プログラミング基礎演習

第7回　ポインタと動的メモリ確保

担当：矢野　史朗（10-413室）
syano@cc.tuat.ac.jp

１．前回のおさらい Linux(shell script)

Shell Script：主に下記の組み合わせ

- 変数
- コマンド置換
- フロー制御(for, if, elif, while, case)
- 関数
- シェルコマンド
- プロセス操作(開始，確認，停止)

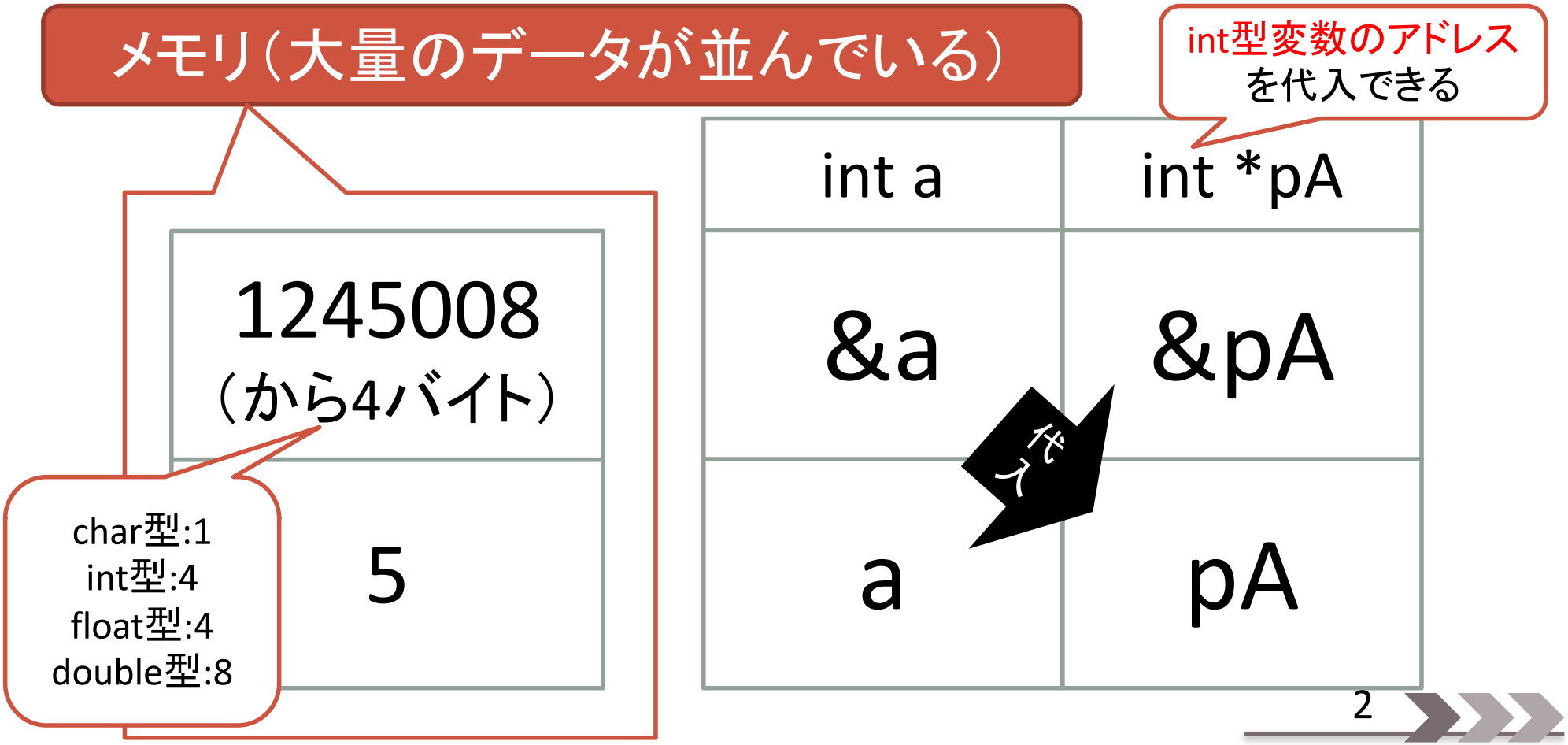
今回は，変数，コマンド置換，for文まで

プロセス：動作中のプログラム

1．前回のおさらい　C言語

ポインタ型

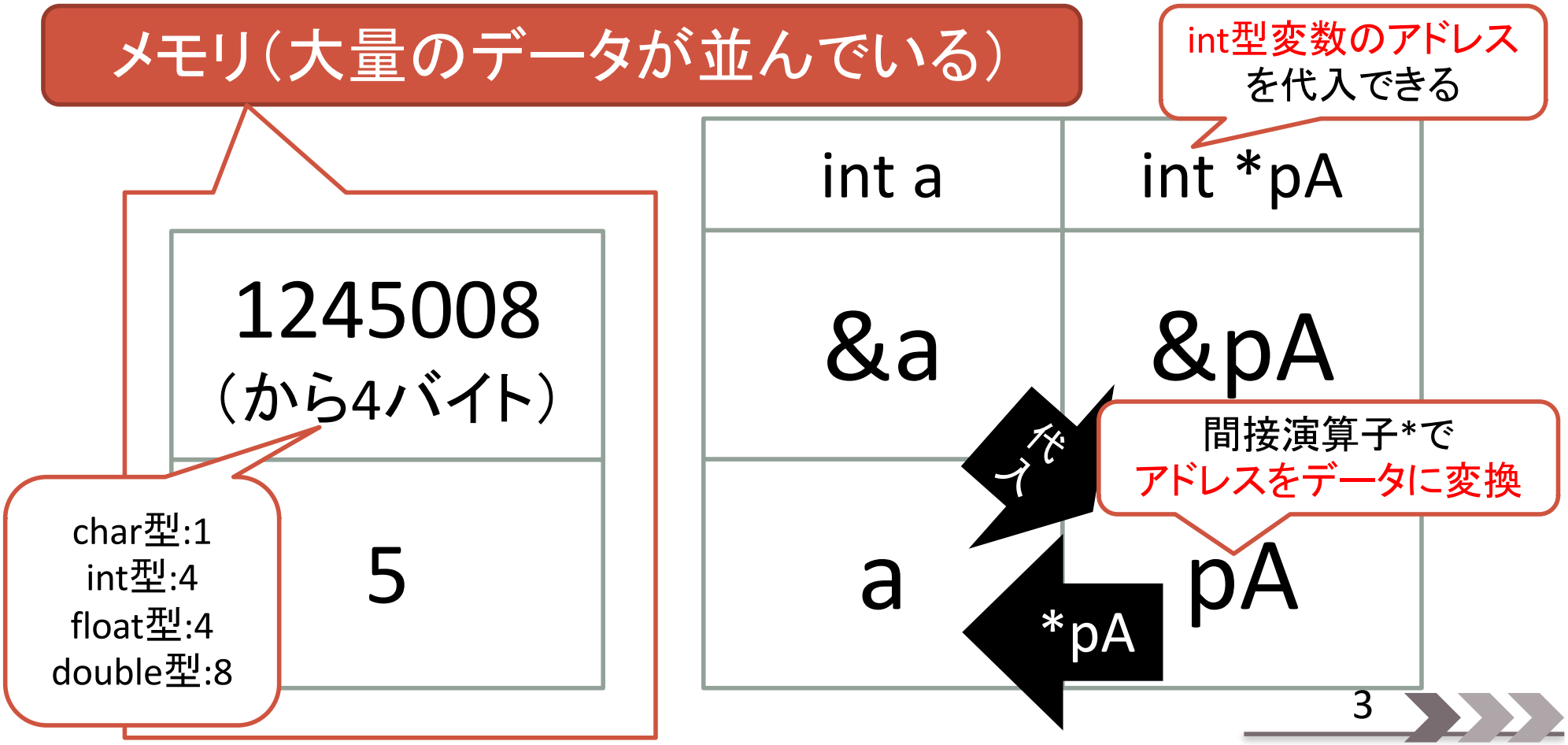
- アドレスを保存するための変数の型の1つ
- メモリ上のアドレスを直接指定したデータのやりとりができる



1．前回のおさらい　C言語

ポインタ型

- アドレスを保存するための変数の型の1つ
- メモリ上のアドレスを直接指定したデータのやりとりができる



1．前回のおさらい　C言語

配列とアドレス

```
int var[3]={1, 2, 3};

// varは，var[0]の
// アドレスを指す

printf("%p¥n",&var[0]);
printf("%p¥n",var);
```

配列名:=先頭の値のアドレスを指すポインタ

```
void func2(int* q){…}

int main(){
        int b[3]={1,2,3};
        func2(b);
}
```

今回の内容

Linux

- シェルスクリプトの続き

C言語

- 前回内容（ポインタ）の補足
- 動的メモリ確保

2. Linux：シェルスクリプト

- # 前回に引き続きC言語の習得で余裕がない人は
  シェルスクリプトは発展的話題と捉えて大丈夫です.

システムに影響するスクリプト，実験的なスクリプトは，仮想環境等の自身の環境で実行してください

学生側のシステム環境が統一されていないことで，演習に支障が出る場合があるので.

2. Linux：シェルスクリプト（if-else-fi文）

If-else-fi構文

```
if command-list
then
        command
else
        command
fi
```

*command-list*部分：　頻繁に用いるのはtestコマンド

test –r *file*　　*file*が読取り可なら真
test –w *file*　　*file*が書込み可なら真
test –x *file*　　*file*が実行可なら真

test *str1* = *str2*　　*str1*と*str2*が「同じ」なら真
test *str1* != *str2*　　*str1*と*str2*が「同じでない」なら真

数値の比較もある．“man test”をしてみよう．

2. Linux：シェルスクリプト（while文）

for構文

```
while command-list
do
        command
done
```

例えば次のようなコードをwhile内に設けることで
while ループから抜けることもできる

```
if command-list
then
        break
fi
```

2. Linux：シェルスクリプト

Linux/シェルスクリプトをもう少し使いこなすための方針

- パスの設定
  - 作成したスクリプトや実行ファイルを含むdirを システムがすぐ検索・発見できるよう教えること
  - ~/.bash_profile　や .bash_rc
- 実行中のプロセスの確認
  - ps –axl
- 接続したデバイスの認識
  - /dev/
  - FHS(Filesystem Hierarchy Standard)が標準的な ディレクトリ構造をルール化（規格）

３．ポインタ

アドレスを保存する変数

```
int   i;
int  *p;

p = &i;
*p = 10;
printf("i = %d \n",i);
```

変数の前に「＊」をつけると、その型のポインタという意味になる

変数名	アドレス	内容
i	100	???
p	104	???
….	….	….
….	….	….

3. ポインタ

アドレスを保存する変数

```
int   i;
int  *p;

p = &i;
*p = 10;
printf("i = %d \n",i);
```

アドレス演算子「&」を使い、変数のアドレスを得る

変数名	アドレス	内容
i	100	???
p	104	100
….	….	….
….	….	….

3. ポインタ

アドレスを保存する変数

```
int   i;
int  *p;

p = &i;
*p = 10;
printf("i = %d \n",i);
```

演算子「＊」を使うと、「指し示す先に」対して演算を行う

```
i = 10
```

変数名	アドレス	内容
i	100	10
p	104	100
….	….	….
….	….	….

4. ポインタと配列

プロ序演習
第11回資料参照

配列はメモリ上に連続して置かれる

```
#include <stdio.h>

void main()
{
        int I[5] = {1,2,3,4,5};
}
```

メモリ上の様子

アドレス	変数名	データ
1244984	I[0]	1
1244988	I[1]	2
1244992	I[2]	3
1244996	I[3]	4
1245000	I[4]	5
1245004	(未使用)	??????

4. ポインタと配列

プロ序演習
第11回資料参照

配列はメモリ上に連続して置かれる

```
#include <stdio.h>

void main()
{
        int I[5] = {1,2,3,4,5};
        int *p = &I[0];
        int k;
        for(k=0; k<5; k++,p++)
                printf("%d,",*p);
}
```

配列の先頭のアドレスを
ポインタ「p」に代入

メモリ上の様子

アドレス	変数名	データ
1244984	I[0]	1
1244988	I[1]	2
1244992	I[2]	3
1244996	I[3]	4
1245000	I[4]	5
1245004	p	1244984

アドレスを加算しながら、5個分の値を表示
ポインタ変数に1を加える ⇒ 次の要素のアドレスを得る（超重要！）

4. ポインタと配列

プロ序演習
第11回資料参照

配列はメモリ上に連続して置かれる

```
#include <stdio.h>

void main()
{
        int I[5] = {1,2,3,4,5};
        int *p = &I[0];
        int k;
        for(k=0; k<5; k++,p++)
                printf("%d,",*p);
}
```

メモリ上の様子

アドレス	変数名	データ
1244984	I[0]	1
1244988	I[1]	2
1244992	I[2]	3
1244996	I[3]	4
1245000	I[4]	5
1245004	p	1244984

実行結果

1,2,3,4,5,

配列は必ず連続しておくことが定められているので、必ずこの結果になる

4. ポインタと配列

配列要素のアドレスの多様な呼び出し

配列名は先頭要素のアドレスとなっている

アドレス	&を使う	（&data[p]+〇）	（&data[p+〇]）	data+〇
1244984	&data[0]	&data[0]	&data[0]	data
1244988	&data[1]	&data[0]+1	&data[0+1]	data+1
1244992	&data[2]	&data[0]+2	&data[0+2]	data+2
1244996	&data[3]	&data[0]+3	&data[0+3]	data+3
1245000	&data[4]	&data[0]+4	&data[0+4]	data+4

4. ポインタと配列

ポインタの利点①

関数等の引数として用いることで、特に大規模な配列の場合，
メモリの大幅な節約になる

使い方の復習

■値を入れ替える関数

```
void swap(int *x, int *y)
{
        int tmp = *x;
        *x = *y;
        *y = tmp;
}
```

ポインタ型変数に渡せる値はアドレス
（＝参照渡し　＝ポインタ渡し）
呼び出し元では「swap(&x, &y);」のように
アドレスを引数にして関数を呼び出す

演算子「＊」を使うと、そのポインタ型変
数が示すアドレスにある値を表現できる
（宣言で使う「＊」と混同しないように）

4. ポインタと配列

ポインタの利点②

配列（メモリ上に連続した領域をもつデータ構造）との相性が良い

■あるプログラムの出力例の一部

I[0]のアドレスは1244984で値は1
I[1]のアドレスは1244988で値は2
I[2]のアドレスは1244992で値は3
I[3]のアドレスは1244996で値は4
I[4]のアドレスは1245000で値は5
F[0]のアドレスは1244964で値は6.000
F[1]のアドレスは1244968で値は7.000
F[2]のアドレスは1244972で値は8.000
F[3]のアドレスは1244976で値は9.000
F[4]のアドレスは1244980で値は1.000

どちらもアドレスが
4ずつ増えている
ことに注目

5. 動的メモリ確保

メモリを確保する関数

C言語ではコンパイル時に配列要素数が固定済みとなっている必要がある(※)

標準入力による要素数の指定

```
int n;
scanf("%d", &n);
double var[n];
```

可変長配列は規格C99で使えるようになったが，C11でオプションに変更されたコンパイラ側で対処してもらえる場合もあるが，動的メモリ確保は知っておくべき

5. 動的メモリ確保

メモリを確保する関数

C言語ではコンパイル時に配列要素数が固定済みとなっている必要がある(※)

✖ 標準入力による要素数の指定

```
int n;
scanf("%d", &n);
double var[n];
```

Cのライブラリには、アプリケーションの実行時にメモリを動的に確保する仕組みが用意されている

- 変数や配列の宣言の仕方の一つと思えば良い
- 本演習では、実際にその宣言の仕方を用いてみる
- メモリの使用状況はプログラマが管理しなければならない

5. 動的メモリ確保

メモリを確保する関数

```
void* malloc(size_t  size)
```

確保したメモリの
先頭アドレスを返す

確保したいメモリの
大きさ（byte数）

5. 動的メモリ確保

メモリを確保する関数

void* malloc(size_t size)

```
int *pData, i, size;

size = scanf("%d", &size);

pData = (int*)malloc(sizeof(int)*10);

for(i = 0; i < 10; i++) pData[i] = i;

free(pData);

pData[5] = 0;
```

5. 動的メモリ確保

メモリを確保する関数

void* malloc(size_t size)

```
int *pData, i, size;

size = scanf("%d", &size);

pData = (int*)malloc(sizeof(int

for(i = 0; i < 10; i++) pDa

free(pData);

pData[5] = 0;
```

配列数を標準入力.
コンパイル時に配列数を固定しないで
済んでいる点に注意.

5. 動的メモリ確保

メモリを確保する関数

void* malloc(size_t size)

```
int *pData, i, size;

size = scanf("%d", &size);

pData = (int*)malloc(sizeof(int)*10);

for(i=0;i<5;i++) pData[i]

pData[5] = 0;
```

汎用ポインタ(void*)を
目的の型のポインタへ
変換する

sizeof演算子を用いることで、
作成したい型の大きさがわかる

5. 動的メモリ確保

メモリを確保する関数

void* malloc(size_t size)

```
int *pData, i, size;

size = scanf("%d", &size);

pData = (int*)malloc(sizeof(int)*10);

for(i = 0; i < 10; i++) pData[i] = i;

free(pData);

pData[5] = 0;
```

確保後は、通常の配列と同じように扱うことができる

確保した領域は、必ずfree関数を用いて解放すること

5. 動的メモリ確保

メモリを確保する関数

void* malloc(size_t size)

```
int *pData, i, size;

size = scanf("%d", &size);

pData = (int*)malloc(sizeof(int)*10);

for(i = 0; i < 10; i++) pData[i] = i;

free(pData);

pData[5] = 0;
```

解放後に確保「していた」領域にアクセスすることは危険

5. 動的メモリ確保

メモリを確保する関数

void* malloc(size_t size)

確保されたメモリ領域をヒープ（heap）と呼ぶ
ヒープに確保された配列を動的配列と呼ぶ
メモリ不足でヒープが確保出来ない時，mallocはNULLを返す
メモリが少ない状況では特に返り値のチェックを怠らないこと

```
int *pData, i;

pData = (int*)malloc(sizeof(int)*10);

if (pData == NULL) abort();

for(i = 0; i < 10; i++) pData[i] = i;

free(pData);
```

6. 本日の課題

■文字列を逆順にする関数の実装

入力として各行に，整数nと，長さnの文字列sが与えられるので，
文字列sを逆順に出力するようにせよ.
なお，入力は複数行にわたるので，最後まで処理せよ.

6. 本日の課題

■入力の形式

入力は以下の通りである.

n_1 s_1
n_2 s_2
⋮ ⋮

整数n_iの大きさは1以上であること以外不明であるが,
実行に支障のない範囲で指定される.
長さn_iの文字列s_iは, 'a'-'z'と'-'からなる.

6. 本日の課題

■出力の形式

r_1
r_2
⋮

以下の通りに出力すること.

文字列s_iを逆順にした，文字列r_iを出力する.
なお，各r_iの後には改行が入る(テンプレートのprintf部分を参照).

6. 本日の課題

■入力例・出力例

例.

入力:

```
5 abcde
20 how-is-your-progress
1 z
```

出力:

```
edcba
ssergorp-ruoy-si-woh
z
```

入力では各行に，文字列の長さ(文字数)5,20,1と文字列"abcde","how-is-your-progress","z"がスペース区切りで入力されている.
出力では，入力の文字列を逆順にした，文字列"edcba"，"ssergorp-ruoy-si-woh","z"が出力されている.
なお，出力の"z"の後には改行が入っている(テンプレートのprintf部分を参照).

6. プログラムの作成およびコンパイル

■プログラムの作成（第7回講義演習課題）

- ソースコードのファイル名は次の通りにする（すべて小文字）
  （EDENアカウント）_dmemory.c
- 提出用ディレクトリ：
  /home/syano/prokiso/
- コマンドでコピーを行う場合：
  cp （EDENアカウント）_dmemory.c /home/syano/prokiso/
- コピーする前にアクセス権を変更しておくこと

[sxx@xx~/prokiso] chmod 755 （EDENアカウント）_dmemory.c

- 提出期限： 11月28日（月） AM10：30（講義開始時）

7. 次回について

第8回（11月28日）
「ポインタと文字列」

Further reading

C言語ポインタ完全制覇