

2021

# 計算機プログラミング演習

第15回 ユーザ定義関数と周辺

- ポインタ -

# 第15回の目標

- ユーザ定義関数の使い方
  - 引数(入力)と戻り値(出力)について再考
  - 関数内の変数と呼び出し側の関係
- ユーザ定義関数のあれこれ
  - 引数(入力)を出力と兼ねたい
    - 引数が入力であり、出力である仕様
    - ポインタが必要

# ユーザ定義関数(復習)

- C言語の関数は数学関数と考えが同じ
- 数学で使われる関数では

$$y = f(x)$$

$f$  : 関数

$x$  : 原因(入力)

$y$  : 結果(出力)

- 関数  $f(x)$  の演算の後に  $x$  が変化することはない

# 先週の復習(1)

```
/* func2.c */  
#include <stdio.h>
```

```
int ftest(int); /* プロトタイプ宣言 */
```

```
int main( void )  
{
```

```
    int b, a = 5;
```

```
    b = ftest(a);  
    printf("%d + 1 = %d\n", a, b);  
    return 0;
```

```
}
```

```
int ftest(int a) {  
    a = a + 1;  
    return a ;
```

```
}
```

変数 a の値 (5)

呼び出し側とは独立した変数 a を作成し、値 (5) を a にコピーする

変数 a の変更は引数に反映されない

return の後の変数 a は戻り値として呼び出し側に返される

# まとめ(1)

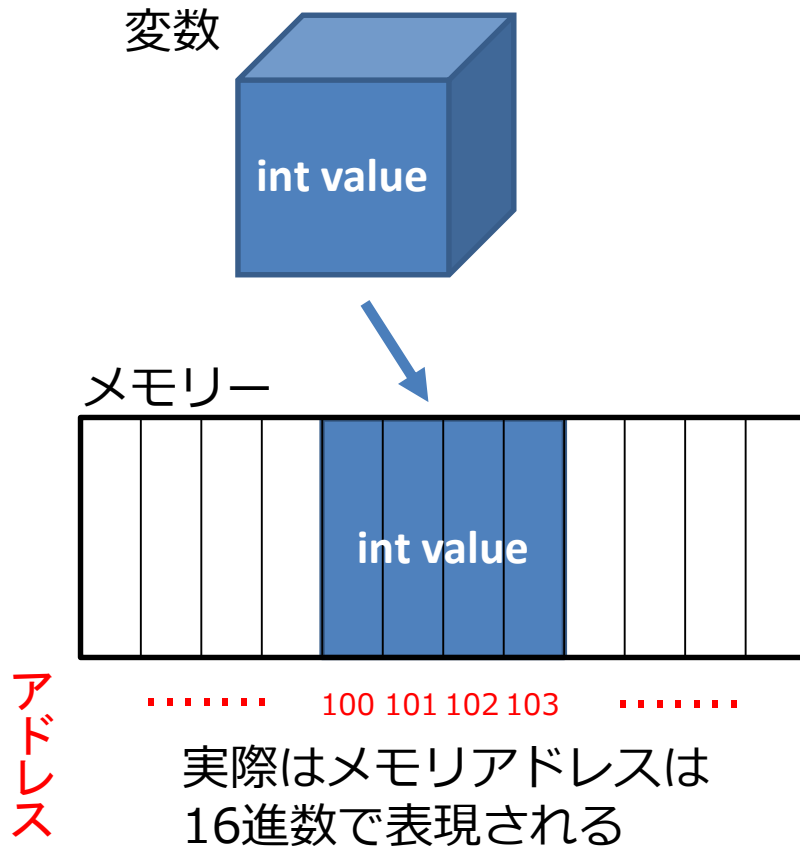
- ユーザ定義関数の引数に使われる変数を関数内で変更しても呼び出し側には反映されない。
  - 関数内の変数(引数も含む)は呼び出し側の変数と独立している。
  - 引数は関数開始時に呼び出し側の引数の値を関数側の引数にコピーしている。
  - 関数終了時に関数側の引数は呼び出し側の引数には反映されない(コピーされない)。
  - 関数の出力は戻り値である。

# それでも変えるには？ もういちど引数について

- 関数内の引数は呼び出し側の引数の値をコピーして使う。
  - 関数内の結果は引数に反映されない。
- 
- 引数に入っているのは数である。
  - 本質的にコンピュータが取り扱うのはすべて「数」である。
  - 変数の場所も 数 で示される。
  - では **変数の場所を示すアドレス(ポインタ)** を引数にしよう！
    - 呼び出し側 も 関数側 も共通の場所(変数)をアクセスできる
    - 呼び出し側 と 関数側 のデータの明示的な共有が可能

# アドレスとポインタ

コンピュータには主記憶装置（メインメモリ）が搭載されている



- 変数を宣言するとメモリ上の領域が確保される
- メモリーの場所を示すアドレスというものがある
- アドレスを扱う際に使用するのがポインタ

ポインタの宣言

データ型 \*ポインタ名;

例) `int *a;`

アドレス取得は変数の前に`&`を付ける

例) `&a;`

# アドレスとポインタ

実際にアドレスを表示してみよう（コンピュータにより表示されるアドレスは異なる）

```
#include <stdio.h>

int main( void )
{
    int a = 5;
    int *p = &a; ポインタにアドレス代入

    printf("変数aのアドレス= %p\n", &a);
    printf("ポインタpに代入されたアドレス= %p\n", p);
    return 0;
}
```

```
C:\Users\Nakano\source\repos\pointer\Debug>pointer
変数aのアドレス = 00BFFB4C
ポインタpに代入されたアドレス = 00BFFB4C
```



# 引数の内容を変えるには(ポインタの使用)

```
/* func2.c */  
#include <stdio.h>
```

```
int ftest(int * ); /* プロトタイプ宣言 */
```

```
int main( void )  
{  
    int b, a = 5;  
  
    b = ftest( &a );  
    printf("%d + 1 = %d\n", a, b);  
    return 0;  
}
```

変数に & を付けてアドレスにし、  
変数 a の場所(数字)を引数にする

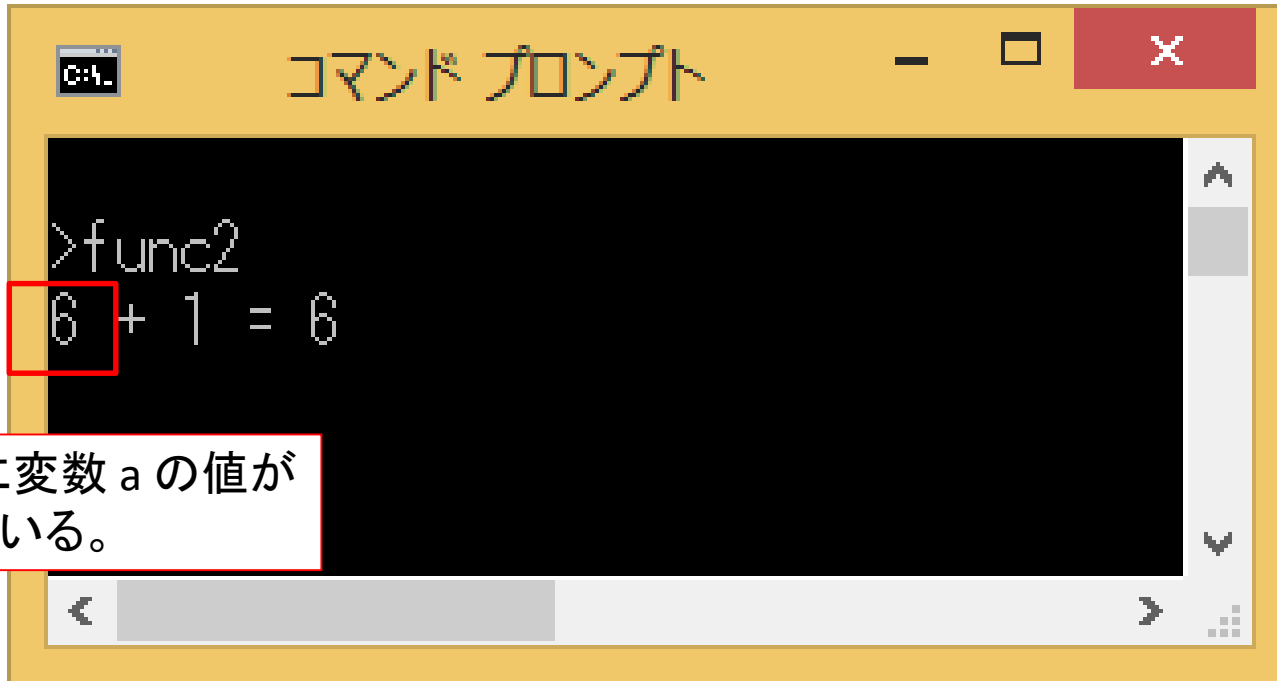
ポインタ変数 p を作成し、変数 a の  
場所をコピーする。

```
int ftest(int * p){  
    *p = *p + 1;  
    return *p ;  
}
```

ポインタ変数 p が示す場所の **内容** に 1 を  
加算する(変数 a の内容が変化する)

Return の後の変数 は戻り値として呼び出し側に返される

# 実行結果



```
>func2
6 + 1 = 6
```

実行後に変数 a の値が  
変化している。

変数 a の場所 (アドレス &a) を受け取り、その場所についての演算を行えば引数内の変数でも変化させることができる。

では、なぜ引数の値を関数内で変化させる  
必要があるのか？

# ポインタについて

- ポインタは必要悪
  - 使わなくて良いなら使わないに越したことはない
    - ポインタとポインタの指し示す内容を混乱しがちになる
    - プログラムが読みにくなる(技巧的なプログラム)
    - バグが生じたときに分析が難しくなる
  - 関数の引数に配列や文字列、構造体の受け渡しに比較的多く使われる
    - 引数の受け渡しに使われるポインタについてのみ説明する

# 複数のデータを持つ変数のやりとりに必要 (配列、文字列、構造体等)(1)

```
/* func3.c */
#include <stdio.h>
int ftest(int); /* プロトタイプ宣言 */

int main( void )
{
    int    b[4], a[4] = { 1, 2, 3, 4 } ;
    int    i ;

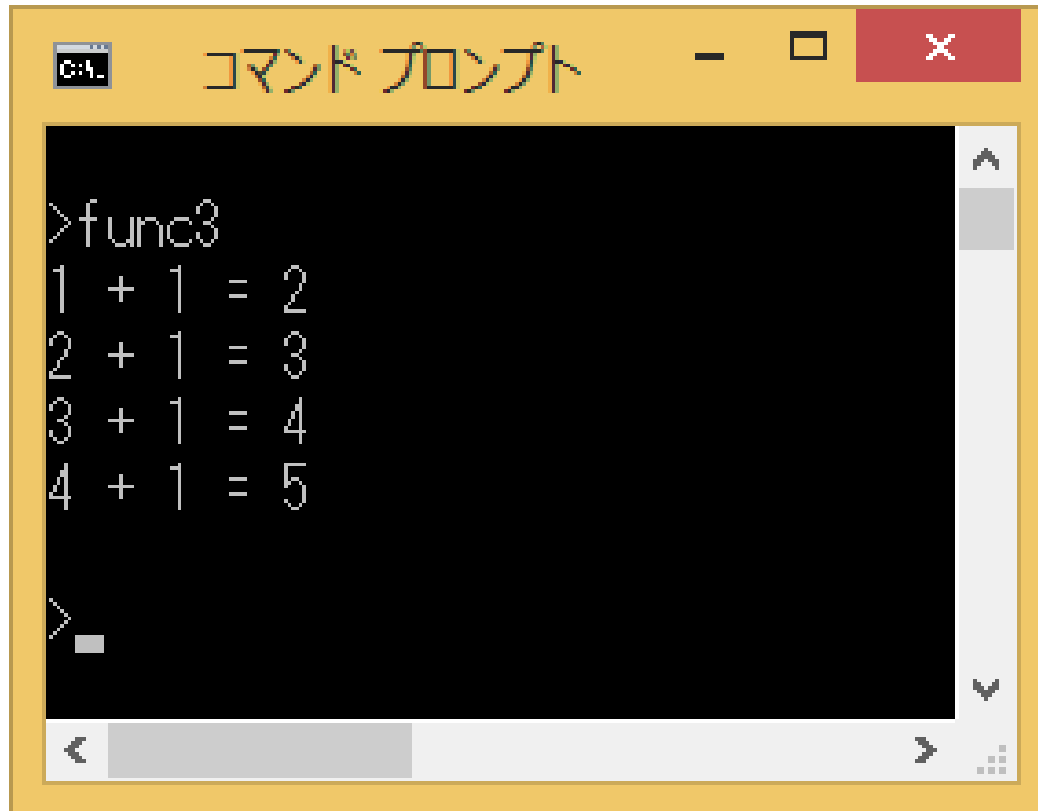
    for ( i = 0 ; i < 4 ; i++ ) {
        b[ i ] = ftest( a[ i ] );
        printf( "%d + 1 = %d\n", a[i], b[i] );
    }
    return 0;
}
```

次スライドにつづく

# 複数のデータを持つ変数のやりとりに必要 (配列、文字列、構造体等)(2)

つづき

```
int ftest(int x ){  
    x = x + 1 ;  
    return x;  
}
```



```
>func3  
1 + 1 = 2  
2 + 1 = 3  
3 + 1 = 4  
4 + 1 = 5  
>
```

配列の個別の要素ごとの演算  
については可能である。

しかしながら、配列全体を関数  
に渡すには引数を配列の個数  
分用意しなければならない。

配列の合計を求めるような関  
数は配列全体を関数に渡さな  
なければならない。

# 複数のデータを持つ変数のやりとりに必要 (配列、文字列、構造体等)(3)

```
/* func4.c */
#include <stdio.h>

void ftest( int [] ) ; /* プロトタイプ宣言 */
void fcopy( int [], int [] ) ;

int main( void )
{
    int    b[4], a[4] = { 1, 2, 3, 4 } ;
    int    i ;

    fcopy( b, a ) ;
    ftest( b ) ;

    for ( i = 0 ; i < 4 ; i++ ){
        printf("%d + 1 = %d\n", a[i], b[i]);
    }
    return 0;
}
```

void は戻り値がないという意味

次スライドにつづく

# 複数のデータを持つ変数のやりとりに必要 (配列、文字列、構造体等)(4)

つづき

```
void fcopy( int x[], int y[] ){
    int i ;

    for ( i = 0 ; i < 4 ; i++ ){
        x[ i ] = y[ i ] ;
    }
    return ;
}

void ftest( int z[] ){
    int i ;

    for ( i = 0 ; i < 4 ; i++ ){
        z[ i ] = z[ i ] + 1 ;
    }
    return ;
}
```

# 複数のデータを持つ変数のやりとりに必要 (配列、文字列、構造体等)(5)

- 関数定義に配列を表す [] を付けて定義しておく、(一次元配列については)配列全体を関数に渡すことができる。
- でも実は、これも **ポインタ** なのである。



# ポインタを使って書き換えてみる(1)

```
/* func4.c */  
#include <stdio.h>
```

int [] を  
int \* に書き換える

```
void ftest( int * ) ; /* プロトタイプ宣言 */  
Void fcopy( int * , int * ) ;
```

```
int main( void )  
{
```

```
    int    b[4], a[4] = { 1, 2, 3, 4 } ;  
    int    i ;
```

```
    fcopy( b, a ) ;  
    ftest( b ) ;
```

配列の変数名だけを引数にすると  
自動的にポインタになっている。

```
    for ( i = 0 ; i < 4 ; i++ ){  
        printf( "%d + 1 = %d\n", a[i], b[i] );  
    }  
    return 0;  
}
```

次スライドにつづく

# ポインタを使って書き換えてみる(2)

つづき

```
void fcopy( int * x , int * y ){  
    int i ;  
  
    for ( i = 0 ; i < 4 ; i++ ){  
        x[ i ] = y[ i ] ;  
    }  
    return ;  
}  
  
void ftest( int * z ){  
    int i ;  
  
    for ( i = 0 ; i < 4 ; i++ ){  
        z[ i ] = z[ i ] + 1 ;  
    }  
    return ;  
}
```

# ポインタを使って書き換えてみる(3)

```
/* func4.c */
#include <stdio.h>

void ftest( int * ) ; /* プロトタイプ宣言 */
Void fcopy( int * , int * ) ;

int main( void )
{
    int    b[4], a[4] = { 1, 2, 3, 4 } ;
    int    i ;

    fcopy( &b[ 0 ] , &a[ 0 ] ) ;
    ftest( &b[ 0 ] ) ;

    for ( i = 0 ; i < 4 ; i++ ){
        printf( "%d + 1 = %d\n" , a[i] , b[i] ) ;
    }
    return 0;
}
```

各配列の先頭データのポインタで  
代用しても同じ結果になる

# 配列とポインタ

- 配列をポインタで書き直してみると
  - 配列を引数にする場合、配列の先頭データ `a[0]` の場所(ポインタ)を渡しているのと等価である。
  - ただし、二次元以上の配列を引数にするときは二次以上の配列のサイズ(インデックス)を渡さなければならない。
    - 次の例を参照のこと
    - インデックスは定数でなければならない
      - 変数不可、`#define` によるマクロは OK

# 二次元以上の配列の受け渡し(1)

```
/* func5.c */  
#include <stdio.h>
```

int [][] ではなく、2番目以降の添字のサイズを記入する

```
void ftest( int[][3] ) ;  
void fcopy( int[][3] , int[][3] ) ;  
  
int main( void )  
{  
    int a[2][3] = { {1, 2, 3}, {4, 5, 6} } ;  
    int b[2][3] ;  
    int i, j ;  
  
    fcopy( b, a ) ;  
    ftest( b ) ;  
  
    for ( i = 0 ; i < 4 ; i++ ){  
        printf("%d + 1 = %d¥n", a[i], b[i]);  
    }  
    return 0;  
}
```

次スライドにつづく

つづき

## 二次元以上の配列の受け渡し(2)

```
void fcopy( int x[][3] , int y[][3] ){
    int i, j ;
    for ( i = 0 ; i < 2 ; i++ ){
        for ( j = 0 ; j < 3 ; j++ ){
            x[ i ][ j ] = y[ i ][ j ] ;
        }
    }
    return ;
}

void ftest( int z[][3] ){
    int i ;

    for ( i = 0 ; i < 2 ; i++ ){
        for ( j = 0 ; j < 3 ; j++ ){
            z[ i ][ j ] = z[ i ][ j ] + 1 ;
        }
    }
    return ;
}
```

# 構造体の引き渡し(1)

- 構造体は必ずしもポインタで渡す必要はない
- 引数に構造体のポインタで渡す必要があるのは、構造体内の変数の内容に変更を加える場合である。
- ポインタで渡された構造体のメンバ変数をアクセスする場合に、 $v \rightarrow x$  のような **-> 演算子 (アロー演算子)** を使わなければならない。
  - ポインタでない場合は  $v.x$  でアクセスする。

## 構造体の引き渡し(2)

```
/* vectest.c */
#include <stdio.h>

typedef struct { /* 3次元ベクトル構造体 */
    double x ;
    double y ;
    double z ;
} vec3d ;

void  vclear( vec3d *v ); /* プロトタイプ宣言 */
vec3d vcopy( vec3d v ); /* c = c1 */

int  main(void) {
    vec3d v0, v1;

    vclear( &v0 );
    v1 = vcopy( v0 );

    return 0;
}
```

次スライドにつづく



## 構造体の引き渡し(3)

つづき

```
void vclear( vec3d * v) {           /* 複素数の零クリア*/

/* ポインタで渡された構造体 v のメンバー変数 x, y, z に代入*/
    v->x = 0;
    v->y = 0;
    v->z = 0 ;
    return;
}

vec3d vcopy( vec3d v1 ) {           /* ベクトルのコピー */
                                     /* v = v1 */
                                     /* 結果のための構造体を用意*/

    vec3d v;

    v.x = v1.x ;
    v.y = v1.y ;
    v.z = v1.z ;

    return v;                       /* v の内容を呼出側にコピー*/
}
```