情報システムプログラミングⅡ（11回目）

2024年7月3日（水）

3～4限

授業内容

- 講義内容（教科書の346～377ページ＋α）
  - ポインタを使うメリット
  - 不可欠なポインタの利用
  - 配列とポインタに関する構文
  - メモリにアクセスする手段と注意点
- 演習課題

ポインタを使うメリット

■メモリの節約

- メモリの容量が限られる状況で有効

```
typedef struct {
  String name;
  int hp;
  int attack;
   :
} Monster;
```

実際には 100 近いメンバを持つはず

```
int main(void)
{
  Monster suzaku = {"朱雀", 100, 80, …};
  printMonsterSummary(suzaku);
  return 0;
}
```

ここに約 100 項目が並ぶ

アロー演算子（->）も利用可
「(*m).name」と
「m->name」は同義

```
void printMonsterSummary(Monster* m)
{
  printf("モンスター %s (HP= %d)", (*m).name, (*m).hp);
}

int main(void)
{
  Monster suzaku = {"朱雀", 100, 80, …};
  printMonsterSummary(&suzaku);
  return 0;
}
```

指定アドレスにある情報にアクセス

suzaku のアドレスを渡す

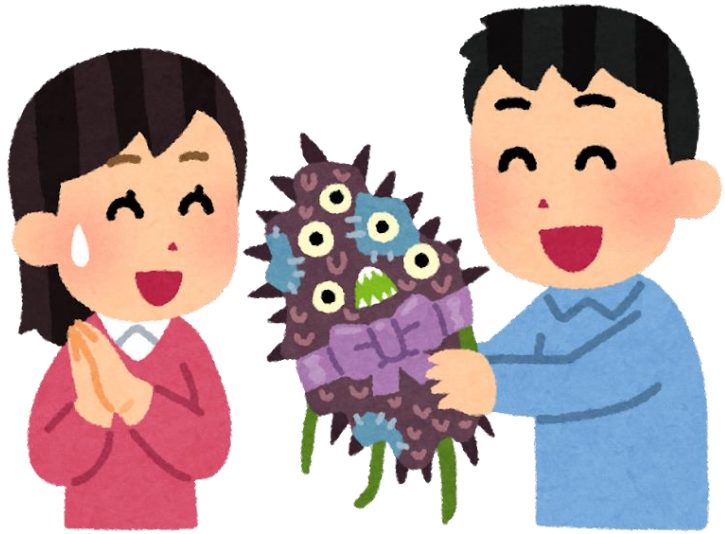
不可欠なポインタの利用

■値渡し

- 関数の引数として値を渡すこと
- 関数内で実引数の値を変更できない

■参照渡し（ポインタ渡し）

- 関数の引数としてアドレスを渡すこと
- 関数内で実引数の値を変更できる



不可欠なポインタの利用

■値渡しと参照渡しの例

- 値渡しでは実引数の値が変わらない
- 参照渡しだと実引数の値が変わる

```
void funcA(int x)      // int型変数を受け取る関数
{
  x = 100;             // 変数を書き換える
}

void funcB(int x[3])   // int型配列を受け取る関数
{
  x[0] = 100;          // 配列の先頭要素を書き換える
}

int main(void)
{
  int a = 5;           // int型変数を宣言(初期値は5)
  int b[3];            // 配列を宣言
  b[0] = 5;            // int型配列の先頭要素を5で初期化

  funcA(a);
  funcB(b);

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

実行結果

```
a=5, b[0]=100
```

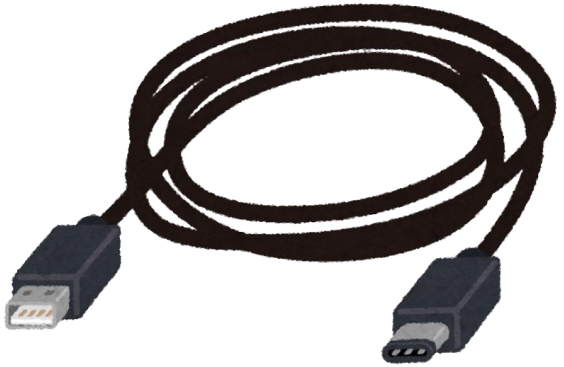
配列とポインタに関する構文

■関数の仮引数や戻り値に配列型を指定した場合

- 自動的にポインタ型に変換される（ポインタ型と見なされる）
  - ➢ 配列名はポインタ変数として扱うことができる

① void funcB(int x[3])
② void funcB(int x[])
③ void funcB(int* x)

すべて③だと見なされる



配列とポインタに関する構文

■関数の実引数に配列名を指定した場合

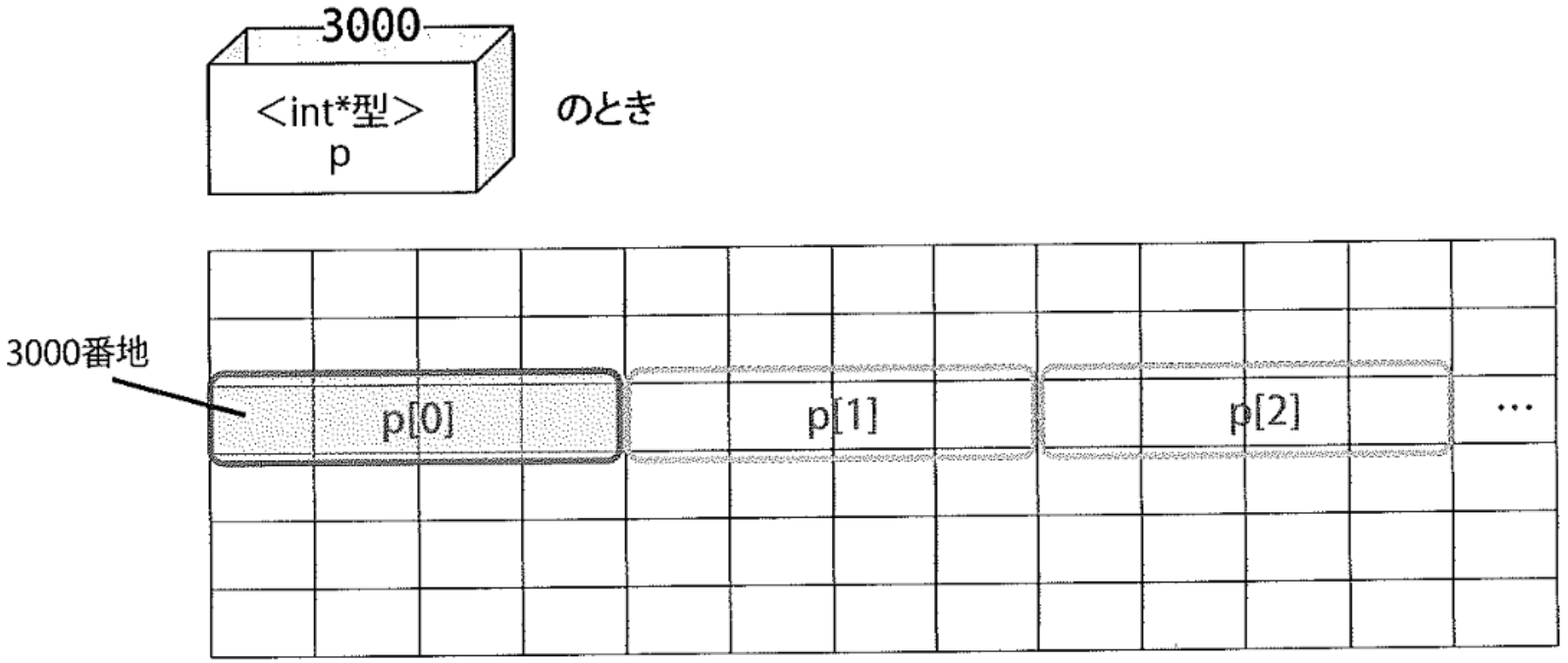
- 自動的に配列の先頭要素の位置を示すアドレスに変換される
- 以下はどちらも配列bの先頭要素のアドレスをfuncB関数に渡している（基本的に上の書き方でよい）

**bと&b[0],
b[0]と*bは同義**

メモリにアクセスする手段と注意点

■添字演算子はポインタ変数にも利用できる

- ポインタ変数に格納されているアドレスを基準として，その前後にアクセスできる



メモリにアクセスする手段と注意点

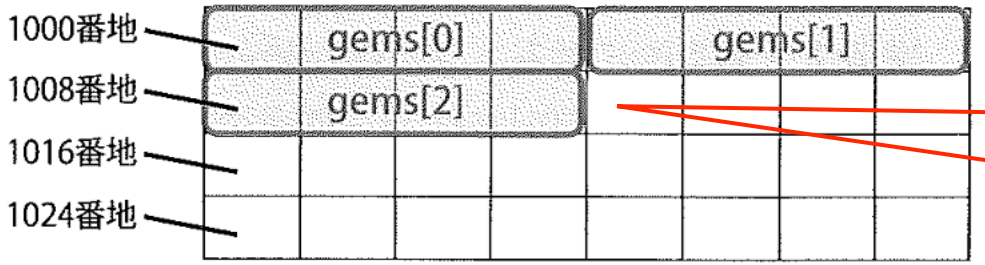
■ポインタ演算

- ポインタ変数（配列名）に加算や減算を行うこと
  - ➢ 型に対応した単位でアドレスが増減する
- 添字演算子とポインタ演算には以下の対応関係がある

ポインタ変数 p があるとき、
p [0]　と　*p　は同じ意味になる。
p[整数]　と　*(p+ 整数)　も同じ意味になる。
※ただし p は void* 型以外であること。

gems[0]と*gems,
gems[1]と*(gems+1),
gems[2]と*(gems+2)は同義

int gems[3];



(gems+2)が指す
アドレスは,
1002ではなく1008

メモリにアクセスする手段と注意点

■添字演算子とポインタ演算の比較

引数を配列とする場合

添字演算子

```
double avg(double a[ ])
{
   double sum = 0.0;

   for(int i=0; i<5; i++) {
      sum += a[i];
   }

   return sum / 5;
}
```

ポインタ演算

```
double avg(double a[ ])
{
   double sum = 0.0;

   for(int i=0; i<5; i++) {
      sum += *(a+i);
   }

   return sum / 5;
}
```

引数をポインタとする場合

ポインタ演算

```
double avg(double* a)
{
   double sum = 0.0;

   for(int i=0; i<5; i++) {
      sum += *(a+i);
   }

   return sum / 5;
}
```

添字演算子

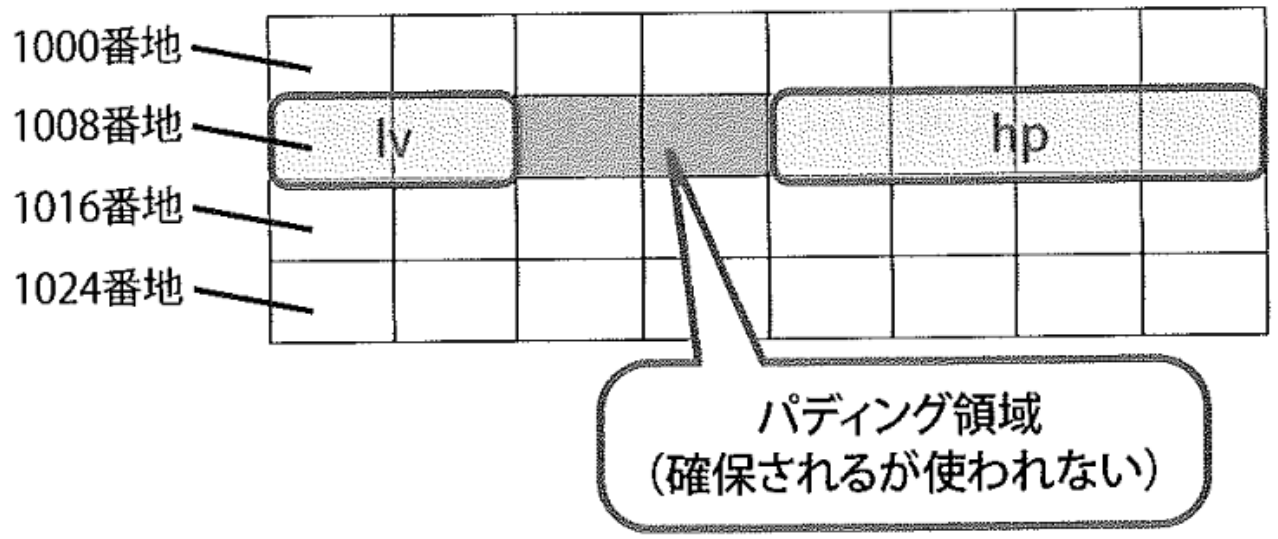
```
double avg(double* a)
{
   double sum = 0.0;

   for(int i=0; i<5; i++) {
      sum += a[i];
   }

   return sum / 5;
}
```

メモリにアクセスする手段と注意点

■構造体のメモリ配置

- 基本的には各メンバ用の領域が連続した場所に確保されるが，実行環境によっては隙間（パディング）が生まれる

```
struct {
   short lv;
   int hp;
} x;
```

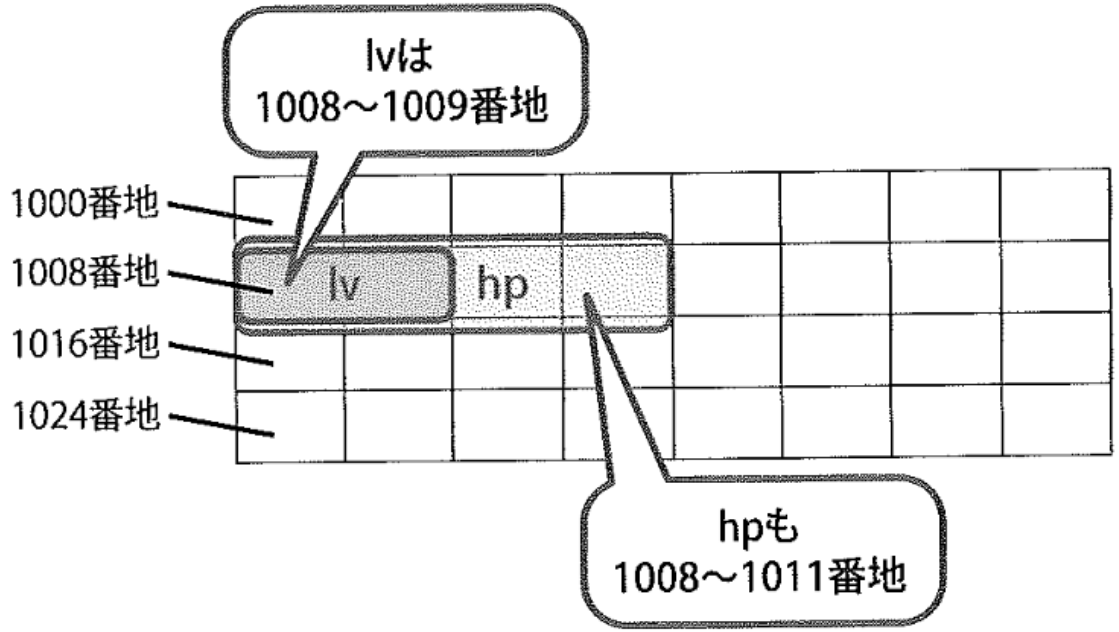


メモリにアクセスする手段と注意点

■共用体

- 構造体と似たデータ型だが，各メンバが同じメモリ領域を利用するために，状況によってはメモリを節約できる
  - ➢ 基本的には1つのメンバしか正常に値を格納できない

```
union {
    short lv;
    int hp;
} x;
```



メモリにアクセスする手段と注意点

■バッファオーバーフロー（バッファオーバーラン）

- バッファ（確保した／利用している領域）を超えてメモリにアクセス（メモリ上の値を操作）してしまうこと
- 自動的に防ぐことはできないのでくれぐれも注意！

```
int age;
int gems[10];
String password;
```