

# 情報システムプログラミングⅡ (**12**回目)

2024年7月12日 (金)

3～4限

# 授業内容

- 講義内容（教科書の378～394ページ） +  $\alpha$ 
  - メモリを扱う標準関数
  - ヒープの利用
  - Pythonにおけるメモリの扱い
- 演習課題

# メモリを扱う標準関数

## ■memcpy関数

- 指定したメモリ領域をコピーする
- memcpy関数を利用するための構文
  - string.hの読み込み（`#include <string.h>`）が必要

```
void* memcpy(void* addr1, const void* addr2, size_t len);
```

addr1 : コピー先の先頭アドレス  
addr2 : コピー元の先頭アドレス  
len : コピーするバイト数  
戻り値 : addr1 と同様

size\_t型は0以上の整数を入れるための型であり、単に0以上の整数を指定するものと認識すればよい

constは後続く値を変更不可とする修飾子  
(必須ではない)

# メモリを扱う標準関数

## ■memcpy関数

```
int main(void)
{
    int a[4] = {19, 20, 29, 29};
    int b[4];
    memcpy(b, a, 16);
    printf("配列aの2つ目の要素は：%d、%p番地に格納\n", a[1],
           &a[1]);
    printf("配列bの2つ目の要素は：%d、%p番地に格納\n", b[1],
           &b[1]);

    return 0;
}
```

16 バイト分をコピー

基本的にchar型は1バイト、  
int型とunsigned int型は4バイト、  
double型は8バイト  
(sizeof関数で確認できる)

メモリ上の値をコピーするのみで、  
アドレスまではコピーされない  
(浅いコピーと深いコピーの  
違いに注意)

### 実行結果

配列aの2つ目の要素は：20、4021255490番地に格納  
配列bの2つ目の要素は：20、4021255470番地に格納

# メモリを扱う標準関数

## ■memcmp関数

- 指定したメモリ領域を比較する
- memcmp関数を利用するための構文
  - string.hの読み込み（`#include <string.h>`）が必要

```
int memcmp(const void* addr1, const void* addr2, size_t len);
```

addr1 : 比較先の先頭アドレス

addr2 : 比較元の先頭アドレス

len : 比較するバイト数

戻り値 : 2つのメモリ領域が同じ内容ならば0

比較先が比較元よりも値が大きければ正の値,  
値が小さければ負の値が返される



# メモリを扱う標準関数

## ■memcmp関数

```
int main(void)
{
    int a[4] = {19, 20, 29, 29};
    int b[4] = {19, 20, 29, 29};
    int r = memcmp(a, b, 16);
    if (r == 0) {
        printf("memcmpで比較した結果、等しいです\n");
    }
    if (a == b) {
        printf("==演算子で比較した結果、等しいです\n");
    }

    return 0;
}
```

16 バイト分を比較

「==」だと同じアドレスにある  
同じ値かを判定することになるが  
(等値判定), **memcmp**関数だと  
異なるアドレスでも同じ値かを  
判定できる (等価判定)

# メモリを扱う標準関数

## ■memset関数

- 指定したメモリ領域を初期化する（指定した値で埋める）
- memset関数を利用するための構文
  - string.hの読み込み（`#include <string.h>`）が必要

```
void* memset(void* addr, int val, size_t len);
```

addr : 書き込み先の先頭アドレス

val : 書き込むデータ (0 ~ 255)

len : 書き込むバイト数

戻り値 : addr と同じ

```
int gems[10];
```

```
memset(gems, 0, 40);
```

40 バイト分の領域を 0 で埋める



# ヒープの利用

## ■メモリ上の値の寿命（メモリ領域が確保される期間）

- 関数内で定義した変数や配列が利用したメモリ領域（スタック領域）は，関数の処理が終了した後に解放される

```
int* readyAges(void)
{
    int ages[4]; // 要素数4のint配列を作成（仮に1000～1015番地）
    return ages; // 先頭アドレス（1000）を返す
}

int main(void)
{
    int* a = readyAges(); // 配列作成を依頼
    a[0] = 19;
    return 0;
}
```

関数終了に伴い 1000 ～ 1015 番地の確保が解除される

1000 ～ 1003 番地に 19 を格納してしまう！



# ヒープの利用

## ■ malloc関数

- ヒープ領域（動的な確保や解放が可能なメモリ領域）を利用できる関数で，確保したメモリ領域は明示的に開放するまで利用できる
- malloc関数を利用するための構文
  - **stdlib.hの読み込み（#include <stdlib.h>）が必要**

```
void* malloc (size_t len);
```

len     : 確保したいバイト数

戻り値 : 確保されたメモリ領域の先頭アドレス

確保に失敗した場合は NULL

（必要に応じて「int\*」などにキャストして使う）

確保するメモリ領域の既存の値は不明だが，メモリ領域を0で埋めた上で確保する**calloc関数**もある

# ヒープの利用

## ■ free関数

- malloc関数またはcalloc関数で確保したヒープ領域のメモリ領域を明示的に解放する関数
- free関数を利用するための構文
  - **stdlib.hの読み込み（#include <stdlib.h>）が必要**

```
void free (void* p);
```

p：過去に malloc 関数で確保したメモリ領域の先頭アドレス

# ヒープの利用

## ■ malloc関数とfree関数の利用例

各型に対応したポインタに  
アドレスを格納する場合、  
キャスト（型変換）が必要

```
int* readyAges(void)
{
    int* ages = (int*) malloc(16);
    return ages;
} // 関数が終了しても、ヒープの9000~9015番地は解放されない

int main(void)
{
    int* a = readyAges(); // 配列作成を依頼
    if (a == NULL) {
        printf("ヒープ確保に失敗しました\n");
    } else {
        a[0] = 19; // 9000~9003番地に19を格納
        free(a);
    }
    return 0;
}
```

ヒープに16バイト確保(仮に9000~9015番地)

使用済みのヒープ領域を解放

確保に失敗した場合は  
NULLが返されるので、  
NULLと比較することで  
エラー処理をしている



# Pythonにおけるメモリの扱い

## ■ ガベージコレクション

- 不要となったメモリ領域を自動的に解放する機能
- Pythonなど多くの高級言語が採用しているがC言語にはない
  - Pythonでは基本的にメモリについて考慮しなくてよい

## ■ Pythonにおけるメモリ上で値が格納されている場所の確認

- アドレスそのものではないが、`id`関数によりオブジェクト（型を持つデータの実体）の識別子を取得できる

```
a = 1
```

```
print(id(a))
```

「1」の値を示す識別子が  
表示される

# Pythonにおけるメモリの扱い

- 値が同じだと異なる変数でも同じ識別子が割り当てられる

```
a = 1  
b = 1  
  
print(id(a))  
print(id(b))
```

「1」の値を示す同じ  
識別子が表示される

- 異なる値を代入すると異なる識別子が割り当てられる

```
a = 1  
print(id(a))  
  
a = 2  
print(id(a))
```

異なる識別子が  
表示される

# Pythonにおけるメモリの扱い

## ■ リストや配列を代入すると同じ識別子が割り当てられる

```
a = [1, 2, 3]
b = a

print(id(a))
print(id(b))
```

同じ識別子が表示され、  
一方で値を変更するともう一方から  
取得できる値も変わる

## ■ copy関数を利用すると明示的に異なる識別子が割り当てられる

```
from copy import copy

a = [1, 2, 3]
b = copy(a)

print(id(a))
print(id(b))
```

異なる識別子が表示され、  
一方で値を変更してももう一方から  
取得できる値は変わらない