

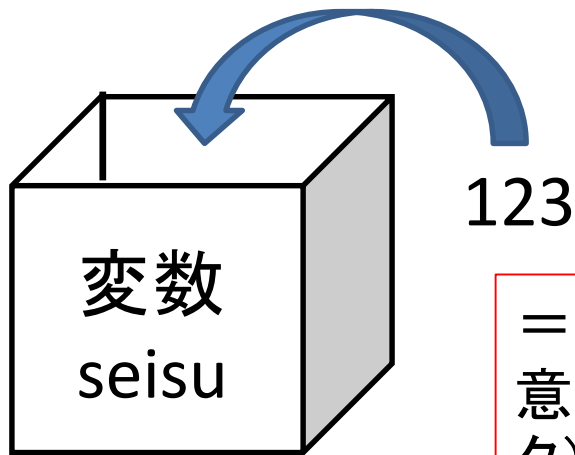
# 2021 計算機プログラミング演習

第16回 講義のまとめ

これまでの内容の復習

# 変数とは

数学にも変数と呼ばれる概念があるが、プログラミングにおける変数は“**色々な種類の値を入れる(代入する)ことができる箱**”である



例)

```
int seisu;    変数の生成  
seisu = 123;  値の代入
```

= (イコール) は数学では両辺が等しいという意味になるが、プログラミングでは右辺(データ)を左辺(変数)に**代入する**という意味になる

変数に入れることができる値としては、

- 整数 (1, 2, 3, )
- 実数 (1.23456)
- 文字列 (“abcdef”)

# 変数のルール

変数を用いるには以下のルールが存在する

- 変数を使用(生成)すると宣言する(変数宣言)
- どの種類の値を入れる変数なのかを指定する(変数の型)
- 変数は名前(変数名)を付ける
- 「変数=値;」のイコールは両辺が等しいということではなく、値を変数に代入することを意味する

# 変数の宣言

- 変数を使用するには、変数の型、変数名を宣言

変数の型 変数名;

例) `int seisu;`  
`double syosu;`

- 同じ変数の型の変数名は同時に複数宣言可能

変数の型 変数1, 変数2, 変数3;

例) `int a, b, c;`

変数名はその変数の役割等が分かるように簡潔に付けるのがよい

# 変数の型

## 第3回の内容

値の種類	変数の型	名称	ビット長	値の範囲
型なし	void	void型	無し	無し
整数 []は省略可能	[signed] short int	(符号付き)短整数型	16	-32768 ~ 32767
	unsigned short int	符号なし短整数型	16	0 ~ 65535
	[signed] int	(符号付き)整数型	32	-2147483648 ~ 2147483647
	unsigned int	符号なし整数型	32	0 ~ 4294967295
	[signed]long int	(符号付き)長整数型	32	-2147483648 ~ 2147483647
	unsigned ling int	符号なし長整数型	32	0 ~ 4294967295
	[signed]long long int	(符号付き)長長整数型	64	-9223372036854775808 ~ 9223372036854775807
	unsigned long long int	符号なし長長整数型	64	0 ~ 18446744073709551615
浮動小数点	float	単精度実浮動小数点型	32	Min: 1.175494351e-38 Max: 3.402823466e+38
	double	倍精度実浮動小数点型	64	Min: 2.2250738585072014e-308 Max: 1.7976931348623158e+308
文字列	[singed] char	符号付き 文字型	8	-128 ~ 127
	[unsigned] char	符号なし 文字型	8	0 ~ 255

# 変数へ値の代入

## 第3回の内容

- 変数を宣言すると同時に値を入れることができる  
(初期化)

変数の型 変数名 = 値;

例) `int num = 0;`

- 変数に入っていた値に加算される

変数 = 変数 + 1;

例) `int num1 = 1;`  
`num1 = num1 + 1;`

# 変数を用いた演算

- 計算の結果を変数に代入

```
num1 = 1 + 3;
```

- 変数同士の演算を変数に代入

```
int num1 = 2;
```

```
int num2 = 3;
```

```
int num3 = 0;
```

```
num3 = num1 + num2;
```



# 変数名のつけ方(1)

## エラーにならない変数名

- アルファベットと数字を使う (半角英数)
  - 記号やスペースは使えない
    - 例外 `_` (アンダースコア)
  - 第1文字は必ずアルファベット
  - C言語の文法ですでに使われている名前(予約語)は使ってはいけない
    - (OK) `a_and_b`, `alpha`, `beta2`, `miyazaki`,
    - (NG) `a b`, `a+b`, `3nensei`, `printf`

# 変数名のつけ方(2)

## エラーにはならないが...

- 大文字と小文字は区別される
- 基本的に小文字にする
- 簡潔でわかりやすい変数名を付ける
  - 使い捨ての変数でない限り, 変数の意味を名前に付ける
  - 英語でなくローマ字でも可
    - (良くない例) a, b, x, y, z (一文字)
    - (良い例) value, sum1, max, goukei, shouhizei

# 変数名のつけ方(3)

- 変数名に 整数/実数の区別を付ける
  - (慣例的に) 整数変数を i, j, k, l, m, n で始まる変数名にする
    - number, num1, iter, member ....
    - ivalue, nfiles, nrow, ncol, ....
  - 変数名の最初に i (int), f (float), d (double) を付ける
    - iClassNumber, fHeight, dWeight,
    - 変数名にスペースが使えないため, 変数名に2つ以上の単語をつなげる場合, 単語の先頭の文字だけ大文字にする

# if の使い方

- if による条件判別(1)

条件式 に合致すれば(真) 処理 を行う。

条件式 に合致しなければ(偽) 処理 を行わない。

処理 は複数行にわたっても構わない。

```
if ( 条件式 )  
{  
    処理 ;  
}
```

# if の使い方

- if による条件判別(2)

条件式 に合致すれば(真であれば) 処理1 を行う。

条件式 に合わなければ(偽であれば) 処理2 を行う。

```
if ( 条件 )  
{  
  
    処理1 ;  
  
} else {  
  
    処理2 ;  
  
}
```

# if の使い方

- よく間違える点
  - 条件内で等号(==)と代入(=)を間違える

```
if ( dvalue2 = 0 ) {
```

 誤り

```
if ( dvalue2 == 0 ) {
```

 正しい

- 間違っているにもかかわらずエラーが出ないので要注意！！
- エラーは出ないが、正しい結果も出ない！！

# 条件式について

## 第5回の内容

### 条件式が真 (True) の場合

```
if ( 条件式 )  
{  
    処理 ;  
}
```

カッコ内の処理を実行

### 条件式が偽 (False) の場合

```
if ( 条件式 )  
{  
    処理 ;  
}
```

カッコ内の処理を実行しない

```
if ( k )  
{  
    処理 ;  
}
```

if ( k ) 変数 k (整数) が

- 0 以外の場合 ⇒ 真
- 0 の時 ⇒ 偽

# 間違いやすい例

## 第5回の内容

```
/*    if_test1.c    */
```

```
#include <stdio.h>
```

```
int main(void)
```

```
{
```

```
    int a = 1;
```

```
    if ( a = 0 )
```

If (a = 0)だと条件式は偽

```
{
```

```
    printf("a はゼロ¥n");
```

```
}
```

```
    printf(" a = %d¥n", a);
```

```
    return 0;
```

```
}
```

カッコ内の処理  
を実行しない

ビルドしても  
正常終了



# for文

- for文は指定した繰り返し回数まで繰り返し処理を行う
- 繰り返し回数が決まっている場合はfor文を使用する
- 繰り返し回数が分からない場合はwhile文を使用する

```
int i;
```

```
for( i = 1; i <= 繰り返し回数; i++ ){  
    実行文;  
}
```

# for の使い方(1)

```
/*      for_test1.c      */  
  
#include <stdio.h>  
  
int main( void ){  
    int    i ;  
  
    for ( i = 1 ; i < 10 ; i++ ) {  
  
        printf( " i = %d\\n", i ) ;  
  
    }  
  
    return 0 ;  
  
}
```

ループ変数 *i* を最初に1に設定し(*i* = 1)、中括弧内の処理 ( printf(...) ) を実行したあと、*i* を1増やす(*i*++)。  
*i* が10未満 (*i* < 10) の間は中括弧内の処理を実行する。

1 から 9 までの数を入力

# for の使い方 (2)

```
/*      for_test1.c      */
```

```
#include <stdio.h>
```

```
int main( void ) {  
    int    i ;
```

```
    for ( i = 1 ; i <= 10 ; i++ ) {
```

```
        printf( " i = %d¥n", i ) ;
```

```
    }
```

```
    return 0 ;
```

```
}
```

ループ変数 `i` を最初に1に設定し(`i = 1`)、  
中括弧内の処理( `printf(...)` )を実行したあと、  
`i` を1増やす(`i++`)。  
`i` が10以下(`i <= 10`)の間は  
中括弧内の処理を実行する。

1 から 10 までの数を出力

# break による脱出

```
/*      for_test2.c      */

#define _CRT_SECURE_NO_WARNINGS 1
#include <stdio.h>

int main( void ){
    int    i, a;
    int    sum = 0 ;

    for ( i = 1 ; i <= 10 ; i++ )
    {
        scanf("%d",&a );

        if ( a < 0 ) { break ; }

        sum = sum + a ;
    }
    printf( "%d 個の合計: %d¥n", i - 1, sum ) ;
    return 0 ;
}
```

初期値は 0

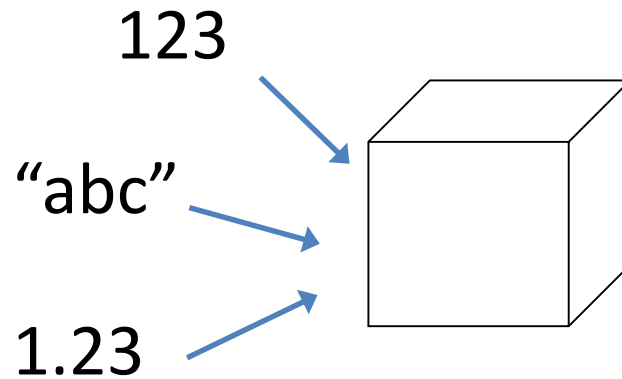
入力値がマイナスならループを中断

**【注意！】** break による中断は脱出条件がわかりにくくなるので、バグのもとになりやすい。できるかぎり避けること。

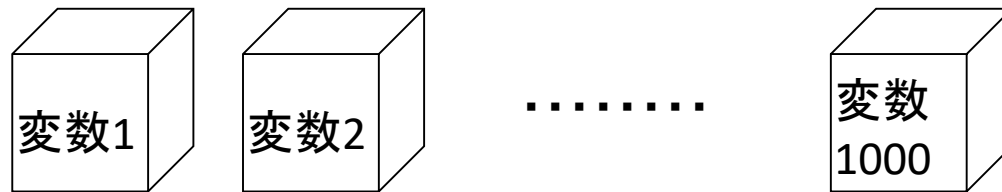
# 配列とは (1/3)

## 第7回の内容

本講義では、変数は決められたデータ型のデータを入れることができる箱のようなものだと教えた



では、次のような計算を行うことを考えよう. ランダムな整数の値を持つ1000個の整数の和を求めたい.



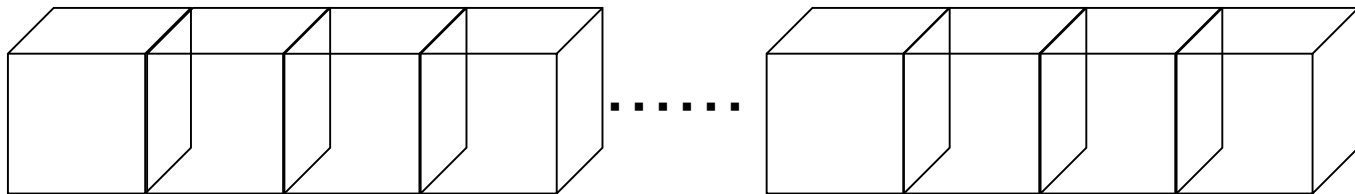
この場合、変数宣言を1000回行い、1000個の変数を用意しなければならない？

# 配列とは (2/3)

## 第7回の内容

変数の宣言を1000回を行うのは、記述が非常に大変であるだけでなく、プログラムも無駄に長くなってしまい非常に見づらい

以下のように同じデータ型を持つ多くの変数を、一度だけの宣言で扱えないだろうか？



実は、C言語を含む多くのプログラミング言語には、上記のような要望を満たす“配列”と呼ばれるものが用意されている

# 配列とは (3/3)

第7回の内容



## 配列の宣言

型名 配列名 [要素数]:

例) int seisu[1000]:

double syosu[100]:

基本的には、変数の宣言と同じだが、[]の中の要素数が配列の数である

注意) 配列の番号(添え字)はゼロから始まる数字である  
したがって、要素数が1000の場合、添字は0~999である

# 配列(4)

## 第7回の内容

### 背番号で区別したプログラム(2)

```
/*      shincho3.c      */
```

```
#include <stdio.h>
```

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

配列の初期値代入

```
    double shincho[6] = { 0, 185, 182, 181, 186,  
                          186 } ;
```

```
    double sum = 0 ;  
    int i = 0;
```

要素数は6 ただし添え字は0～5である

```
    for ( i = 1 ; i <= 5 ; i++ ) {  
        sum = sum + shincho[ i ] ;  
    }
```

添え字1～5の配列  
に存在する身長  
データで平均身長  
を算出

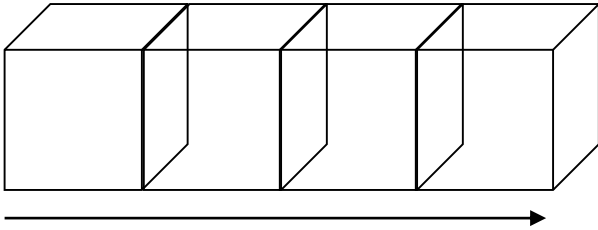
```
    printf( "平均身長  %f¥n", sum / 5 ) ;  
    return 0 ;
```

```
}
```

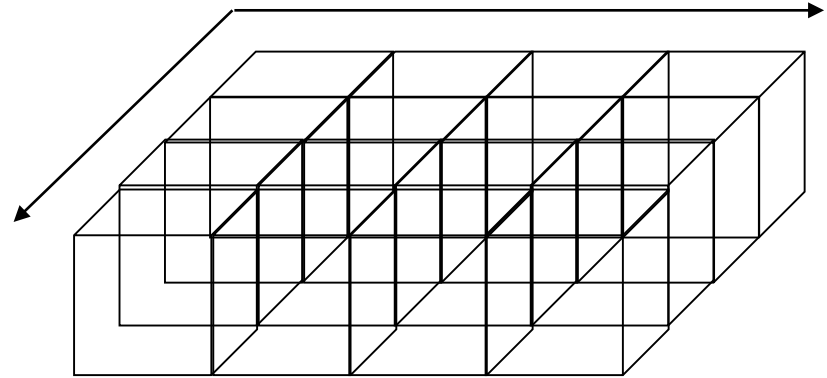


# 2次元配列とは

第8回の内容



1次元配列



2次元配列

## 配列の宣言

型名 配列名 [要素数] [要素数];

例) `int seisu[10][20];`

配列は多次元化することは可能. ただし, 下記の点に注意

- 3次元以上は人にとって理解しにくい
- メモリを消費するので本当に必要か注意すること

# 二次元配列(1)

- 出席番号1～8の生徒の国語と数学と英語成績

添え字 →		0	1	2	3	4	5	6	7
↓	番号	1	2	3	4	5	6	7	8
	国語	45	64	58	66	51	76	45	39
	数学	50	72	92	62	33	88	53	62
	英語	65	84	43	54	45	65	35	38

データが2次元の行列(3行8列)となっている

# 二次元配列(2)

## 生徒の合計と教科の平均を求めるプログラム

```
// two_dim_test.c

#include <stdio.h>

int main( void )
{
    //      二次元配列の初期代入

    int    mark[8][3] = {
        { 45, 50, 65 }, { 64, 72, 84 },
        { 58, 92, 43 }, { 60, 62, 54 },
        { 51, 33, 45 }, { 76, 88, 65 },
        { 45, 53, 35 }, { 39, 62, 38 } } ;
```

# 平均を求めるプログラム

## 第8回の内容

```
int  sum, i, j ;

for (i = 0; i <= 7; i++) {
    sum = 0 ;
    for (j = 0; j <= 2; j++) {
        sum = sum + mark[ i ][ j ];
    }
    printf("生徒 %d の合計 %d ¥n", i, sum );
}

for ( i = 0 ; i <= 2 ; i++ ) {
    sum = 0 ;
    for ( j = 0 ; j <= 7 ; j++ ) {
        sum = sum + mark[ j ][ i ] ;
    }
    printf("教科 %d の平均点: %d¥n", i, sum / 8 );
}
return 0 ;
}
```

# 文字列

- char 型で扱われる
  - char 型とは本来 8bit (1 byte) の整数  
( -128~127, unsigned 0~255)
  - (日本語と違って) 英語に使われる文字は記号を入れても100種類程度
  - 文字(例えば 'A')に番号( 65 ) を付けて規格化(ASCII, JIS など)されている。
  - よって char 型は「文字の入れ物」として使われることが多い。
- char 型の配列を使う
  - 文字一つ('A')だけではあまり意味をなさない。
  - 意味を表すためには複数の文字を連ねる。-> 配列
  - 扱うのが単語や文章になり、必要な長さ(配列の大きさ)がまちまちである。
  - char (一文字) 'A',      文字列 "ABC"

# char 型について考える(1)

```
/* char_test.c : 文字コード 65 ('A') と 97 ('a') を表示 */
#include <stdio.h>

int main( void ){

    char la, sa, lz;

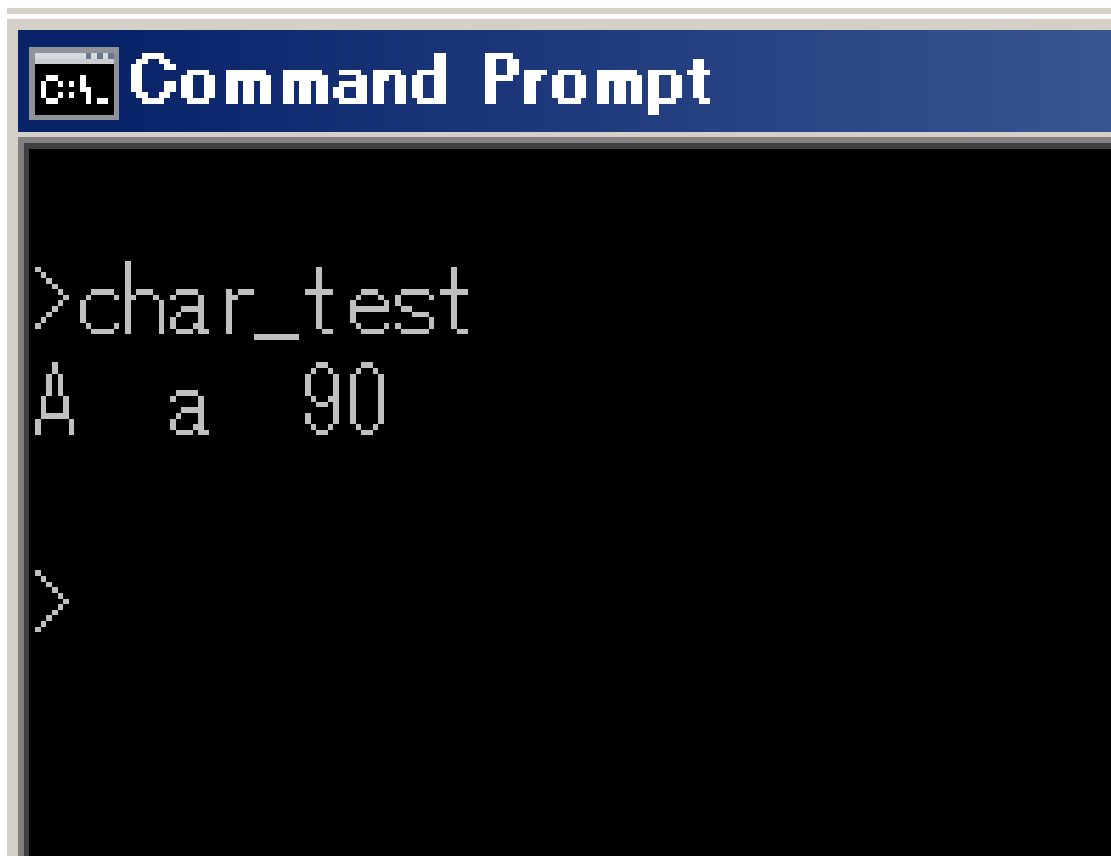
    la = 65;
    sa = 97;
    lz = 'Z';

    printf("%c %c %d\n", la, sa, lz );

    return 0;
}
```

# 実行結果

- `%c` で文字に、`%d` で数字として表示されている。



```
C:\>char_test
A  a  90

>
```

The screenshot shows a Windows Command Prompt window with a blue title bar that reads "C:\> Command Prompt". The command prompt has a black background with white text. The first line shows the command `>char_test`. The second line shows the output `A a 90`, where 'A' and 'a' are separated by two spaces, and '90' is separated by two spaces. The third line shows the prompt `>`.

# char 型について考える(2)

```
/* char_test.c      */  
  
#include <stdio.h>  
  
int main( void ){  
  
    char ch;  
  
    ch = '電';  
    printf("%c %d¥n", ch, ch);  
  
    return 0;  
}
```

A terminal window showing the output of the program. The first line shows the character '電' followed by a space and the number 100. The second line shows the character '電' followed by a space and the number 100.

日本語(漢字コード)は2バイトコードなので char (1バイト)では収まらない。(一文字としては使えない。)



# 文字列 ( char 型の配列 )

## 第9回の内容

- 文字列に使う変数の定義
  - 扱う文字数+1 以上の大きさの配列を使う。
  - 日本語(漢字コード)を使う場合は (文字数)\*2 + 1 以上
  - +1 には文字の終了コード(  $\backslash 0$  ) が必要なため。

```
/* char_test.c : 文字列の定義と初期値代入 */  
#include <stdio.h>
```

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

```
    char date[9] = "Thursday";  
    char jdate[7] = "木曜日";
```

文字列の初期値代入

```
    printf(" %s は %s です\n", jdate, date);
```

```
    return 0;
```



```
}
```

日本語(漢字コード)は char の配列で表示可

# ファイルの入出力方法(1)

第10回の内容

- 共通の使い方

- `fopen()` でファイルを「開く」
  - ファイル名 `"filename"`
  - 書き込み (`w`) か 読み込み (`r`) か
  - テキストファイル か バイナリファイル (`b`) か
  - ファイルポインタを取得する
- `fclose()` でファイルを「閉じる」
  - `fopen()` で開いたファイルは必ず閉じる(重要)

```
FILE * fp ;           // ファイルポインタの宣言
fp = fopen("filename", "r" ) ;    // ファイルを開く
:
fclose( fp ) ;         // ファイルを閉じる
```

# ファイルの入出力方法(2)

- テキストファイル

- `fopen()` で開いたファイルに対して

- `fprintf()` でテキストデータを書き出す

```
fprintf( fp, "%d\n", i ) ;
```

- `fscanf()` でテキストデータを読み込む

```
fscanf( fp, "%d", &i ) ;
```

# 教科書 p.351 のプログラム(1)

第10回の内容

```
/*    file_test1.c    */

#define    _CRT_SECURE_NO_WARNINGS    1

#include    <stdio.h>

int    main(void) {

    FILE    * file;        // ファイルポインタ

    file = fopen( "test.txt" , "w"); // ファイルを開く
                                    // "w" で書き込み指定

    fclose(file);        // ファイルを閉じる

    return 0;
}
```

fopen() で開いたファイルは**必ず** fclose()  
で閉じる！(エラーは出ないが、危険)

# 実行結果

第10回の内容

実行前

実行

実行後

```
コマンドプロンプト
Microsoft Windows [Version 10.0.19042.1055]
(c) Microsoft Corporation. All rights reserved.

C:\Users\Nakano>cd C:\Users\Nakano\source\repos\file_test1\Debug

C:\Users\Nakano\source\repos\file_test1\Debug>dir
ドライブ C のボリューム ラベルは Windows です
ボリューム シリアル番号は 3427-890F です

C:\Users\Nakano\source\repos\file_test1\Debug のディレクトリ
2021/06/15  11:14    <DIR>          .
2021/06/15  11:14    <DIR>          ..
                39,936 file_test1.exe
                389,412 file_test1.ilc
                438,272 file_test1.pdb
                3 個のファイル             867,620 バイト
                2 個のディレクトリ  723,820,519,424 バイトの空き領域

C:\Users\Nakano\source\repos\file_test1\Debug>file_test1

C:\Users\Nakano\source\repos\file_test1\Debug>dir
ドライブ C のボリューム ラベルは Windows です
ボリューム シリアル番号は 3427-890F です

C:\Users\Nakano\source\repos\file_test1\Debug のディレクトリ
2021/06/15  11:16    <DIR>          .
2021/06/15  11:16    <DIR>          ..
                39,936 file_test1.exe
                389,412 file_test1.ilc
                438,272 file_test1.pdb
                0 test.txt
                4 個のファイル             867,620 バイト
                2 個のディレクトリ  723,819,995,136 バイトの空き領域

C:\Users\Nakano\source\repos\file_test1\Debug>
```

実行ファイル **file\_test1.exe** と同じディレクトリに  
test.txt というサイズ 0 のファイルが作成されている。

# 関数とは

これまで演習や課題でプログラムを作成したが、例えばこれらのプログラムを別のプログラム(main関数)でも利用したい場合を考えよう

- 必要な変数や処理部分をコピー＆ペーストをする

確かにこれで可能ではあるが、使いたいプログラムの数が多ければ多いほど、main関数の記述が膨大になり、煩雑なプログラムになる。

そこで、以前作成したプログラムの再利用を目的として、プログラムを再利用しやすい形に部品化を行う

プログラムでは、この部品のことを関数と呼ぶ

# ユーザー定義関数

- 実は関数の記述は既に行っている.
- main()もmain関数と呼ばれる関数である
- 記述方法はユーザー定義関数もmain関数と同様である

```
#include <stdio.h>
```

```
int main(void)
```

```
{
```

ここに処理を記述

```
return 0;
```

```
}
```

# ユーザー定義関数

第12回の内容

以下の様な消費税計算を行う関数を独立して作成する.

```
int tax (int kakaku)
{
    int zeikomi = 0; //初期化

    zeikomi = 1.08 * kakaku; //計算

    printf ( "%d¥n", zeikomi ); //表示

    return 0;
}
```



# 自作関数の記述位置

第12回の内容

## 1. main関数の後に記述

```
#include <stdio.h>
```

```
int main(void)
{
    return 0;
}
```

```
int tax(int kakaku)
{
    int zeikomi =0; //初期化
    zeikomi = 1.08*kakaku; //計算
    printf("%d¥n", zeikomi); //表示
    return 0;
}
```

## 2. main関数の前に記述

```
int tax(int kakaku)
{
    int zeikomi =0; //初期化
    zeikomi = 1.08*kakaku; //計算
    printf("%d¥n", zeikomi); //表示
    return 0;
}
```

```
#include <stdio.h>
```

```
int main(void)
{
    return 0;
}
```

記述の位置としてはどちらでもよいが、ただし1.の場合はこのままでは関数を使うことはできない

# ユーザー定義関数の使用

## 1. main関数の後に記述

```
#include <stdio.>
```

```
int tax(int); //プロトタイプ宣言
```

```
int main(void)
{
    return 0;
}
```

```
int tax(int kakaku)
{
    int zeikomi = 0; //初期化
    zeikomi = 1.08*kakaku; //計算
    printf("%d¥n", zeikomi); //表示
    return 0;
}
```

- 1.の場合main関数の前に作成した自作関数の簡単なリストを記述する必要がある
- このリスト記述を  
**プロトタイプ宣言**と呼ぶ
- プロトタイプ宣言により関数の位置を気にする必要がなくなるが、基本的にはmain関数の後ろに記述する方が見やすく良い

# ユーザー定義関数の使用

```
#include <stdio.>
```

```
int tax(int); //プロトタイプ宣言
```

```
int main(void)
```

```
{
```

```
    tax (100);
```

```
    return 0;
```

```
}
```

```
int tax(int kakaku)
```

```
{
```

```
    int zeikomi =0; //初期化
```

```
    zeikomi = 1.08*kakaku; //計算
```

```
    printf("%d¥n", zeikomi); //表示
```

```
    return 0;
```

```
}
```

- 自作関数を使用する場合はmain関数内で自作関数を呼び出す必要がある

- 関数に任意の値を渡す仕組みを**引数**という

## 実引数

main関数内で関数の呼び出しと同時に渡す値のことを指す

## 仮引数

自作関数で定義された変数の型と変数名のことを指す

# 戻り値を用いたプログラム例2

第12回の内容

プログラム例1との違いをきちんと理解すること！！

ソースコード

実行

```
1  #include <stdio.h>
2
3  int tax(int);
4
5  int main(void) {
6      int value = 0;
7      value = tax(100);
8      printf("%d\n", value);
9      return 0;
10 }
11
12 int tax(int kakaku)
13 {
14     int zeikomi = 0;
15     zeikomi = 1.08 * kakaku;
16     return zeikomi;
17 }
18
```

```
C:\Users\Nakano\source\repos\tax\Debug>tax
108
```

# 構造体を使う

- 変数のグループを作る (構造体)
  - 複数の変数をいくつかまとめて、一つの変数のように扱う。
  - 配列と考えが似ているが、異なった型の変数が混在していても良い。
    - まとめる変数は配列であっても、構造体であっても構わない
  - ただし、構造体内部の**変数名** (メンバー変数) と **型**、**順番**などの定義は関数側も呼び出し側も共通でなければならない。
    - 呼び出し側と関数を別ファイルに分割している場合は重要

# 構造体の定義

```
struct 構造体型名 {  
    変数の型 変数名 ;  
    変数の型 変数名 ;  
    :  
};
```

構造体型 stoch の定義

```
struct stoch {  
    int sum ; /* 合計 */  
    int ave ; /* 平均 */  
};
```

```
int ave1, sum1 ;  
struct stoch st1 ;
```

stoch 型の構造体 st1 を作成

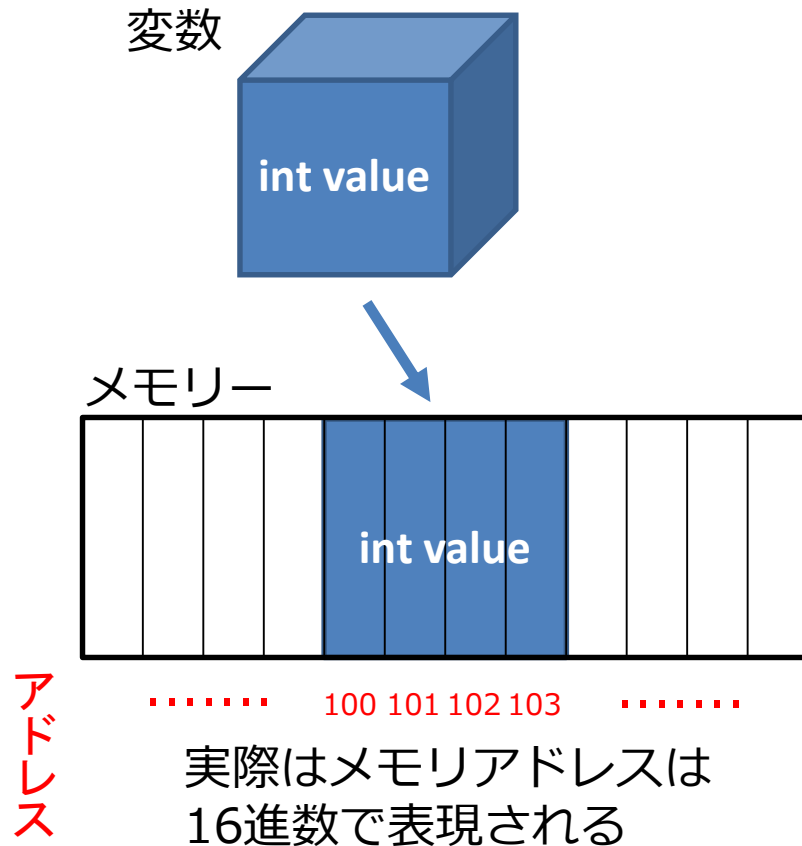
```
st1 = iaverage( n, data1[] ) ;  
ave1 = st1.ave ;
```

構造体 st1 のメンバー変数 ave  
をアクセスする

# アドレスとポインタ

第15回の内容

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
```



# ポインタの使用

## 第15回の内容

```
/* 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;
```

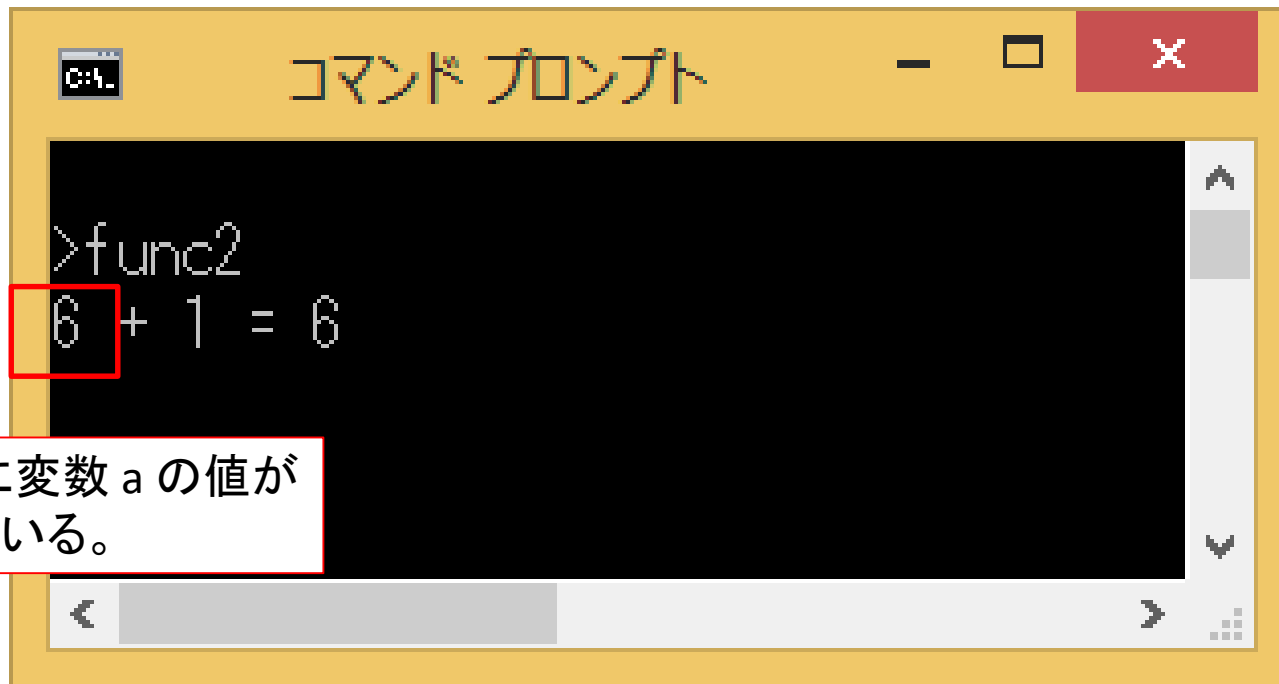
```
    return *p ;
```

```
}
```

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。

プログラミング作成にあたって

# プログラミング作成について

ここまでの講義の中で、皆さんにはC言語を通してプログラミングの基礎について学びました.

自分でコードを作成し自分で実行する, つまり作成したコードは基本的には, 自分以外は見ることがないことが前提でした.

# 良いコードとは？

まず以下のことを各自考えてみましょう。何人かあてて答えてもらいます。

1. 良いコードの定義として考えられることを最低3つ挙げよ。
2. 何故良いコードを書く必要があるのか？

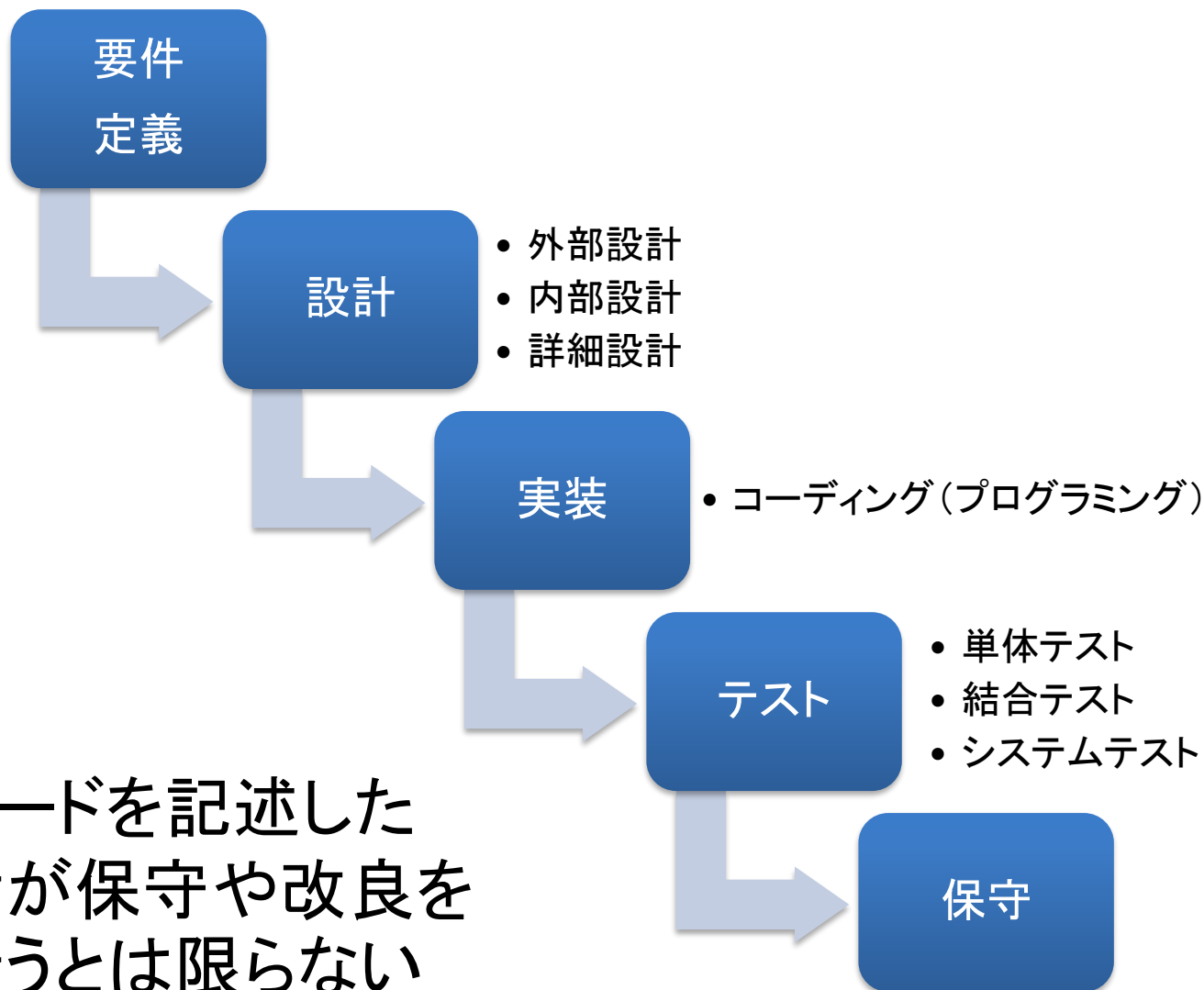
# 良いコード（プログラミング）とは

- 機能が正確に動作する.
- 常識の範囲の速度で動作する.
- コードが簡潔に分かりやすく書かれている.
  - ✓ 他人\*が見ても理解しやすい(可読性).
  - ✓ コードの改良がし易い(保守性).
  - ✓ 無駄な記述がない.

プログラミングの作法(規約)というものがある.

\*ここで言う他人とは, 未来の自分も含みます.

# 開発工程について



# プログラミング作法

プログラミングにも、文章や手紙のような作法がある。作法に従わなくてもプログラムは動く(文章なら内容を理解できる)。しかし、記述されたコードを本人以外理解することができない状況や、仮に他人が理解できていても時間がかかることになり開発や保守の効率が大幅低下する要因となる。そこで多くの場合、プログラミング作法(又は規約)に従って記述することが推奨される。

- 誰が見ても分かりやすいコードを記述するための作法(常識)がいくつかある。
- 作法に従わなくても、基本的にはコンパイルエラーにはならない。
- 個人で作成して保守する場合も作法にしたがって記述した方があとで(数日後～数年後)困らない。
- 独りよがりにならずに、読み手のことを考える。



# 作法の例

例1 下記のコードを見やすくするにはどうしたらよいか？

```
17 #include <stdio.h>
18
19 int selection(int *num, int idx);
20 int main(int argc, const char * argv[]) {
21     int idx=10;
22     int num[]={2,5,6,1,4,9,8,7,0,3};
23     selection(num, idx);
24     return 0;
25 }
26 int selection(int *num, int idx){
27     int temp_space=0;
28     int min_idx=0;
29     for (int i=0; i<idx; i++) {
30         min_idx=i;
31         for (int j=i+1; j<idx; j++) {
32             if ( num[min_idx]>num[j]) {
33                 min_idx=j;
34             }
35         }
36         temp_space=num[i];
37         num[i]=num[min_idx];
38         num[min_idx]=temp_space;
39     }
40     printf("Selection sort の結果は\n");
41     for (int i=0; i<idx; i++) {
42         printf("%d,", num[i]);
43     }
44     printf("\n");
45     return 0;
46 }
```

# 作法の例

例1 下記のコードを見やすくするにはどうしたらよいか？  
⇒必要な箇所に空白を入れる。

```
17 #include <stdio.h>
18
19 int selection(int *num, int idx);
20
21 int main(int argc, const char * argv[]) {
22     int idx = 10;
23
24     int num[] = {2,5,6,1,4,9,8,7,0,3};
25
26     selection(num, idx);
27
28     return 0;
29 }
30 }
```

```
32 int selection(int *num, int idx){
33
34     int temp_space = 0;
35     int min_idx    = 0;
36
37     for (int i = 0; i < idx; i++) {
38
39         min_idx = i;
40
41         for (int j = i + 1; j < idx; j++) {
42
43             if ( num[min_idx] > num[j]) {
44
45                 min_idx = j;
46             }
47         }
48
49         temp_space    = num[i];
50         num[i]        = num[min_idx];
51         num[min_idx]  = temp_space;
52     }
53
54
55     printf("Selection sort の結果は\n");
56     for (int i = 0; i < idx; i++) {
57
58         printf("%d,", num[i]);
59     }
60     printf("\n");
61
62     return 0;
63 }
```

# 作法の例

例2 下記のコードをより見やすくするにはどうしたらよいか？

```
17 #include <stdio.h>
18
19 int selection(int *num, int idx);
20
21 int main(int argc, const char * argv[]) {
22     int idx = 10;
23
24     int num[] = {2,5,6,1,4,9,8,7,0,3};
25
26     selection(num, idx);
27
28     return 0;
29 }
30 }
```

```
32 int selection(int *num, int idx){
33
34     int temp_space = 0;
35     int min_idx = 0;
36
37     for (int i = 0; i < idx; i++) {
38
39         min_idx = i;
40
41         for (int j = i + 1; j < idx; j++) {
42
43             if ( num[min_idx] > num[j]) {
44
45                 min_idx = j;
46             }
47         }
48
49         temp_space = num[i];
50         num[i] = num[min_idx];
51         num[min_idx] = temp_space;
52     }
53
54
55     printf("Selection sort の結果は\n");
56     for (int i = 0; i < idx; i++) {
57
58         printf("%d,", num[i]);
59     }
60     printf("\n");
61
62     return 0;
63 }
```

# 作法の例

例2 下記のコードをより見やすくするにはどうしたらよい？  
⇒位置合わせ

```
17 #include <stdio.h>
18
19 int selection(int *num, int idx);
20
21 int main(int argc, const char * argv[]) {
22     int idx = 10;
23
24     int num[] = {2,5,6,1,4,9,8,7,0,3};
25
26     selection(num, idx);
27
28     return 0;
29 }
30
```

```
32 int selection(int *num, int idx){
33
34     int temp_space = 0;
35     int min_idx    = 0;
36
37     for (int i = 0; i < idx; i++) {
38         min_idx = i;
39
40         for (int j = i + 1; j < idx; j++) {
41             if ( num[min_idx] > num[j]) {
42                 min_idx = j;
43             }
44         }
45
46         temp_space = num[i];
47         num[i]     = num[min_idx];
48         num[min_idx] = temp_space;
49     }
50
51     printf("Selection sort の結果は\n");
52     for (int i = 0; i < idx; i++) {
53         printf("%d,", num[i]);
54     }
55     printf("\n");
56
57     return 0;
58 }
59
60
61
62
63
```

# 作法の例

例3 下記のコードをより見やすくするにはどうしたらよいか？

```
17 #include <stdio.h>
18
19 int selection(int *num, int idx);
20
21 int main(int argc, const char * argv[]) {
22
23     int idx = 10;
24
25     int num[] = {2,5,6,1,4,9,8,7,0,3};
26
27     selection(num, idx);
28
29     return 0;
30 }
```

```
32 int selection(int *num, int idx){
33
34     int temp_space =0;
35
36     int min_idx = 0;
37
38     for (int i = 0; i< idx; i++) {
39
40         min_idx = i;
41
42         for (int j = i + 1; j < idx; j++) {
43
44             if ( num[min_idx] > num[j]) {
45
46                 min_idx = j;
47             }
48         }
49
50         temp_space  = num[i];
51         num[i]      = num[min_idx];
52         num[min_idx] = temp_space;
53     }
54
55     printf("Selection sort の結果は\n");
56     for (int i = 0; i < idx; i++) {
57
58         printf("%d,", num[i]);
59     }
60     printf("\n");
61     return 0;
62 }
```

# 作法の例

例3 下記のコードをより見やすくするにはどうしたらよいか？  
⇒字下げ(インデント)

```
17 #include <stdio.h>
18
19 int selection(int *num, int idx);
20
21 int main(int argc, const char * argv[]) {
22
23     int idx = 10;
24
25     int num[] = {2,5,6,1,4,9,8,7,0,3};
26
27     selection(num, idx);
28
29     return 0;
30 }
```

```
32 int selection(int *num, int idx){
33
34     int temp_space = 0;
35
36     int min_idx = 0;
37
38     for (int i = 0; i < idx; i++) {
39
40         min_idx = i;
41
42         for (int j = i + 1; j < idx; j++) {
43
44             if ( num[min_idx] > num[j]) {
45
46                 min_idx = j;
47             }
48         }
49
50         temp_space = num[i];
51         num[i] = num[min_idx];
52         num[min_idx] = temp_space;
53     }
54
55     printf("Selection sort の結果は\n");
56     for (int i = 0; i < idx; i++) {
57
58         printf("%d,", num[i]);
59     }
60     printf("\n");
61
62     return 0;
63 }
```

# 作法の例

例4 下記のコードをより見やすくするにはどうしたらよいか？

```
17 #include <stdio.h>
18
19 int selection(int *a, int b);
20
21 int main(int argc, const char * argv[]) {
22
23     int b = 10;
24
25     int a[] = {2,5,6,1,4,9,8,7,0,3};
26
27     selection(a, b);
28
29     return 0;
30 }
```

```
32 int selection(int *a, int b){
33
34     int c =0;
35
36     int d = 0;
37
38     for (int i = 0; i< b; i++) {
39
40         d = i;
41
42         for (int j = i + 1; j < b; j++) {
43
44             if ( a[d] > a[j]) {
45
46                 d = j;
47             }
48         }
49
50         c = a[i];
51         a[i] = a[d];
52         a[d] = c;
53     }
54
55     printf("Selection sort の結果は\n");
56     for (int i = 0; i < b; i++) {
57
58         printf("%d,",a[i]);
59     }
60     printf("\n");
61
62     return 0;
63 }
```

# 作法の例

例4 下記のコードをより見やすくするにはどうしたらよいか？  
⇒ユーザが定義する変数, 関数, 定数, クラスに適切な命名を行う.

```
17 #include <stdio.h>
18
19 int selection(int *num, int idx);
20
21 int main(int argc, const char * argv[]) {
22     int idx = 10;
23
24     int num[] = {2,5,6,1,4,9,8,7,0,3};
25
26     selection(num, idx);
27
28     return 0;
29 }
30 }
```

```
32 int selection(int *num, int idx){
33
34     int temp_space = 0;
35
36     int min_idx = 0;
37
38     for (int i = 0; i < idx; i++) {
39         min_idx = i;
40
41         for (int j = i + 1; j < idx; j++) {
42             if ( num[min_idx] > num[j]) {
43                 min_idx = j;
44             }
45         }
46
47         temp_space = num[i];
48         num[i] = num[min_idx];
49         num[min_idx] = temp_space;
50     }
51
52     printf("Selection sort の結果は\n");
53     for (int i = 0; i < idx; i++) {
54         printf("%d,", num[i]);
55     }
56     printf("\n");
57
58     return 0;
59 }
60
61
62
63 }
```



# 名前の決め方について

自分で定義した変数，関数の名前を決めるのもセンスが問われるが，一般的には，以下のようなことを目安に考えるとよい．

- 命名には記述したコード内で統一したルールを用いて一貫性を持たせること．
- 関数にはどのような処理（動作）を行うかが分かるような名前をつける．
- 変数の有効範囲（スコープ）に応じて名前を付ける．
  - ✓ グローバル変数の場合，変数名から内容が説明されている名前．
  - ✓ ローカル変数の場合，短く簡潔な名前でもよい．
- 無駄なく簡潔に命名する．
- 自分のコードを読む際に必要な情報を与えること．

# 作法の例

例5 下記のコードの作成者以外が理解するためにコメントをつけてみよう

```
17 #include <stdio.h>
18
19 int selection(int *num, int idx);
20
21 int main(int argc, const char * argv[]) {
22
23     int idx = 10;
24
25     int num[] = {2,5,6,1,4,9,8,7,0,3};
26
27     selection(num, idx);
28
29     return 0;
30 }
```

```
32 int selection(int *num, int idx){
33
34     int temp_space = 0;
35
36     int min_idx = 0;
37
38     for (int i = 0; i < idx; i++) {
39
40         min_idx = i;
41
42         for (int j = i + 1; j < idx; j++) {
43
44             if ( num[min_idx] > num[j]) {
45
46                 min_idx = j;
47             }
48         }
49
50         temp_space = num[i];
51         num[i] = num[min_idx];
52         num[min_idx] = temp_space;
53     }
54
55     printf("Selection sort の結果は\n");
56     for (int i = 0; i < idx; i++) {
57
58         printf("%d,", num[i]);
59     }
60     printf("\n");
61
62     return 0;
63 }
```

# 作法の例

## 例5

下記のコードをより見やすくするにはどうしたらよいか？

⇒読み手の助けになるようなコメントを記述する.

```
17 #include <stdio.h>
18
19 //プロトタイプ宣言
20 int selection(int *num, int idx);
21
22 int main(int argc, const char * argv[]) {
23
24     //配列の要素数
25     int idx = 10;
26
27     //配列の確保
28     int num[] = {2,5,6,1,4,9,8,7,0,3};
29
30     //Bubble sort関数の呼び出し
31     selection(num, idx);
32
33     return 0;
34 }
```

```
36 int selection(int *num, int idx){
37
38     //一時的な退避用の領域確保
39     int temp_space = 0;
40     //最小値を持つ添え字
41     int min_idx = 0;
42
43     for (int i = 0; i < idx; i++) {
44         //比較対象の要素
45         min_idx = i;
46         //最小値をもつ添え字との比較
47         for (int j = i + 1; j < idx; j++) {
48
49             if ( num[min_idx] > num[j]) {
50
51                 //より最小な値を持つ要素の添え字の保存
52                 min_idx = j;
53             }
54         }
55         //交換
56         temp_space = num[i];
57         num[i] = num[min_idx];
58         num[min_idx] = temp_space;
59     }
60
61     //SORTされたかを確認
62     printf("Selection sort の結果は\n");
63     for (int i = 0; i < idx; i++) {
64
65         printf("%d,", num[i]);
66     }
67     printf("\n");
68
69     return 0;
70 }
```

# コメントについて

- コードの内容を正確に記述すること.
- 簡潔に分かりやすく.
- 関数やグローバル変数にはコメントがあれば理解の助けになる.
- コードとは関係ないことを書かない.

# 参考文献

題目: プログラミング作法 THE PRACTICE OF PROGRAMMING

著者: Brinan W. Kermighan, Rob Pike, 福崎 俊博 訳

出版社: アスキー

出版年: 2000年

ISBN-13: 978-4756136497