Programming II 0x0a

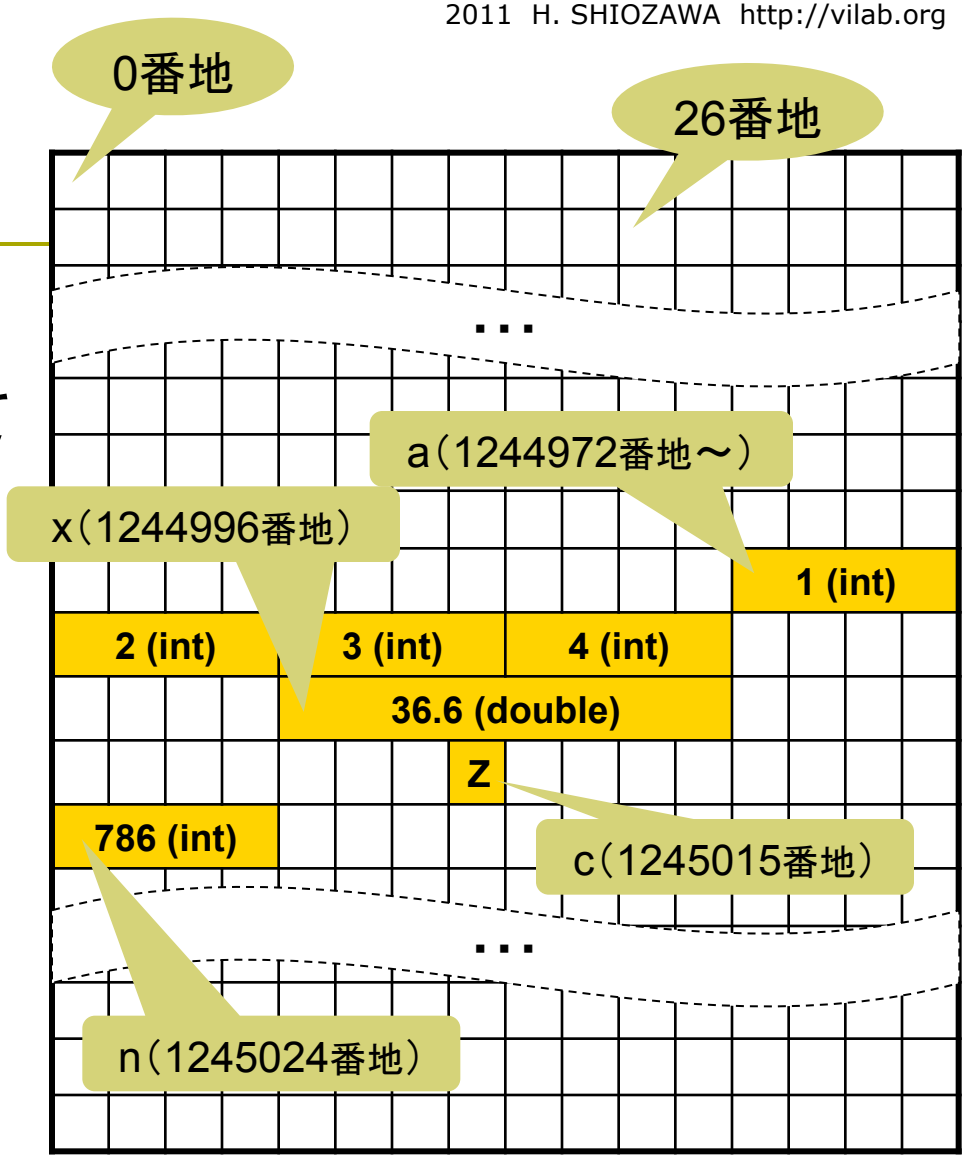
ポインタと関数（2011.06.11）

塩澤秀和 http://vilab.org

メモリ空間(復習)

- メモリアドレス(番地)
  - コンピュータはどうやって情報を記憶するのか？
  - メモリには通し番号の“番地”がついている

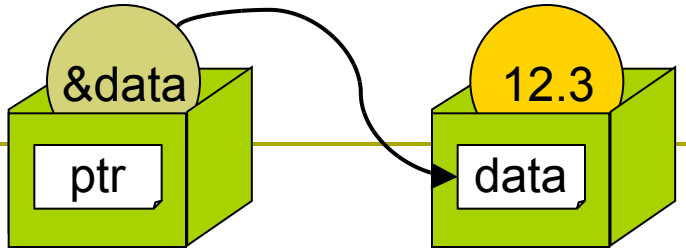
```
int n = 786;
char c = 'Z';
double x = 36.6;
int a[4] = {1,2,3,4};
```



メモリ空間のイメージ

- データのサイズ(p.22)
  - データの種類によってサイズが決まっている
  - char 8ビット＝1バイト　int 32ビット＝4バイト(32ビット機)
  - float 32ビット＝4バイト　double 64ビット＝8バイト

ポインタ変数(復習)



- ポインタ(pointer)とは
  - データの格納された場所を示す値＝アドレス
  - または、そのアドレスを格納するための変数

- ポインタ変数の定義
  - 定義(宣言):「型名 ＊ 変数名;」
  - 必ずメモリの中のデータの型に対応したポインタ型を使うこと

メモリの中のデータの型	対応するポインタの定義例
int	int *ptr;
double	double *ptr;
char	char *ptr;

- 代入と初期化
  - ptr = &data; ← ポインタptrに変数dataのアドレスを代入
  - int *p = &n; ← ポインタpの値をnのアドレスで初期化

& 演算子（復習）

- &演算子（アドレス演算子）（p.147）
  - 変数の格納場所のアドレス（番地）を求める
    【使い方】 &変数名
  - アドレス（ポインタ）を表示する例
    printf("%p", &x); ← %pはポインタのための書式
    printf("%d", (int) &x); ← 無理やり10進数で表示する例

scanf で
おなじみの&

- sizeof 演算子（p.154）
  - 変数やデータ型のサイズ（バイト数）を求める
    sizeof(変数名) 例： sizeof(x)
    sizeof(データ型名) 例： sizeof(int)
  - サイズを表示する例： printf("%d", (int) sizeof(x));

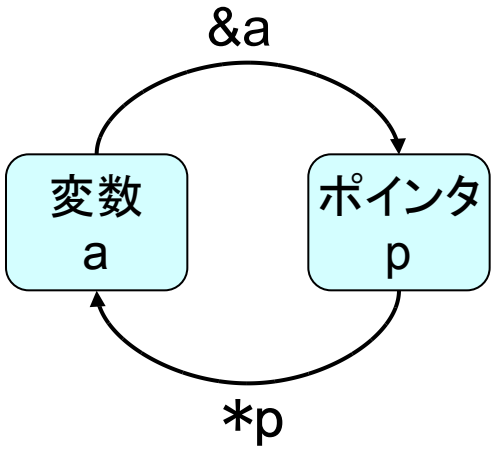
＊演算子（復習）

- ＊演算子（間接演算子）（p.147）
  - ポインタが指す先のメモリを参照する

    x = *p;　⇒ メモリのp番地のデータを、変数xに代入する

    *p = 5;　⇒ メモリのp番地に、データ「5」を代入する
  - つまり… p = &a とすると、*p が変数 a の“別名”になる

- 「&」と「＊」は逆の関係
  - &: 番地を調べる　　＊: 番地をたどる
  - *(&a) ＝ *p ＝ a　　&(*p) ＝ &a ＝ p



- ポインタ使用上の注意
  - **指す先がないポインタを使わないように注意すること！**

関数にポインタを渡す(p.163)

- ポインタ引数
  - 関数にポインタ(変数がある番地)を渡す
  - 関数は番地をたどって変数の中身を変更可能

```
#include <stdio.h>
void read_int(int *ptr);

int main(void)
{
  int num;
  read_int(&num);
  printf("num = %d\n", num);
  return 0;
}

/* ポインタを受け取る関数 */
void read_int(int *ptr)
{
  int a;
  printf("整数を入力: ");
  scanf("%d", &a);
  *ptr = a;
}
```

num の番地を渡す

ポインタ変数

関数の最初に ptr = # と代入される

*ptr は num の別名になっている

- scanfの「&」
  - scanf は変数のポインタを受け取る関数
  - なぜそうするのだろう？

引数渡しのしくみ(復習)

- 普通の引数では(p.83)
  - 関数内で変数(仮引数)の値を変更しても、呼び出し元の変数(実引数)は変わらない
  - 右の例でplus10の中でxを変えても、mainのaには影響しない
  - 理由:引数は値がコピーされるから(「値渡し」call by value)
- でも、配列引数は例外
  - 配列の場合は、値のコピーではなく、**"場所"**が渡される
  - 関数の中で、呼び出し元の配列の要素を変更できてしまう
  - 「参照渡し」call by reference

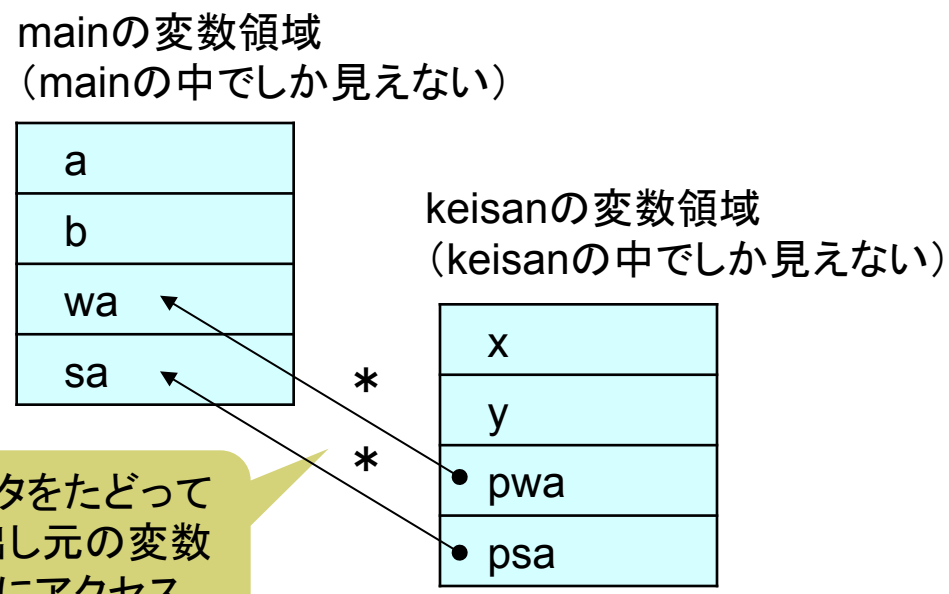
```
#include <stdio.h>

void plus10(int x)
{
  /* 関数内で変数の値を変えても */
  x = x + 10;
}

int main(void)
{
  int a;

  scanf("%d", &a);
  plus10(a);
  /* 元の変数の値は変わらない */
  printf("%d\n", a);
  return 0;
}
```

関数から複数の値を戻す

ここでは分かりやすいように、変数名を変えたが、教科書の「リスト7-12」も理解せよ

```
/* 和と差を一緒に求める関数 */
/* リスト7-12(p.164)一部変更 */
#include <stdio.h>

void keisan(int x, int y,
     int *pwa, int *psa);

int main(void)
{
  int a = 2, b = 5;
  int wa, sa;

  keisan(a, b, &wa, &sa);

  printf("和=%d, 差=%d",
         wa, sa);
  return 0;
}
```

```
void keisan(int x, int y,
     int *pwa, int *psa)
{
  *pwa = x + y;
  *psa = x - y;
}
```



ポインタをたどって呼び出し元の変数領域にアクセス

関数で配列を書き換える(復習)

```
/* 関数内で配列の全要素を2倍する */
/* リスト5-18 (p.110) を一部変更     */
#include <stdio.h>
/* 定数マクロの利用例 */
#define N 3

void multi2(int b[N]);

int main(void)
{
  int i, a[N];

  for(i = 0; i < N; i++) {
    printf("a[%d] = ", i);
    scanf("%d", &a[i]);
  }

  multi2( a );
```

実引数

```
  for(i = 0; i < N; i++) {
    printf("a[%d] = %d\n",
           i, a[i]);
  }
  return 0;
}
```

aの内容が変わっていることを確認

```
/* 配列の全要素を2倍 */
void multi2(int b [N])
{
  int i;

  for(i = 0; i < N; i++) {
    b[i] = b[i] * 2;
  }
}
```

仮引数

関数内で配列の中身を書き換えられる！

配列引数とポインタ

- 実は、関数に渡しているのは…
  - 配列全体のコピーではなく、先頭要素へのポインタだけ
    ⇒ ポインタを渡しているから、配列の中身を変更できる
  - どちらの形式で書いても意味は同じ（自分で確認せよ）

	配列としての書き方	ポインタとしての書き方
関数呼び出し	`multi2(a);`	`multi2(&a[0]);`
関数定義	`void multi2(int b[])` `{`	`void multi2(int *b)` `{`

- 配列の名前≒ポインタ（次回）
  - array ⇔ &array[0]
  - ポインタ ptr に対しても ptr[i] のような書き方ができる

演習問題

10a. int型変数のアドレスを引数に渡すと、その変数の中身に直接 10 を加える関数 plus10 を作成しなさい。

- 関数プロトタイプ void plus10(int *ptr);

10b. 下記のプロトタイプを参考に、整数の割り算の商と余りを求めるための関数 warizan を作成しなさい。

- 関数プロトタイプ void warizan(int a, int b, int *sho, int *amari);

10c. 倍精度型へのポインタ p1, p2 を引数に取り、p1 と p2 の指す先の変数の値を交換する関数 koukan を作成しなさい。

- 関数プロトタイプ void koukan(double *p1, double *p2);

10d. ポインタ a が指すdouble型配列の先頭から n 個の要素をそれぞれ x 倍する関数 xbai を作成しなさい。

- 関数プロトタイプ void xbai(double *a, int n, double x);