

C言語を用いたプログラミング7

Key Words:

- ・変数のアドレス(メモリ上の記憶場所)
- ・ポインタ(アドレスを記憶する変数)
- ・アドレス演算子 &
- ・間接演算子 *
- ・添字演算子 []
- ・配列を関数で処理する方法

目的:

変数名ではなく、アドレスを利用して
格納データにアクセスしたい

変数の宣言、変数のアドレス

```
double x=3.14;
```

- (1) メモリ上にdouble型の記憶領域を確保する。
- (2) その領域を変数名xでアクセスできるようにする。
- (3) xの領域に3.14を記憶(格納)する。

プログラム7.1



```
printf("...%p...", &x);
```

データの格納場所(アドレス)を表示する

アドレス演算子	意味
&	変数(データ)の格納場所(アドレス)を求める

C言語の特徴

変数名だけでなく、アドレスでも格納データにアクセスできる

この特徴を利用した処理

分類	特徴
1 関数	変数の独立性を超えて、同じ格納場所(アドレス)にアクセスできる。
2 配列	データが連続した領域に格納されているため、領域の先頭からのオフセットで要素の格納場所(アドレス)を指定できる。
3 関数 + 配列	配列の格納場所(アドレス)を受ける関数を作成すれば、コンパクトで効率的なデータ処理が可能となる。

特徴1: 関数

変数の独立性を超えて、同じ格納場所(アドレス)にアクセスできる

呼出側:&

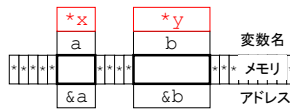
データの格納場所
(アドレス)を伝える

```
int main(void) {  
    int a;  
    double b;  
    ...  
    func(&a, &b)  
    ...  
}
```

関数側:*

そのアドレスを記憶し、
格納データを処理する

```
func(int *x, double *y) {  
    *x=...  
}
```



プログラム例

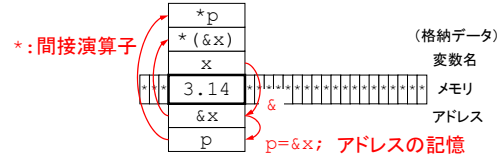
- ・プログラム6.3 (swap関数:2数の交換)
- ・独自問題6.2 (sort2関数:2数の昇順入れ替え)

特徴1の原理

```
double x=3.14;  
double *p;  
p=&x;
```

- (1) アドレスを記憶する変数pをメモリ上に確保する。
- (2) そのアドレスの格納データをdouble型として扱う。
この段階ではアドレスは不定。

プログラム7.2



間接演算子	意味
*	指定されたアドレスの格納データにアクセスする (アドレスを経由して間接的にアクセスする)

独自問題7.1

次のプログラムを実行し、アドレスを利用して
格納データにアクセスできることを確認しなさい。

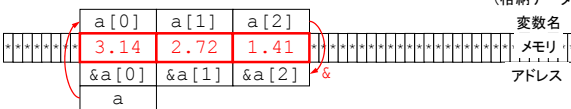
```
#include <stdio.h>  
  
int main(void) {  
    double x=3.14;  
    double *p;  
  
    p=&x;  
  
    printf("格納場所(アドレス):%p,%p\n", &x, p);  
    printf("格納データ:%f,%f,%f\n", x, *(&x), *p);  
  
    return 0;  
}
```

特徴2: 配列

データが連続した領域に格納されているため、
領域の先頭からのオフセットで指定できる。

```
double a[3]={3.14,2.72,1.41};
```

[]: 添字演算子



配列名: データの格納場所(領域の先頭アドレス)

添字演算子	意味
[]	指定されたオフセットで格納データにアクセスする

特徴2の原理

```
double a[3]={3.14,2.72,1.41};  
double *p;  
p=a;
```

配列の(先頭)アドレスを記憶



独自問題7.2

次のプログラムを実行し、アドレスを利用して
配列データにアクセスできることを確認しなさい。

```
#include <stdio.h>  
  
int main(void) {  
    double a[3]={3.14,2.72,1.41};  
    double *p;  
  
    p=a;  
    printf("配列名で:%f,%f,%f\n",  
           a[0],a[1],a[2]);  
    printf("アドレスのオフセットで:%f,%f,%f\n",  
           p[0],p[1],p[2]);  
  
    return 0;  
}
```

特徴3: 関数+配列

10

配列の格納場所(アドレス)を受ける関数を作成すれば、コンパクトで効率的なデータ処理が可能となる。

呼出側: 配列の格納場所(アドレス)を関数に伝える

```
double a[3]={3.14,2.72,1.41};
func(a,3);
```

関数名(配列名, 要素数) 要素数も関数に伝えることで関数の独立性を高くする

関数側: そのアドレスを記憶し、オフセットで要素を指定する。

```
double func(double p[], int n) {
    処理
}
```

あるいは

```
double func(double *p, int n) {
    処理
}
```

配列の非効率な渡し方

11

呼出側: 要素毎に格納データを渡す

```
double a[3]={3.14,2.72,1.41};
func(a[0],a[1],a[2]);
```

関数側: 対応する数の変数で受ける。

```
double func(double x0,double x1,double x2) {
    処理
}
```

- データを複写するため、非効率
 - データ数が限定され、汎用性がない
 - forによる反復処理ができない
- 現実的ではない

独自問題7.3

12

次の設計仕様を満たす関数を作成し、実行確認するプログラムを作成しなさい。

関数名	sum
書式	double sum(double a[], int n)
機能	配列に格納された数値の合計を求める
引数	1次元配列a[], (double型) 要素数n, (int型)
戻り値	合計

$$\text{式 } S_n = a_0 + a_1 + \dots + a_{n-1} = \sum_{i=0}^{n-1} a_i$$

アルゴリズム

$$\begin{cases} S_0 = 0 \\ S_{i+1} = S_i + a_i \\ (i = 0, 1, 2, \dots, n-1) \end{cases}$$

プログラム

```
s=0;
for (i=0;i<n;i++) {
    s+=a[i];
}
```

独自問題7.3 (呼出側)

13

```
#include <stdio.h>

/* ここにプロトタイプ宣言 */

int main(void) {
    double x[3]={3.14,2.72,1.41};
    double s;

    s = sum(x,3);
    printf("合計=%f\n",s);

    return 0;
}
```

独自問題7.4

14

次の設計仕様を満たす関数を作成し、実行確認するプログラムを作成しなさい。

関数名	max
書式	double max(double a[], int n)
機能	配列に格納された数値の中から最大値を求める
引数	1次元配列a[] (double型) 要素数n (int型)
戻り値	最大値

式

$$M_n := \max\{a_0, a_1, \dots, a_{n-1}\} \longrightarrow \begin{cases} M_1 = a_0 \\ M_{i+1} = \max\{M_i, a_i\} \\ (i = 1, 2, \dots, n-1) \end{cases}$$

アルゴリズム

独自問題7.4 (呼出側)

15

```
#include <stdio.h>
/* ここにプロトタイプ宣言 */

int main(void) {
    int i;
    double x[5],xmax;

    printf("%d個の実数を入力して下さい\n",5);
    for (i=0;i<5;i++) {
        printf("x[%d]=" ,i);
        scanf("%lf",&x[i]);
    }
    xmax = max(x,5);
    printf("最大値=%f\n",xmax);

    return 0;
}
```

ポインタ(アドレスを記憶する変数)

16

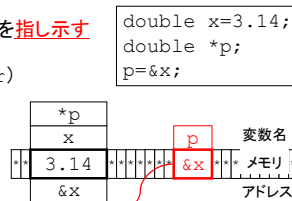
C言語では頻繁に、アドレスを用いて格納データにアクセスする。

↓
アドレスを記憶する変数が有ると便利

↓
変数の機能: 格納データの場所を指し示す

↓
変数の名称: ポインタ(pointer)

(用語)
指し示す: point
指し示すもの: pointer

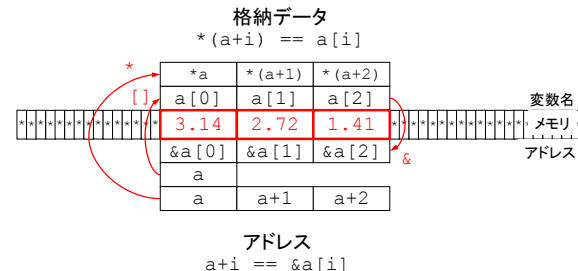


double型データの格納場所を指し示している
point
「pはdouble型へのポインタ」

添字演算子[]と間接演算子*の関係

17

```
double a[3]={3.14,2.72,1.41};
```



コンパイル時に $a[i]$ は $*(a+i)$ に変換して処理される。

アドレスを利用する場合の注意点

18

格納データへのアクセス方法	安全性	理由
変数名による指定	安全	宣言した変数領域しかアクセスできない。
アドレスによる指定(配列, ポインタ)	危険	任意の場所にアクセスできるため、場所を間違えると ・データ破壊 ・暴走

```
double a[3]={3.14,2.72,1.41};
a[3]=10.5;
```

範囲外を指定している

配列はアドレスを経由してアクセスするため、
・コンパイル時にはエラーを生じない。
・実行時に暴走する。