2019年度　計算機システム（演習）
第1回
2019.11.29

遠藤 敏夫（数理・計算科学系 教授）
野村 哲弘（数理・計算科学系 助教）

はじめに

- 講義・演習のページ
  - OCW-iの「計算機システム」のページ
  - 内容：授業のスライド、連絡事項等
  - 授業： 火曜5,6限 (13:20～14:50)，金曜5~8限 (13:20～16:35)
- 演習内容
  - W621で講義 [前半]＋W73F計算機室で演習 [後半]
  - C言語プログラミングの基礎
  - MIPSシミュレータを用いたMIPSアセンブリプログラミング
  - C言語によるMIPSシミュレータの作成
- 質問等は…
  - 講義・演習の授業後 @演習室
  - メール
    - compsys19@el.gsic.titech.ac.jp

各種連絡・資料について

- 各種連絡はOCW-iのメーリングリストを使用します
  - 事務連絡(休講情報，講義や演習について，etc)
  - ML配信時は同じ内容をOCW上にも掲載予定
  - 追加申告予定者は早めに登録してください
- 講義・演習資料のPDFはOCW-iにアップロードされます
  - 特に演習の資料にあるコマンドラインの例はコピーペーストでは動かないことがあります
    - MS Officeがハイフンをダッシュに自動変換する所為

成績について

- 以下の２点から評価
  - 講義の試験 (学期末に１度行う)
  - 演習
    - プログラミング課題 (全8回程度の予定)

授業の概要

- 目的： 高水準言語が中間言語を経て如何にしてプロセッサ上で実行（計算）されるかを実践を通して理解する

```
int i = 2;
int j = 3;
i  = i  +  j;
```

高水準言語
(C言語)

→ C言語の基本

コンパイラ

→ コンパイラの授業

```
li   $t0,  2
li   $t1,  3
add   $t0,  $t0,  $1
```

アセンブリ言語
(MIPS)

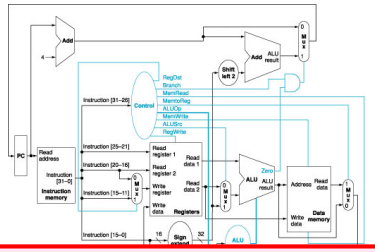
→ MIPSプログラミング
(MIPSシミュレータ)

アセンブラ

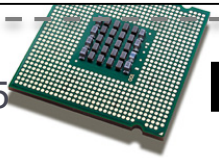
→ オプション課題

```
0000001010101 10
0111101100000 10
0101111000011 01
0000010000001 01
```

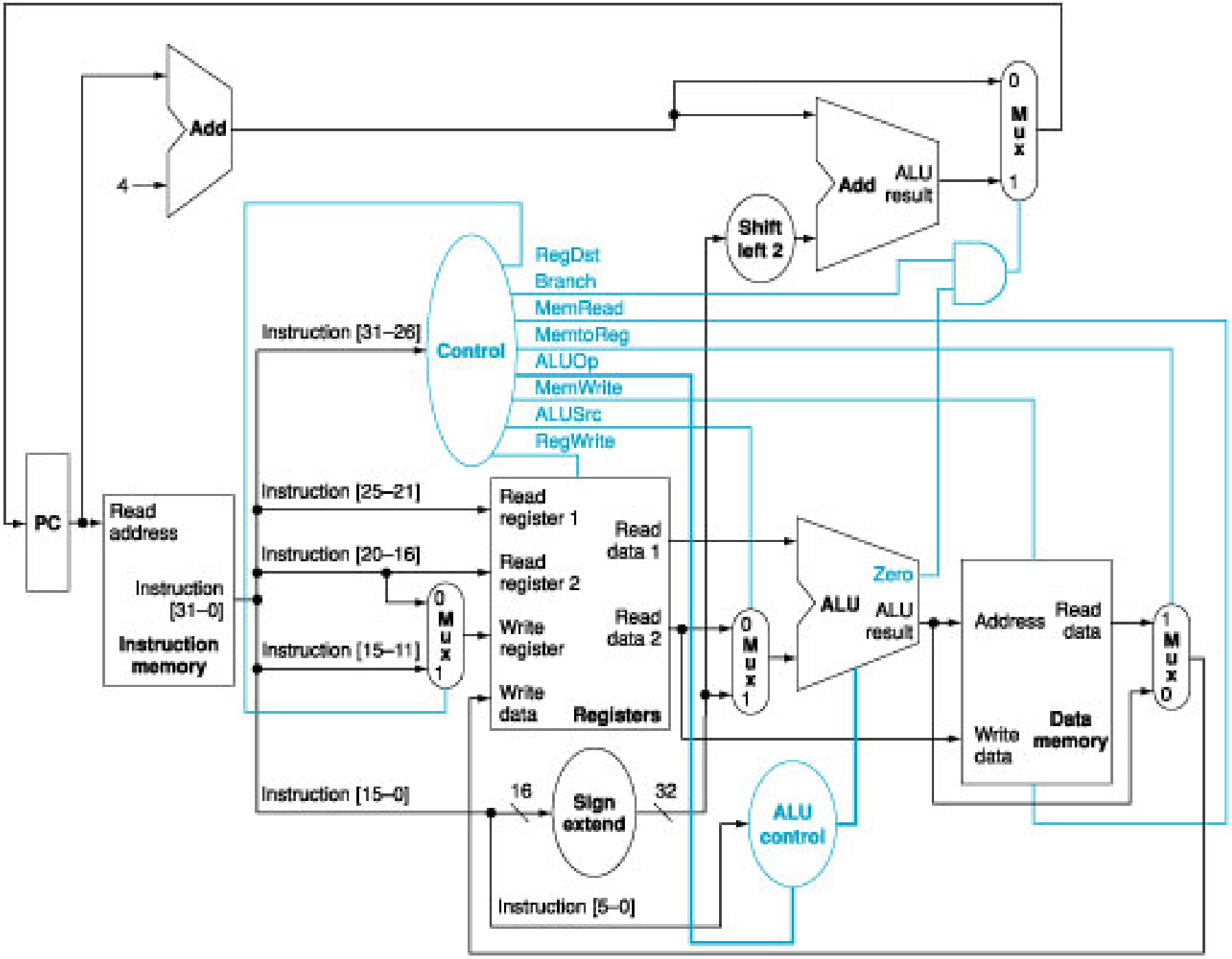
機械語
(MIPS)



自作MIPSシミュレータ
(Cを使用)



計算結果


Add
4
0
M
u
x
1
Add
ALU result
Shift left 2
RegDst
Branch
MemRead
MemtoReg
ALUOp
MemWrite
ALUSrc
RegWrite
Control
Instruction [31–26]
PC
Read address
Instruction [31–0]
Instruction memory
Instruction [25–21]
Instruction [20–16]
Instruction [15–11]
Instruction [15–0]
Instruction [5–0]
Read register 1
Read register 2
Write register
Write data
Read data 1
Read data 2
Registers
16
Sign extend
32
ALU control
Zero
ALU
ALU result
Address
Read data
Write data
Data memory

演習の日程（予定）

- 第1回：
- 第2回：
- 第3回：
- 第4回：
- 第5回：
- 第6回：
- 第7回：
- 第8回：

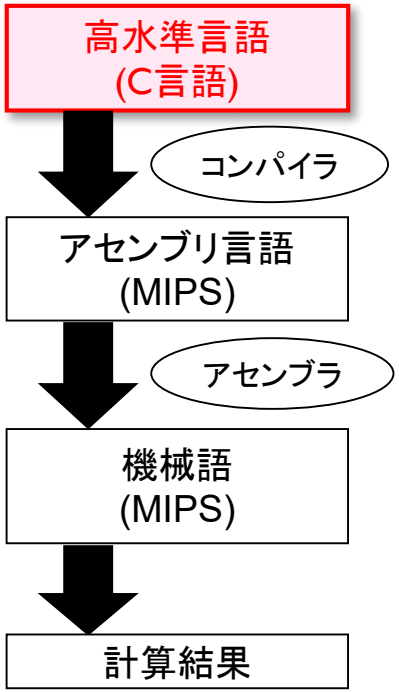
C言語の基礎：制御構文、配列、ポインタ

C言語の基礎：構造体、連結リスト

MIPS：基礎、SPIM、各種命令、配列

MIPS：サブルーチン (caller/callee-save) 、動的メモリ割り当て

MIPSシミュレータ作成



本日の内容

C言語の基礎, ポインタと配列, 作業環境準備

C言語

- 手続き型言語
  - 手続き（ルーチン or 関数 => Javaのメソッドに相当）の組み合わせでプログラムを実装
- Java にない特徴
  - クラスという概念がない
  - 明示的にポインタを用いてメモリにアクセス可能
- 用途
  - 電化製品などの組み込みシステム (Nintendo DS)
    - 今ではJava, Swiftなど色々あるが…
  - システムプログラミング (Unix, Linux, compiler)
  - 科学技術計算 (物理、化学、天文学、、、)
    - その他、Fortranもよく使われる

CとJavaの類似点

- 基本データ型の種類
  - char, short, int, long, float, double, void など
- 演算子
  - =, ==, <=, >= +, -, *, /, %, ++, --, &&, ! など
- 制御構文
  - (for), while, switch, break, continue, return　など
- コメント
  - /* comment */
  - // comment (元はC++のみだった)

CとJavaのプログラム比較

- 1から10までの和を計算するプログラム
  - for文内の局所変数(*i*)はあらかじめ宣言する必要がある。
    - コンパイラによってはどちらでも受け付けるが、右のようにするのが無難
      - 左側はC99 という比較的最近の規格で導入された記法

sample1.c

```
class Sum {
  /* 総和を計算 */
  public static void main(String[] args) {
    int sum = 0;
    for (int i = 1; i <= 10; i++) {
      sum = sum + i;
    }
  }
}
```

```
/* 総和を計算 */
int main() {
  int sum = 0;
  int i;
  for (i = 1; i <= 10; i++) {
    sum = sum + i;
  }
  return 0;
}
```

CとJavaのプログラム比較：print

- ライブラリで提供されている関数を使う場合には「#include」でヘッダーファイルを読込む
  - printf関数はCの標準ライブラリで提供されている関数で、関数の情報（型情報）がヘッダーファイル stdio.h で宣言されている
  - Javaのimportに相当

sample2.c

```
class Sum {
  /* 総和を計算 */
  public static void main(String[] args) {
    int sum = 0;
    for (int i = 1; i <= 10; i++) {
      sum = sum + i;
    }
    System.out.println( “sum= “+sum);
  }
}
```

```
#include <stdio.h>
/* 総和を計算 */
int main(int argc, char **argv)
  int sum = 0;
  int i;
  for (i = 1; i <= 10; i++) {
    sum = sum + i;
  }
  printf( “sum=%d¥n", sum);
  return 0;
}
```

printf関数

- 文字列をフォーマットして出力する関数
  - int printf (“フォーマット文字列”, 変数1, 変数2, ...)

```
int i      = 100;
double d   = 0.01;
char c     = 'x';
char *str = "ABC";

printf("i=%d¥n", i);
printf("d=%f¥n", d);
printf("c=%c¥n", c);
printf("str=%s¥n", str);
printf("i=%d, d=%f¥n", i, d);
```

フォーマット指定子	意味
%d	整数
%f	浮動小数点数
%c	文字
%s	文字列

CとJavaの相違点 (1/2)

- boolean型が無い
  - ライブラリとしてはある (stdbool.h)
  - 条件判断には、false<=>"0"、true<=>"0以外"を使用
    - e.g.) 1 == 2は0、 1<= 2は1の値をとる

sample3.c

```
#include <stdio.h>
int main(void)
{
  int no=2;
  if (no % 2) {
    printf("no is odd");
  } else {
    printf("no is even");
  }
}
```

```
no is even
```

sample4.c

```
#include <stdio.h>
int main() {
  int no1 = 1;
  int no2 = 2;
  printf("no1==no1 -> %d¥n", no1 == no1);
  printf("no1==no2 -> %d¥n", no1 == no2);
  printf("no1<=no2 -> %d¥n", no1 <= no2);
}
```

```
no1==no1 -> 1
no1==no2 -> 0
no1<=no2 -> 1
```

CとJavaの相違点 (2/2)

- 配列へのアクセス
  - 配列のサイズを超えてアクセスしても検知されない
  - 配列の長さをしっかりチェック
    - e.g.) バッファオーバーラン
- (関数の定義)
- ポインタ型・演算子
  - アドレス演算子: &、間接演算子： *、アロー演算子： ->
- String型がない
  - charの配列として表現
- 例外処理が無い
  - 関数の戻り値などを細かくチェック
  - e.g.) malloc()など
- (ローカル変数はブロックの先頭でしか宣言できない)
  - ブロック:「{...}」で囲まれた部分
  - 「{...}」で囲ってブロックを意図的に作っても良い
  - (※最近のコンパイラだと、どれもコンパイルはできる)

配列の定義とアクセス (1/2)

- 定義方法

```
int a[3];
int b[3] = {1, 2, 3};
int c[]  = {1, 2, 3};
int d[3] = {1, 2};
```

- int a[3]; ← サイズ3の配列を作成、初期化なし
- int b[3] = {1, 2, 3}; ← サイズ3の配列を作成、各要素を初期化
- int c[] = {1, 2, 3}; ← 同上
- int d[3] = {1, 2}; ← サイズ3の配列を作成
  0番、1番要素のみ1, 2 で初期化、後は 0

- アクセス方法
  - 最初の要素の添字は 0
  - i 番目の要素は a[i]
  - 配列のサイズを超えてアクセスしても例外は投げられない
    - ただし、動作が不安定になるか、異常終了するので、確認は必要

配列の定義とアクセス (2/2)

sample5.c

```
#include <stdio.h>
int main() {
  int a[3];
  int b[3] = {1, 2, 3};
  int c[]  = {1, 2, 3};
  int d[3] = {1, 2};
  int i;
  for (i = 0; i < 3; i++) {
    printf("a[%d]=%d, ", i, a[i]);
    printf("b[%d]=%d, ", i, b[i]);
    printf("c[%d]=%d, ", i, c[i]);
    printf("d[%d]=%d ¥n"  , i, d[i]);
  }
  printf("a[%d]=%d¥n", 3, a[3]);
  return 0;
}
```

出力

```
a[0]=1627408016, b[0]=1, c[0]=1, d[0]=1
a[1]=-1,         b[1]=2, c[1]=2, d[1]=2
a[2]=1,          b[2]=3, c[2]=3, d[2]=0
a[3]=4198562
```

※配列全体を0で初期化

- int a[3] = {0}

配列の範囲を超えているので、本当は誤り

関数の定義 (1/2)

sample6.c

```
#include <stdio.h>
int sum(int f, int l)
{
  int sum = 0;
  int i;
  for (i = f; i <= l; i++) {
    sum += i;
  }
  return sum;
}

int main()
{
  int s;
  s = sum(1, 10);
  printf("sum=%d¥n", s);
  return 0;
}
```

- int sum (int f, int l)
  - fからl (f <=l)の総和を求める関数

戻り値型 関数名 仮引数の宣言

int sum (int f, int l)

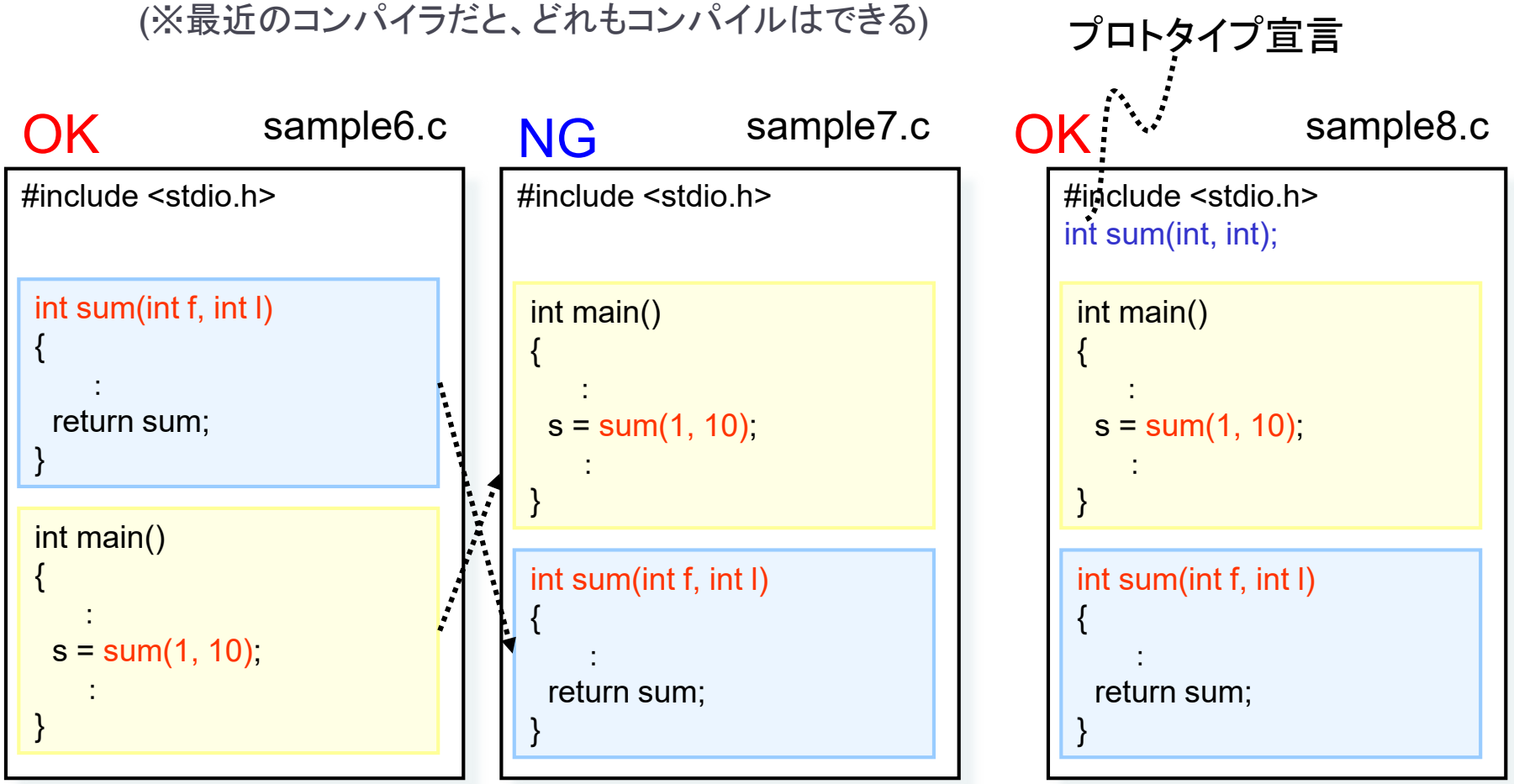
関数名 実引数

sum (1, 10)

関数の定義 (2/2)

- 注意：関数は使用する前に定義
  - しかしプロトタイプ宣言を行えばOK

    (※最近のコンパイラだと、どれもコンパイルはできる)



値渡しと参照渡し

sample9.c

```
#include <stdio.h>
void sum(int f, int l, int s, int a)
{
  int sum = 0;
  int i;
  for (i = f; i <= l; i++) {
    sum += i;
  }
  s = sum;
  a = sum / (l – f + 1);
}

int main()
{
  int s = 0, a = 0;
  sum(1, 10, s, a);
  printf("s=%d, a=%d¥n", s, a);
  return 0;
}
```

- 総和と平均を同時に計算したい。
  - しかし、関数の戻り値は1つ
  - void sum(int f, int l, int s, int a)を定義

出力

```
s=0, a=0
```

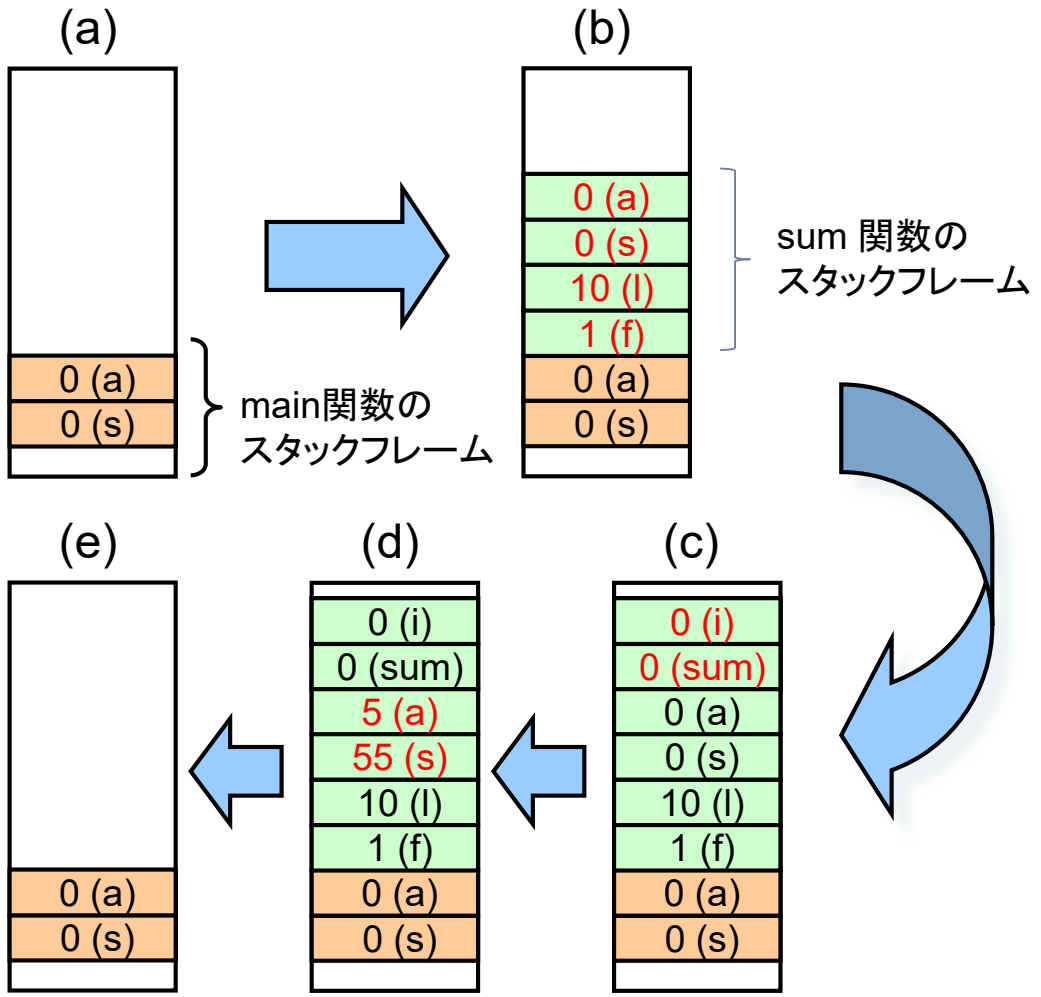
- どうすればよいか？

スタック領域

1. 関数内の引数、局所変数の値は、その関数が呼ばれるとスタック領域に内容が積まれる
2. 関数の実行が完了すると、その局所変数の値は解放される（他の用途で使われる）

```
#include <stdio.h>
void sum(int f, int l, int s, int a)
{
  int sum = 0;          } ..... (c)
  int i;
        :
  s = sum;              }
  a = sum / (l – f + 1); } ..... (d)
}

int main()
{
  int s = 0, a = 0; ..... (a)
  sum(1, 10, s, a); ..... (b)
        : (e)
}
```

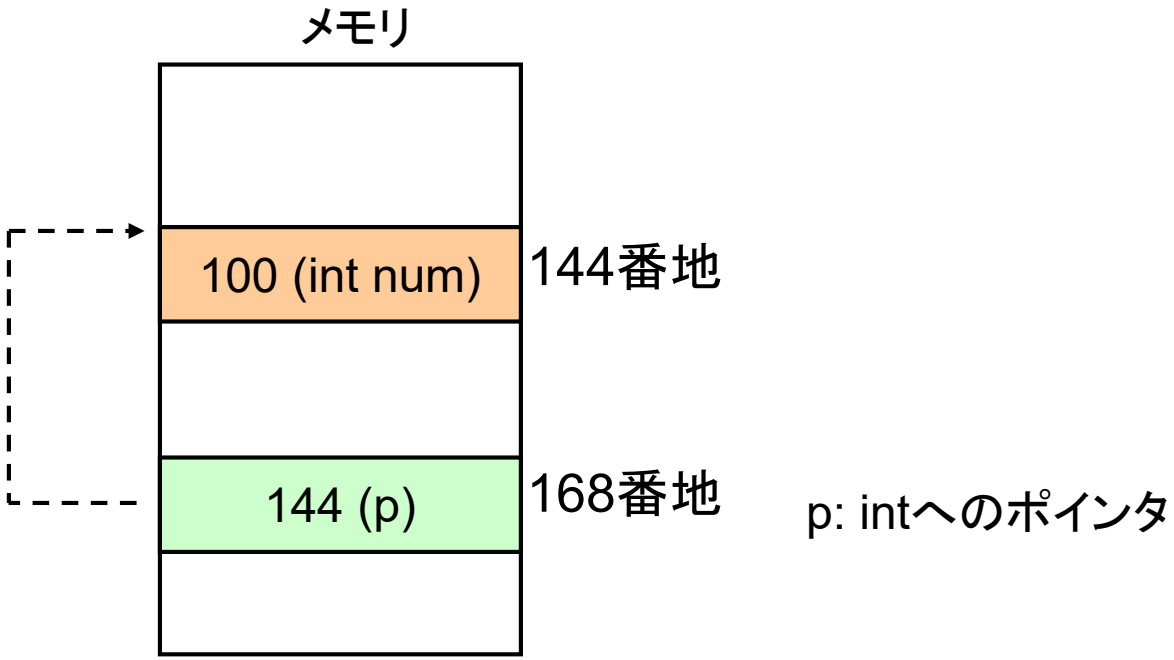


ポインタとは(1/2)

- メモリ上の変数のアドレスを保持する型
  - ポインタ型のサイズは4バイト （32 bit system）
    - 0x00000000 ～ 0xffffffff
  - 例
    - int *p: int型の変数のアドレスを保持する型
    - char *p: char型の変数のアドレスを保持する型
  - XXX型の変数のアドレスを保持するポインタのことを、「XXXへのポインタ」と言う
    - 例
      - int *p: 「intへのポインタ」、
      - char *p:「charへのポインタ」

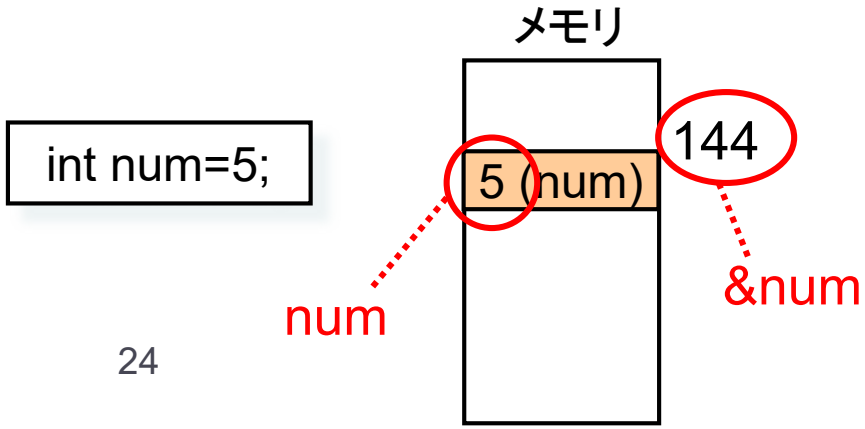
ポインタとは(2/2)

- もちろん、(intやcharのように)ポインタもメモリ上に置かれている
  - 変数のアドレスを保持することによって間接的にアクセスすることができる



アドレスについて

- int,charなどの変数はすべてメモリ上のある場所に存在
  - その場所(住所)のことをアドレスという
- ポインターが参照するためにはその変数のアドレスを渡す必要がある
- 変数のアドレスを取得
  - &num
    - アドレス演算子
    - numのアドレスを取得
    - int型へのポインタを扱えるようになる



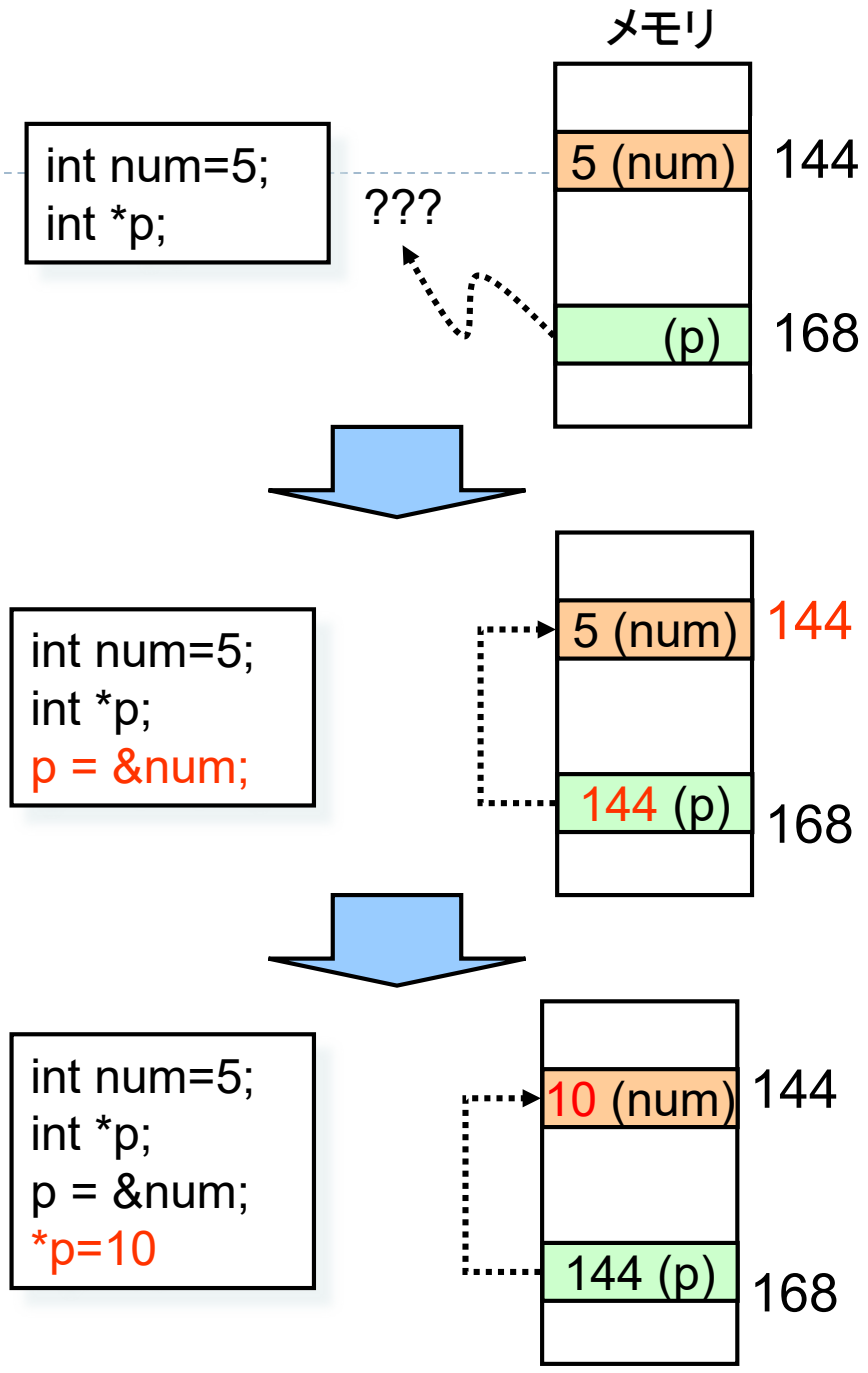
sample10.c

```
#include <stdio.h>
int main()
{
   int nx;
   double dx;
   int vc[3];
   printf("&nx   =%p¥n", &nx);
   printf("&dx   =%p¥n", &dx);
   printf("&vc   =%p¥n", &vc);
   printf("&vc[0]=%p¥n", &vc[0]);
   printf("&vc[1]=%p¥n", &vc[1]);
   printf("&vc[2]=%p¥n", &vc[2]);
}
```

```
&nx    =0x22ccac
&dx    =0x22cca0
&vc    =0x22cc90
&vc[0]=0x22cc90
&vc[1]=0x22cc94
&vc[2]=0x22cc98
```

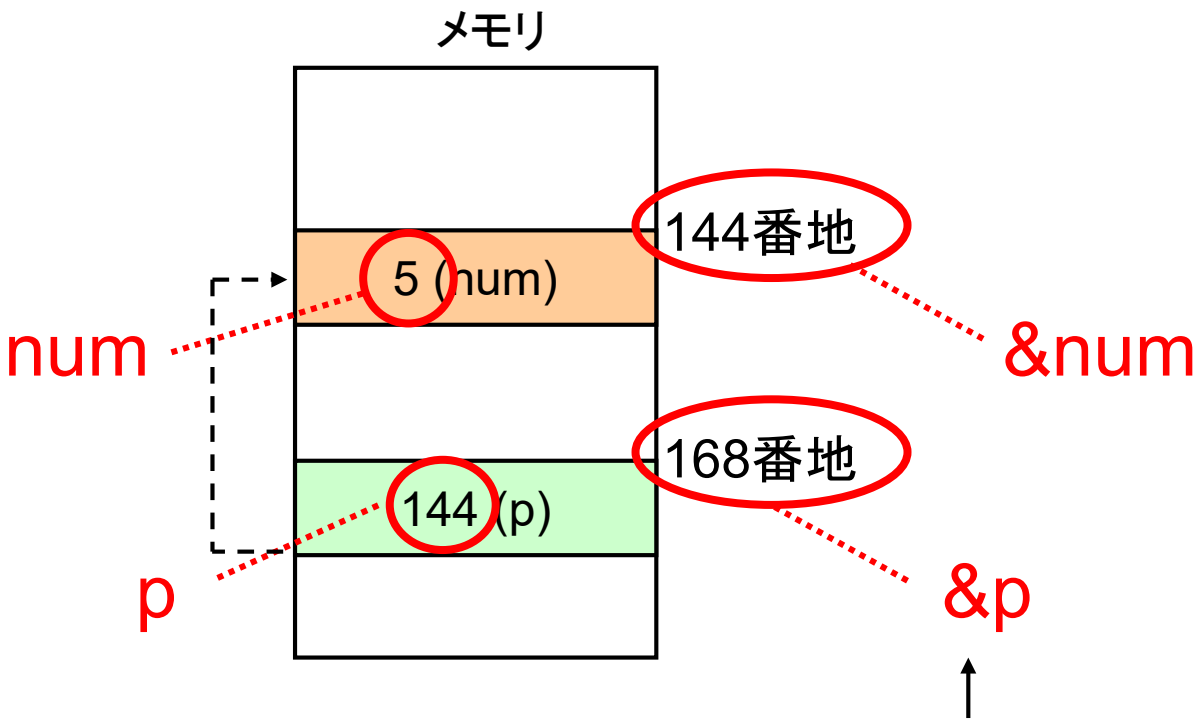
ポインタ型

- ポインタ型の宣言
  - int *p;
    - int型へのポインタ p を宣言
- ポインタ型への代入
  - p = &num
    - pはnumを指す
- ポインタが指す変数の値にアクセス
  - *p
    - 間接演算子
    - *pの値は5
- *p=x; とすることにより、pが指すアドレス上の値をxで置き換えられる
  - 右図の場合、num=10としなくても、numの値は5から10に変わっている



ポインタのまとめ

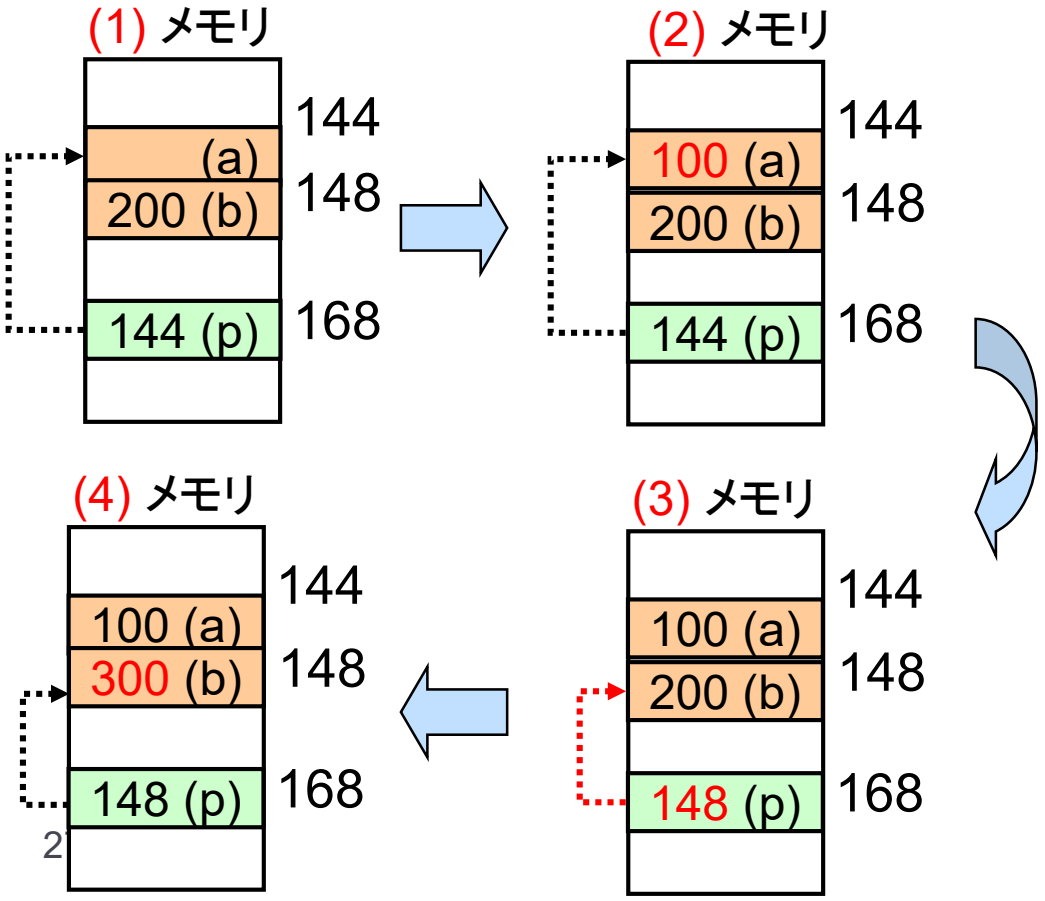
```
int num=5;
int *p;
p = &num;
```



- p = 144なので *pは5

間接演算子による値の代入

- 間接演算子を用いてポインタ経由で値の代入ができる



sample11.c

```
#include <stdio.h>
int main()
{
   int a;
   int b=200;
   int *p;
   p = &a;

   *p = 100;
   printf("*p=%d¥n", *p);
   p = &b;
   printf("*p=%d¥n", *p);
   *p = 300;
   printf("*p=%d¥n", *p);
   printf(“ p=%p¥n", p);
   printf(“&p=%p¥n", &p);
}
```

(1)

(2)

(3)

(4)

ポインターを引数とする関数　(Swap関数)

- 引数にポインタを取る関数の定義

```
void swap(int *a, int *b){
    …….
}
```

- 引数にポインタを取る関数の呼び出し

```
int num1 = 10, num2 = 20;
int *p1, *p2;
swap(&num1, &num2);
p1 = &num1; p2 = &num2;
swap(p1, p2);
```

アドレスを渡す。
どちらでもよい。

- int型引数を2つ取り、値を交換する関数
  - Javaではint型は値渡しなので、実現不可能
  - Cではポインタを引数に取ることにより、実現可能
    - 間接演算子を用いて値を書き換える

Swapのコード

sample12.c

```
#include <stdio.h>
void swap(int *a, int *b)
{
  int tmp;
  tmp = *a;
  *a = *b;
  *b = tmp;
}
int main()
{
  int num1 = 10, num2 = 20;
  int *p1, *p2;

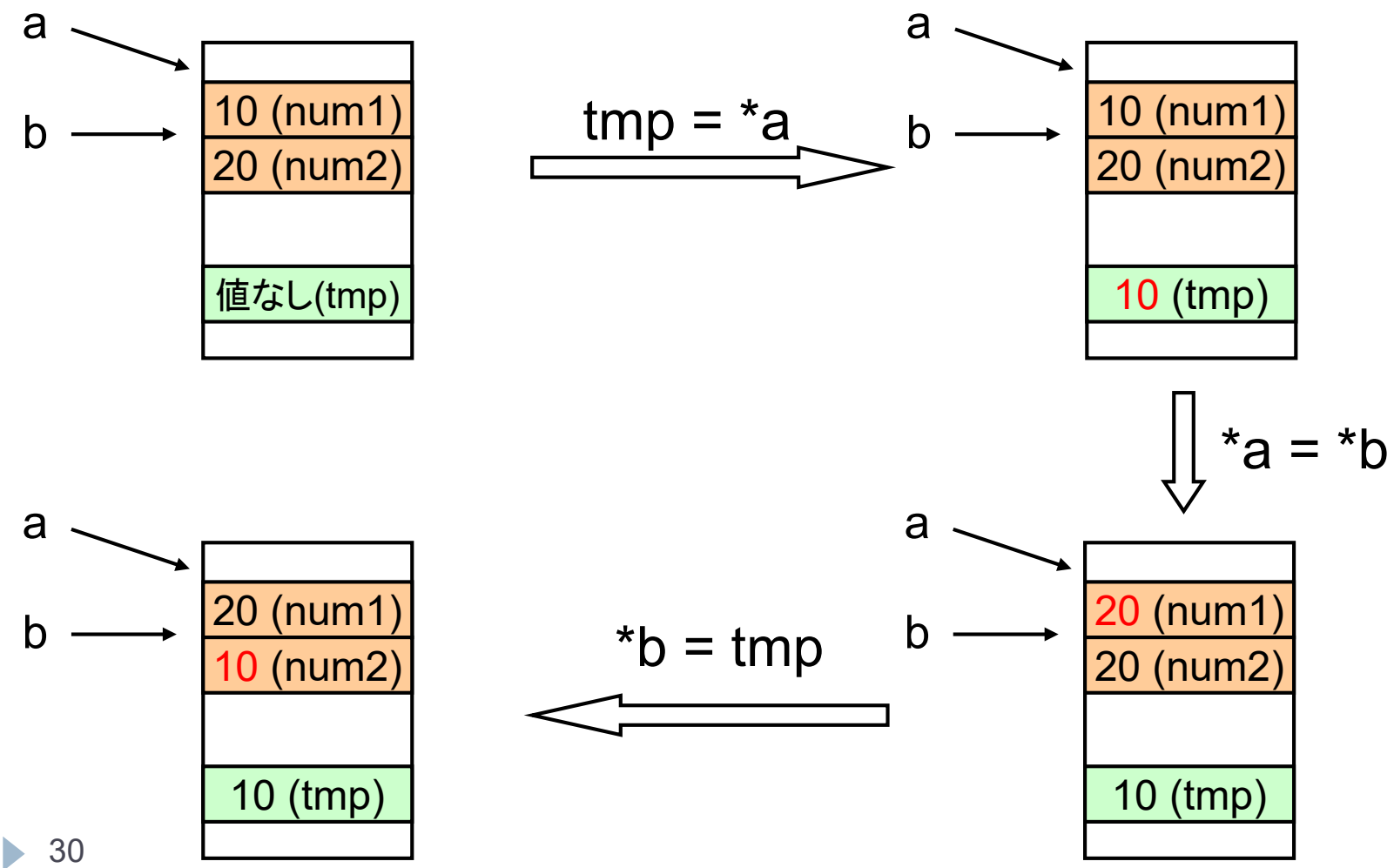
  printf("num1=%d, num2=%d¥n", num1, num2);
  swap(&num1, &num2);
  printf("num1=%d, num2=%d¥n", num1, num2);

  p1 = &num1; p2 = &num2;
  swap(p1, p2);
  printf("num1=%d, num2=%d¥n", num1, num2);
}
```

出力

```
num1=10, num2=20
num1=20, num2=10
num1=10, num2=20
```

Swap関数実行中のメモリ変化



総和と平均を計算する関数

sample13.c

```
#include <stdio.h>
void sum(int f, int l, int *s, int *a)
{
  int sum = 0;
  int i;
  for (i = f; i <= l; i++) {
    sum += i;
  }
  *s = sum;
  *a = sum / (l - f + 1);
}
int main()
{
  int s = 0, a = 0;
  sum(1, 10, &s, &a);
  printf("s=%d, a=%d¥n ", s, a);
  return 0;
}
```

- void sum (int f, int l, int *s, int *a)
  - f: 初項
  - l: 末項
  - s: 総和のポインタ
  - a: 平均のポインタ

出力

```
s=55, a=5
```

NULLポインタ(空ポインタ)

- 「NULL」を代入されたポインタ

```
int *p = NULL;
```

  - 有用なデータを指していない
  - ポインタを返す関数で、戻り値が無いときにも使用
- NULLポインタへのアクセスは実行時エラー

```
int num = 100;
int *p = &num;

printf("%d¥n", *p);
p = NULL;
printf("%d¥n", *p);
```

←― 100を表示

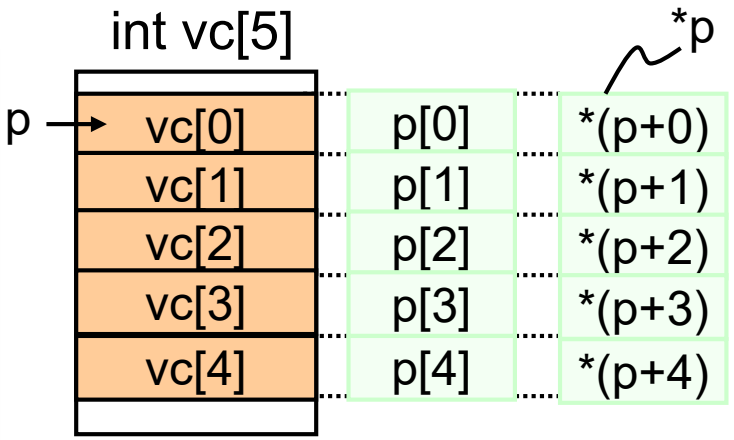
←― 実行時エラー

配列とポインタ (1/2)

- ポインタを用いた配列へのアクセス
  - 先頭要素を指すポインタに i 加えた値は、先頭から i 番目の要素を指すポインタとなる

sample14.c

```
#include <stdio.h>
int main(void)
{
  int i;
  int vc[5] = {1,2,3,4,5};
  int *p = &vc[0];
  for (i = 0; i < 5; i++) {
    printf("vc[%d]=%d, p[%d]=%d, *(p+%d)=%d¥n“,
        i, vc[i],   i, p[i],   i, *(p+i));
  }
  return 0;
}
```




```
vc[0]=1, p[0]=1, *(p+0)=1
vc[1]=2, p[1]=2, *(p+1)=2
vc[2]=3, p[2]=3, *(p+2)=3
vc[3]=4, p[3]=4, *(p+3)=4
vc[4]=5, p[4]=5, *(p+4)=5
```

配列とポインタ (2/2)

- 添字演算子[ ] を伴わない配列名は、配列の先頭要素へのポインタとなる

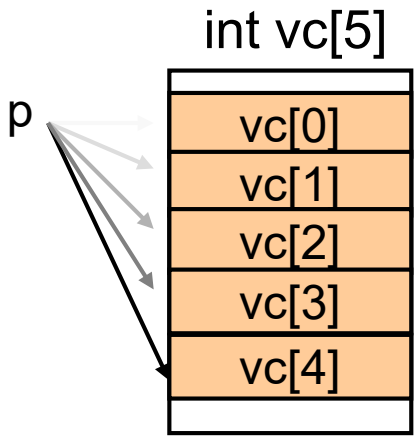
同じ

```
int *p = &vc[0];   ⟺   int *p = vc;
```

- ポインタが指すアドレスを変更しながらのアクセスも可能
  - 注意： 使用後の指すアドレスは変更されたまま

```
for (i = 0; i < 5; i++) {
    p = p + 1; // p++でもよい
    printf("vc[%d]=%d, p[%d]=%d, *(p+%d)=%d¥n“,
        i, vc[i],   i, *p,   i, *p);
 }
```

```
printf("vc[%d]=%d, p[%d]=%d, *(p+%d)=%d¥n“,
        i, vc[i],   i, *p,   i, *(p++));
```



配列を引数に取る関数

- 宣言

```
int func1(int a[], int size);
int func2(int *a, int size);
```

  - 配列を引数として渡すことはできない。
  - コンパイラでは後者として解釈される

- 呼び出し

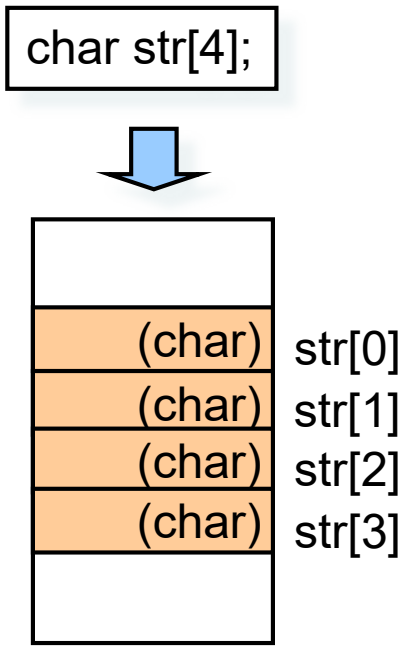
```
int a[] = {1,2,3,4};
func1(a, 4);
func2(a, 4);
```

配列名は先頭要素へのポインタ
（アドレス &a[0]）になる
&a[0]の型は int *

配列とポインタの違い (1/2)

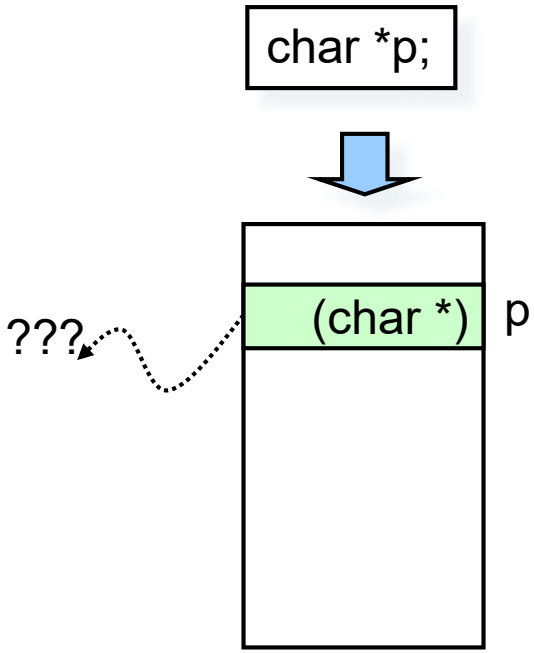
- 宣言時のメモリ配置

配列



charを格納できる
スペースが4つ確保される

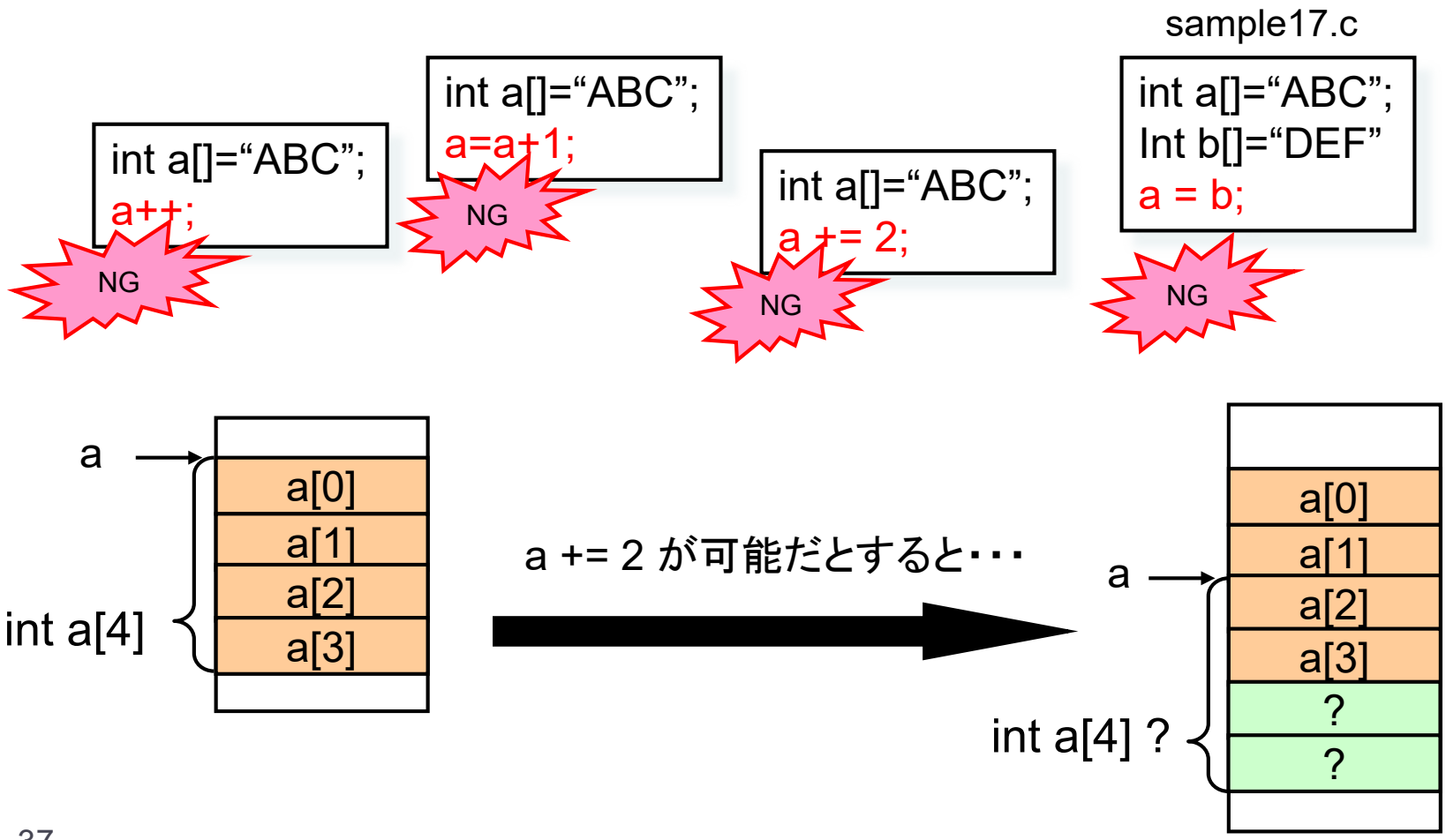
ポインタ



charのアドレスを指す
スペースが1つ確保される

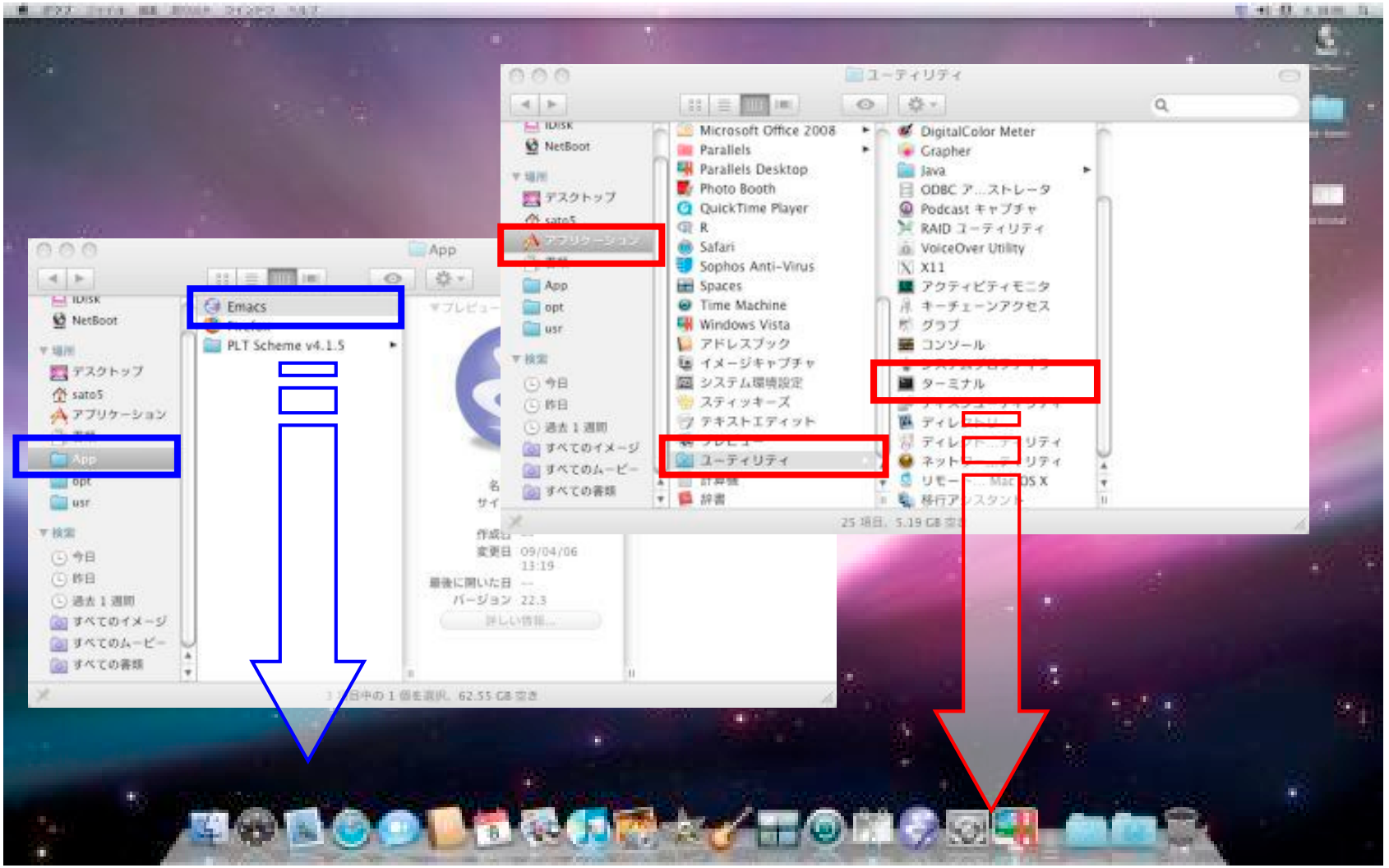
配列とポインタの違い (2/2)

- intやcharと同様、配列のアドレスは変更できない



本日の課題

作業環境準備


App
Emacs
PLT Scheme v4.1.5
ユーティリティ
Microsoft Office 2008
Parallels
Parallels Desktop
Photo Booth
QuickTime Player
R
Safari
Sophos Anti-Virus
Spaces
Time Machine
Windows Vista
アドレスブック
イメージキャプチャ
システム環境設定
スティッキーズ
テキストエディット
ユーティリティ
DigitalColor Meter
Grapher
Java
Podcast キャプチャ
RAID ユーティリティ
VoiceOver Utility
X11
アクティビティモニタ
キーチェーンアクセス
グラブ
コンソール
ターミナル

作業用ディレクトリ

- ターミナル を起動
- 作業ディレクトリ（例：~/Class/compsys/ex01）の作成
  - ```
    $ mkdir –p ~/Class/compsys/ex01
    ```
  - このディレクトリ以下にファイルを作成
- 作業ディレクトリのパーミッションの設定
  - ```
    $ chmod 700 ~/Class
    ```
  - 他人に見られないように
- 作業ディレクトリに移動
  - ```
    $ cd ~/Class/compsys/ex01
    ```
- コンパイル & 実行

```
$ gcc –Wall –o ex01 ex01.c
$ ./ex01
```
```

作業環境準備

- エディタ
  - 好きなものを用いて下さい
    - Atom, Emacs, Vim, CotEditor, Sublime Text, VSCode, Notepad...

- 演習室Mac以外で作業する場合
  - 各自で環境を準備してください
  - Windows
    - Cygwinなど
  - レポート提出前に、演習室Macで動作検証することを推奨します

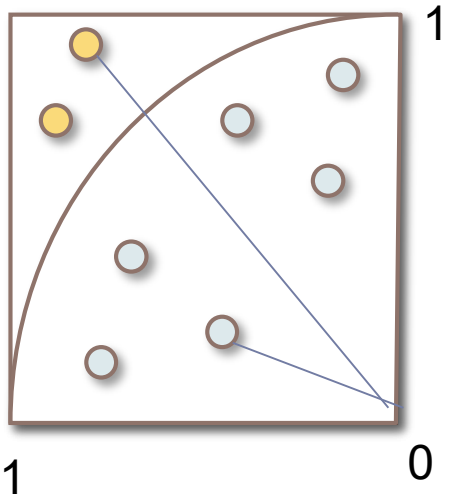
知っておくべきUNIXコマンド

- UNIXコマンドって？
  - LinuxやMacのターミナル上で使えるコマンド
- 知っておくべきコマンド
  - cd
  - ls
  - mkdir
  - cp
  - mv
  - rm
- コマンドの詳細は man [コマンド名] で見られます
  - C言語の関数、特別なファイルの説明なども

課題1： 円周率の計算

- モンテカルロ法を用いて円周率を計算せよ
- モンテカルロ法： 乱数を用いてシミュレーションや数値計算を行う手法

- 円周率の場合
  - [0,1]×[0,1]の正方形に乱数を用いて点を打つ
  - 扇形内部の点の割合 (およそ0.785) から扇の面積を求める。S とする
  - S= pi/4 から pi を計算



- サンプル数が多いほどより高い精度
  - ただし、非常に計算時間が長くなる

課題1

- 疑似乱数の生成

```
#include<stdio.h>
#include<stdlib.h>
int main()
{
    int sai, i;
    for (i = 0; i < 10; i++) {
        sai = rand() % 6 + 1;
        printf("%d¥n", sai);
    }
     return 0;
 }
```

- int monte_carlo (int n, double *pi)
  - n:　サンプル数
  - pi: 円周率
  - 戻り値:　計算完了＝＞0　エラー=> -1
    - エラー例：引数が正しくないとき
- テストするmain関数も書いて提出
  - 正しく動いていることを示す出力をさせること
    - pi 関数前後の pi の内容を表示

課題2：配列ソート

- intの配列を受け取り、その配列をソートする関数を書け
  - void sort_int_array (int *ary, int ary_size)
    - 第1引数 *ary:　検索対象の整数配列
    - 第2引数 ary_size:　配列のサイズ
  - テストするmain関数も書いて提出
    - 正しく動いていることを示す出力をさせること
      - sort_int_array前後の配列の内容を表示

- ソートアルゴリズムは過去の他講義の課題で(C言語以外を含めて)実装したことのないものにすること
  - 例: マージソート・シェルソート・基数ソート
  - 使用したアルゴリズムを明記すること
  - qsort()などの既存のソート関数は使用禁止

課題の補足 (第1回課題)

- 課題1
  - rand() の最大値は RAND_MAX (stdlib.hで定義)
  - double x = rand() / RAND_MAX → x の値は 0
    - 整数同士の除算では1以下の結果は 0 と扱われるため
    - 浮動小数点にキャストする必要あり
- ランダム関数の出力内容を実行毎に変える方法
  - srand((unsigned int)time(NULL)) (#include <time.h> が必要)
    - time(NULL) で現在時刻を取得し、srand で乱数系列のseedを与える
    - unsigned int 型 : 符号なしの整数値

課題提出

- 〆切: 12/13 (金) 23:59
  - OCW-iから提出すること
  - 遅れても（減点しますが）受け付けます。
- 提出物: 以下のファイルを1つのファイルにzip圧縮したもの
  - ドキュメント (txt形式、図や数式が書けなければpdfでも可)
    - 各課題の実行結果
    - プログラムソースの簡単な説明、工夫したところ
    - 感想、質問等（次回以降の授業でフィードバックします）
  - プログラムソース (課題1, 2)
    - テスト用のmain関数も含む (コンパイルできて正しく実行できること)
- 全てのファイル名は半角英数字(日本語不可)でお願いします
  - レポートのファイルを含む、文字化けの原因になるので

課題の補足 (課題全体に関して)

- 新しいC言語規格でのみ許される記法について
  - 演習室のMacで普通にコンパイルできるのであれば使用可
  - コンパイル時に特殊な指定が必要であればその旨レポートに記載すること
- プログラム・関数の入出力仕様について
  - 課題で指定されていない箇所について、例えばコマンドライン引数に応じて入力サイズを変えるなどの工夫は適宜行ってください(むしろ推奨します)
  - 関数の型が指定されている場合でも、エラーを返すために返り値の仕様を変更することは差し支えありません