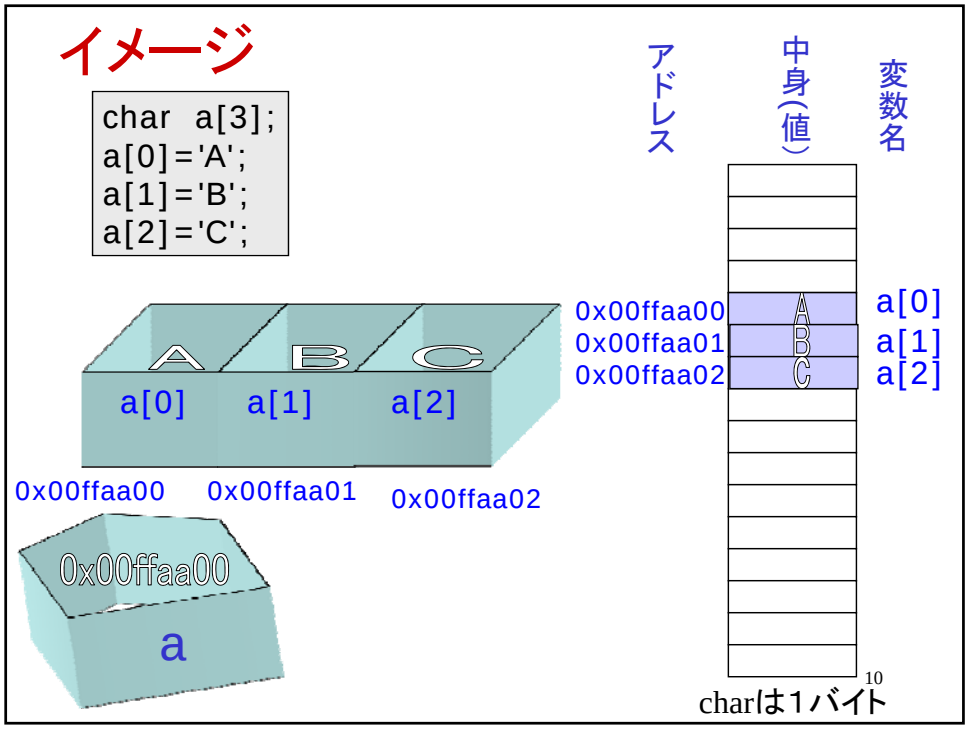
```
#define  MAX  5
char  a[MAX];
```

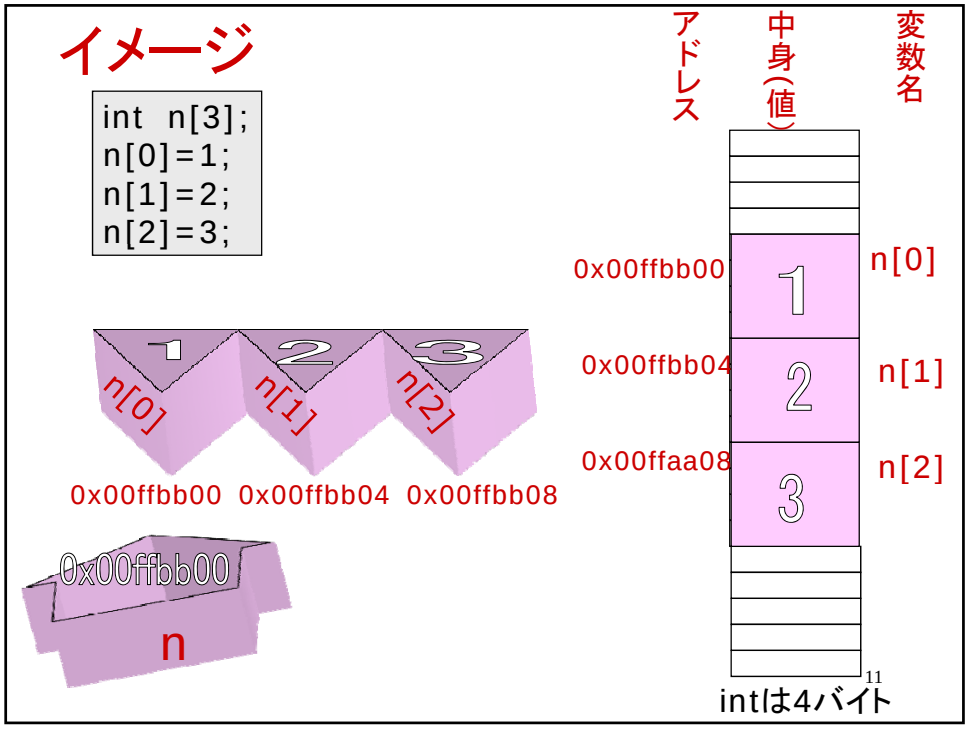
と宣言するとアドレスの連続した
5個のchar変数がメモリ上に確保され、
その先頭のアドレスがaにはいる。
つまり、aには「&a[0]の値（アドレス）」が保持されている。

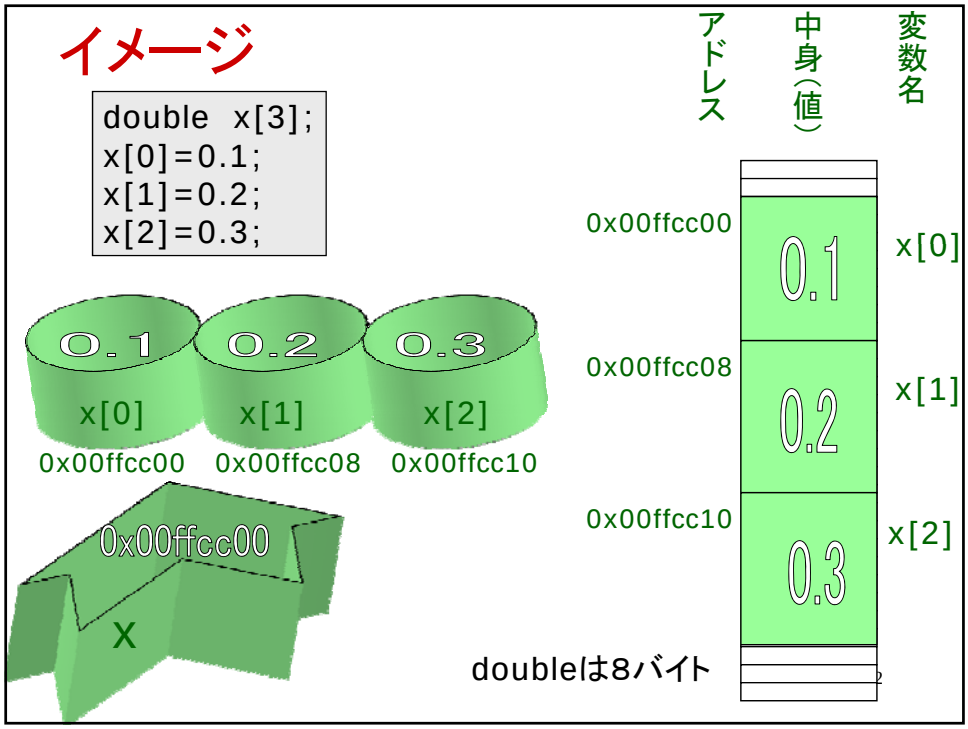


8


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。
配列名
a
p
ポインタ（配列の先頭要素のアドレスを持つ）
a[0] p[0]
a[1] p[1]
a[2] p[2]
a[3] p[3]
a[4] p[4]
9


イメージ
char a[3];
a[0]='A';
a[1]='B';
a[2]='C';
A B C
a[0] a[1] a[2]
0x00ffaa00 0x00ffaa01 0x00ffaa02
0x00ffaa00
a
アドレス
中身（値）
変数名
0x00ffaa00
0x00ffaa01
0x00ffaa02
A
B
C
a[0]
a[1]
a[2]
10
charは1バイト


イメージ
int n[3];
n[0]=1;
n[1]=2;
n[2]=3;
1
2
3
n[0]
n[1]
n[2]
0x00ffbb00
0x00ffbb04
0x00ffbb08
0x00ffbb00
n
アドレス
中身（値）
変数名
0x00ffbb00
0x00ffbb04
0x00ffaa08
n[0]
n[1]
n[2]
11
intは4バイト


イメージ
double x[3];
x[0]=0.1;
x[1]=0.2;
x[2]=0.3;
0.1
0.2
0.3
x[0]
x[1]
x[2]
0x00ffcc00
0x00ffcc08
0x00ffcc10
0x00ffcc00
x
アドレス
中身（値）
変数名
0x00ffcc00
0x00ffcc08
0x00ffcc10
x[0]
x[1]
x[2]
doubleは8バイト

練習2

```
/* pointer_array.c ポインターと配列実験 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;
         }

      /*     次に続く       */
```

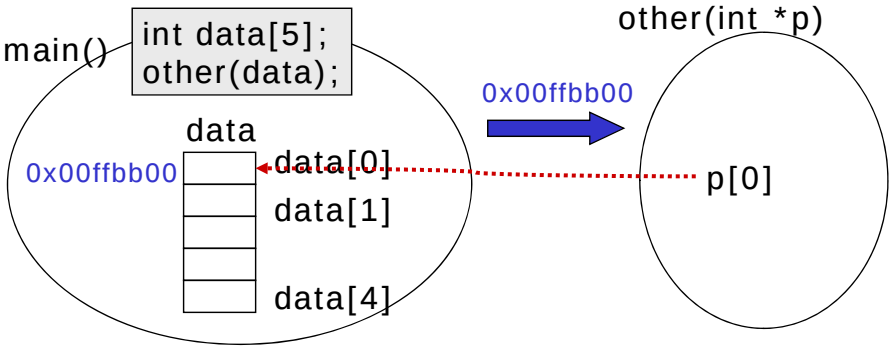
13

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

14

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。

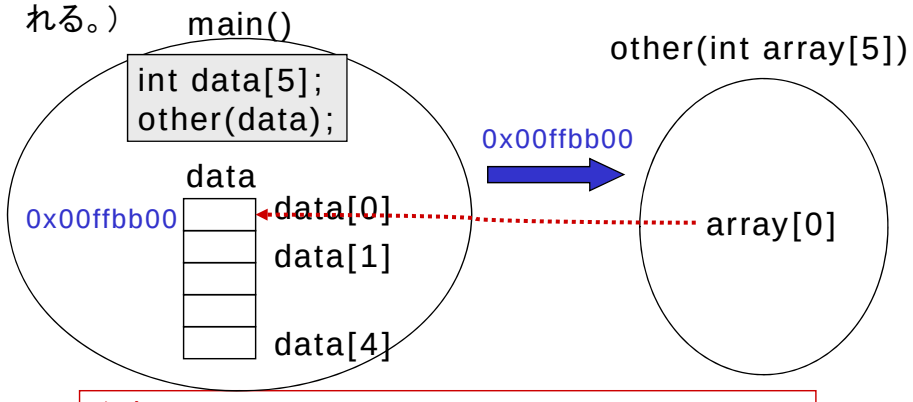


受け取った他の関数の中では、配列の先頭要素のアドレスの入ったポインタを配列名のように使うことができる。

15

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。

16

練習3

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

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

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

17

```
        /*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*/

        /*次に続く*/
```

18

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

/*      次に続く         */
```

注意：
呼ばれる方では、配列の最後に気を付ける事。

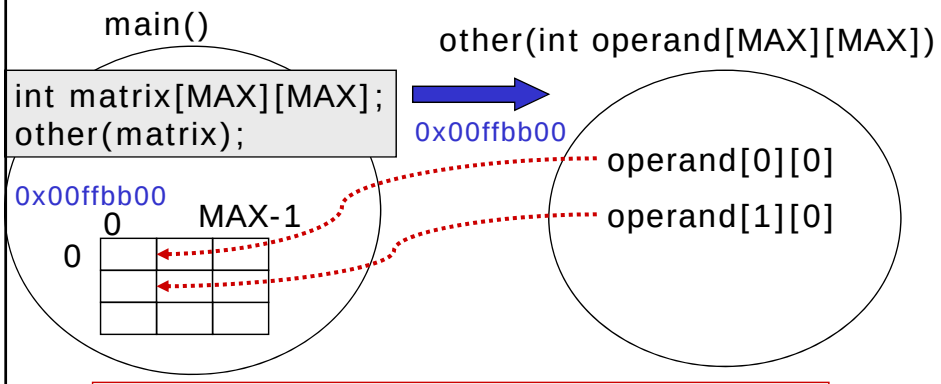
19

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

20

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。

21

他の関数内の2次元配列の2つの行を交換する

```
/*
        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：arg_matrix.c
        実行ファイル：arg_matrix
        説明：関数間での2次元配列の受け渡しの効果を調べるための
              プログラム。他の関数内の行列（2次元配列）の2行を交換
              する関数を作成する。

        入力：標準入力からGYO×RETU個のint値を行列の成分として
              受け取る。ただし、同じ行の値が連続しているとする。
              さらに、交換したい行番号を2つ標準入力から受け取る。
        出力：標準出力に、行交換前の行列および行交換後の行列を
              出力する。
/*      次のページに続く          */
```

22

```
/*      続き     */
/* ヘッダファイルの読み込み*/
#include <stdio.h>
/* マクロの定義 */
#define  GYO 3
#define  RETU 4

/* プロトタイプ宣言*/
void print_matrix(int matrix[GYO][RETU]);
       /*配列matrixを表示する関数*/
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*/
void  swap_rows(int matrix[GYO][RETU],int row1,int row2);
       /*配列matrixのrow1行とrow2行を入れ替える関数*/

/*      次のページに続く          */
```

```
/*続き*/

/*main関数開始*/
int      main()
{
        /*ローカル変数宣言*/
        int      matrix[GYO][RETU];/*データ入力用配列*/
        int      i;        /*行番号を制御、ループカウンタ*/
        int      j;        /*列番号を制御、ループカウンタ*/
        int      row1; /*交換したい行番号1*/
        int      row2;   /*交換したい行番号2*/

/*続く   main関数*/
```

24

```
/*続き　main関数内*/
        /*入力処理*/
        printf(“行ごとに入力して下さい。¥n");
        for(i=0;i<GYO;i++){
           for(j=0;j<RETU;j++){
              scanf("%d",&matrix[i][j]);
            }
        }

        printf(“交換したい行番号を2つ入力して下さい。¥n");
        printf(“（0以上%d未満）“,GYO);

        scanf(“%d”,&row1);
        scanf(“%d”,&row2);
        if(row1<0||GYO<row1||row2<0||GYO<row2){
           printf(“不正な入力です。");
           return -1;
         }

/*続く　main関数*/
```

```
/*続き　main関数内*/
        /*交換前の行列の表示*/
        printf(“交換前¥n”);
        print_matrix(matrix);

         /*行列の行交換*/
        printf(“¥n%3d 行と%3d行を交換中¥n¥n”,row1,row2);
        swap_rows(matrix,row1,row2);

        /*交換後の行列の表示*/
        printf(“交換後¥n”);
        print_matrix(matrix);

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

/*続く*/
```

```
/*続き  print_matrixの定義開始*/
/*行列を表示する関数
仮引数matrix:表示したい行列
戻り値:なし(void)
*/
void print_matrix(matrix[GYO][RETU]){
        /*ローカル変数宣言*/
        int     i;      /*行番号を制御、ループカウンタ*/
        int     j;      /*列番号を制御、ループカウンタ*/

        /*表示処理*/
        for(i=0;i<GYO;i++){
          for(j=0;j<RETU;j++){
             printf("%3d",matrix[i][j]);
          }
           printf("¥n");
        }

        return;
}
/*print_matrix関数定義終了 続く*/
```

```
/*続き  関数swapの定義開始*/
/*他の関数の2変数の値を入れ替える関数
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)
*/
void swap(int *p,int *q){
        /*ローカル変数宣言*/
        int     temp;  /*交換のために値を一時的の保存する*/

        /*交換処理*/
        temp=*p;
        *p=*q;
        *q=temp;

        return;
}
/*関数swapの定義終了 続く*/
```

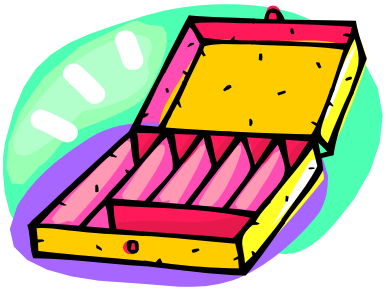
```
/*続き  関数swap_rowsの定義開始*/
/*行列の2行を入れ替える関数
仮引数matrix:GYO×RETUの行列
仮引数row1,row2:交換すべき行番号
      （呼び出された時点で(0<=row1<GYO) && (0<=row2<GYO)
      を満たす。）
戻り値:なし(void)*/
void swap_rows(matrix[GYO][RETU],int row1,int row2){
        /*ローカル変数宣言*/
        int      j;        /*列番号を制御、ループカウンタ*/

        /*交換処理*/
        for(j=0;j<RETU;j++){
            swap(&matrix[row1][j],&matrix[row2][j]);
        }

        return;
}
/*関数swap_rowsの定義終了*/
/*プログラム（arg_matrix.c)の終了*/
```

実行例

```
$./arg_matrix<arg_matrix.in
行ごとに入力して下さい。
交換したい行番号を2つ入力して下さい。
（0以上％d未満）
交換前
 1  2  3  4
 5  6  7  8
 9 10 11 12

 0行と  2行を交換中

交換後
 9 10  11 12
 5  6  7  8
 1  2  3  4
$
```

30

第13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。

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

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、定数、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。）

ー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

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;
```

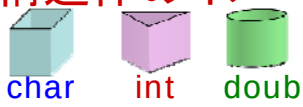
この2つで、一つの型を表わしているので注意すること。

参考

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

6

構造体のイメージ



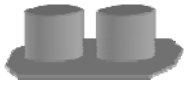
既存の型

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。

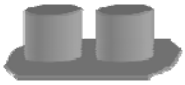


struct complex型の雛形

7

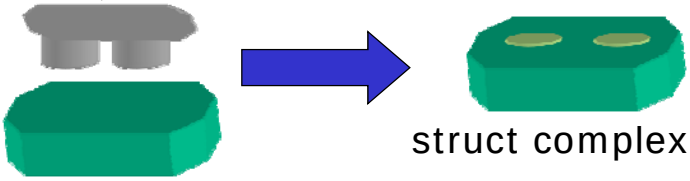
構造体型の変数宣言

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



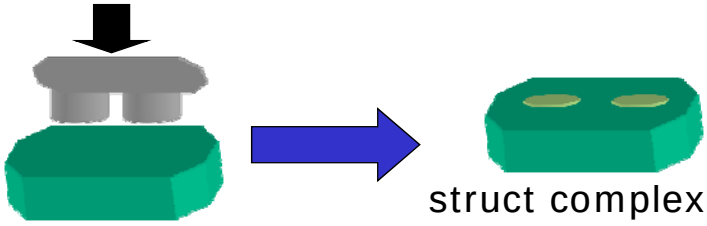
struct complex型の雛形

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。

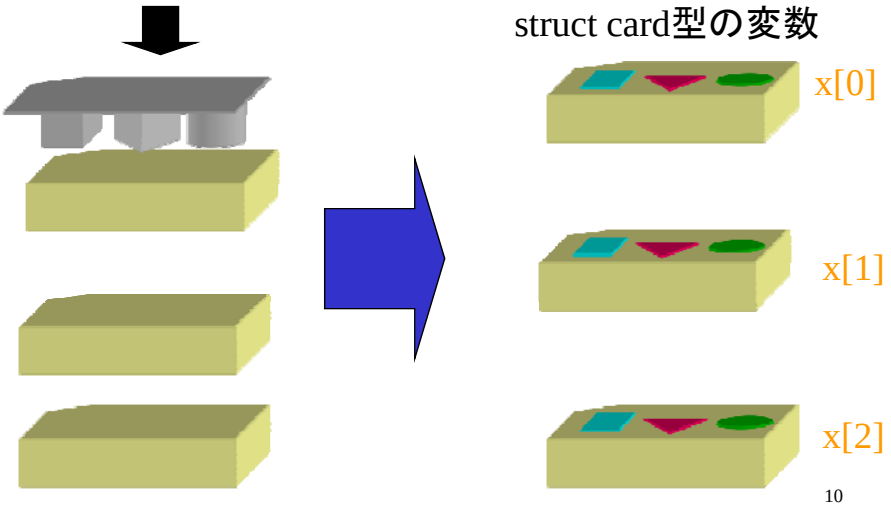
struct card 型の雛形

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

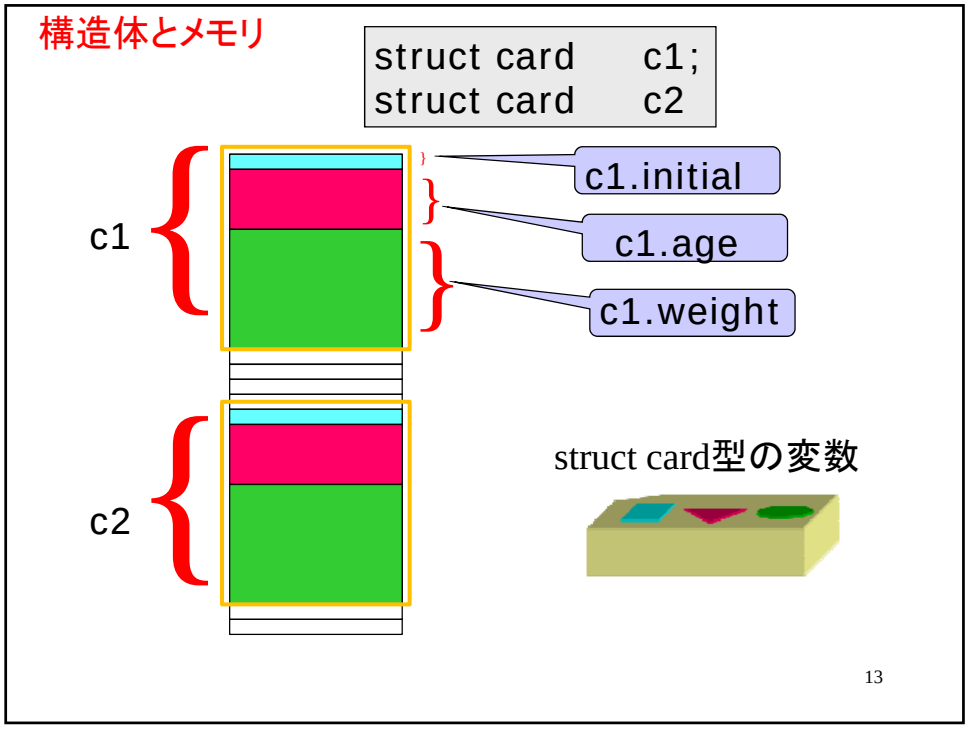
struct complex型の変数

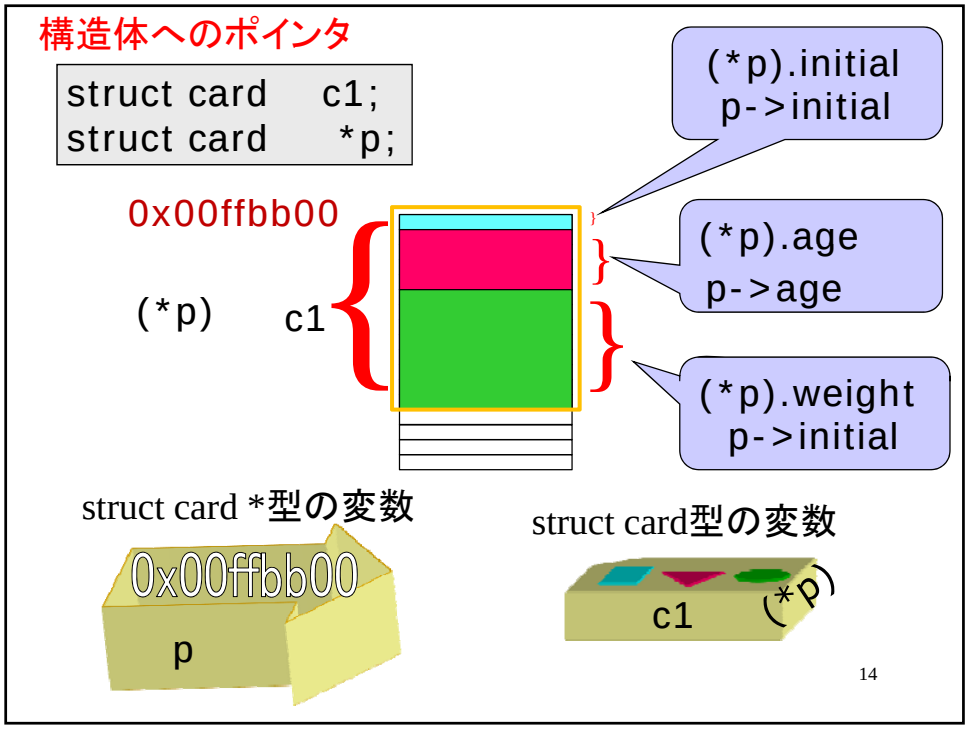
z1

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;
```

間違い例

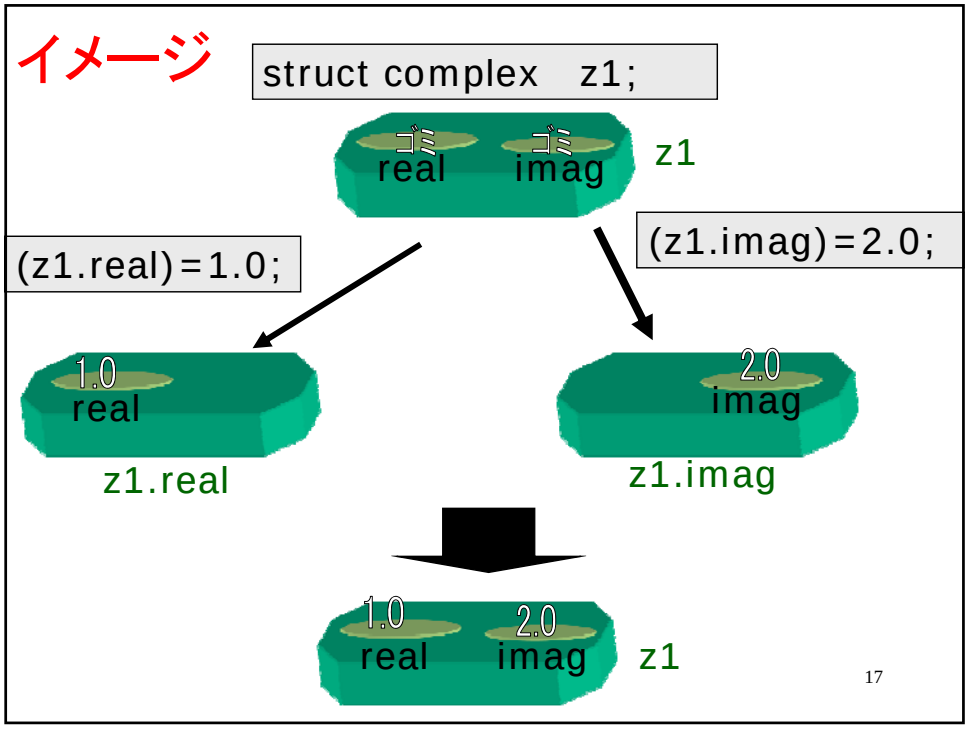
```
z1=1.0+2.0i;
```

複素数だからってこんなふうにはかけない。

```
z1=(1.0,2.0);
```

ベクトル風にもかけない。

16



構造体と代入演算子2（構造体への値の入れ方2）

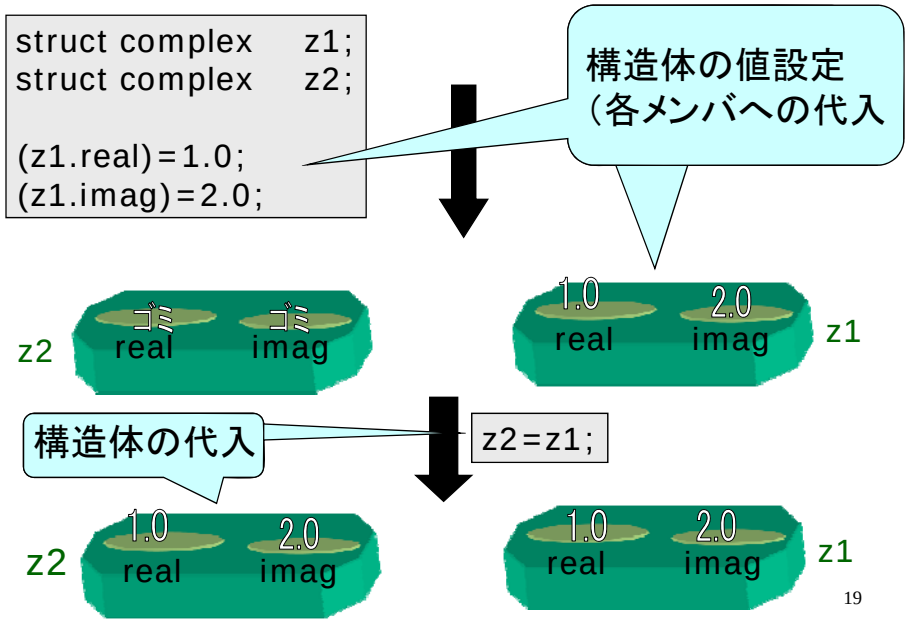
同じ型の構造体同士で代入する。

```
struct complex      z1;
struct complex      z2;

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

z2=z1;
```

18

イメージ

```
struct complex    z1;
struct complex    z2;

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



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 %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*/
```

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: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。
*/
/*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(void);
/*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  plus_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関数開始*/
int   main()
{     /*ローカル変数宣言*/
      struct complex  z1;     /*複素数1*/
      struct complex  z2;     /*複素数2*/
      struct complex  sum; /*複素数の和を蓄える変数*/

      /*入力処理*/
      z1=scan_complex();
      z2=scan_complex();

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

/*main関数続く*/
```

25

```
/*続き main関数*/

      /*出力処理*/
      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、関数scan_complexの定義*/
/*標準入力から複素数を受け取る関数。
実部、虚部の順にdouble 値を受け取る。
仮引数 :なし(void)
戻り値:読み込まれた複素数。
*/
struct complex  scan_complex(void)
{
        /*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、関数scan_complexの定義*/
/*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:表示される複素数
戻り値:なし(void)
*/
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

```
/*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  plus_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plus_complexの終了    */
/*プログラムpluscomp.c の終了*/
```

29

実行結果

```
$./pluscomplex
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