

# 制御文

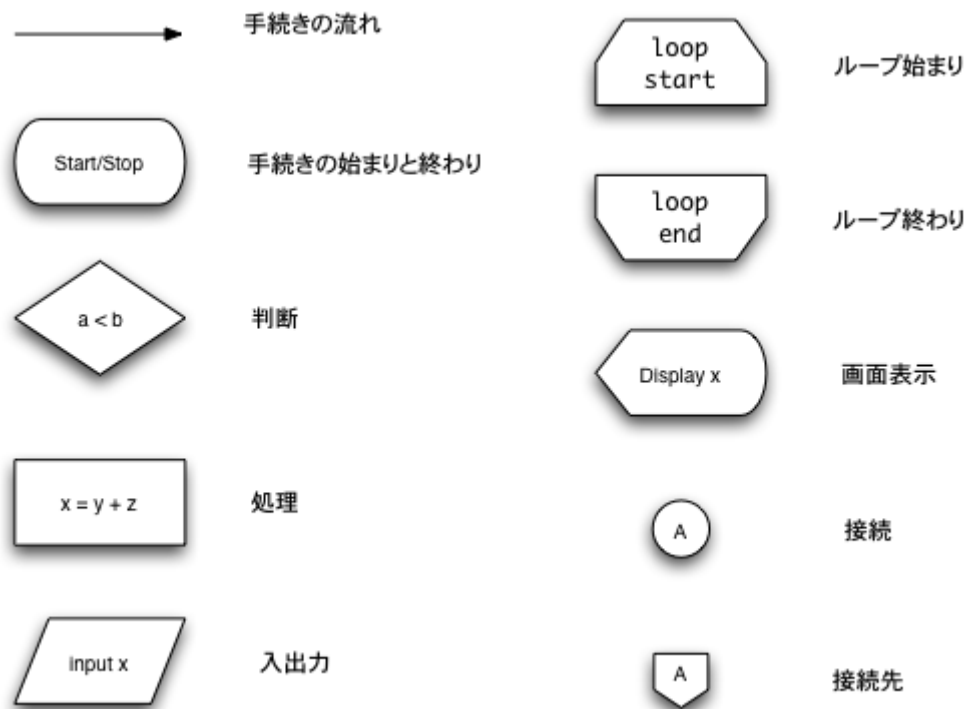
プログラムの実行(流れ)の制御を行う文を, ここでもう一度詳しく解説する.

- [フローチャート](#)
- [条件式](#)
  - [演習問題1](#)
- [if 文](#)
- [switch 文](#)
- [for 文](#)
  - [演習問題2](#)
  - [演習問題3](#)
- [while 文](#)
- [do - while 文](#)
- [無限ループ](#)
- [よくある間違い](#)
- [レポート課題](#)

## フローチャート

フローチャート(流れ図)は, プログラムの実行手続きの流れを図によって表示するものである. その書き方は JIS 規格で決まっている. JIS X 0121:1986 [情報処理用流れ図・プログラム網図・システム資源図記号](#)

## フローチャートで用いる部品



## フローチャートの例

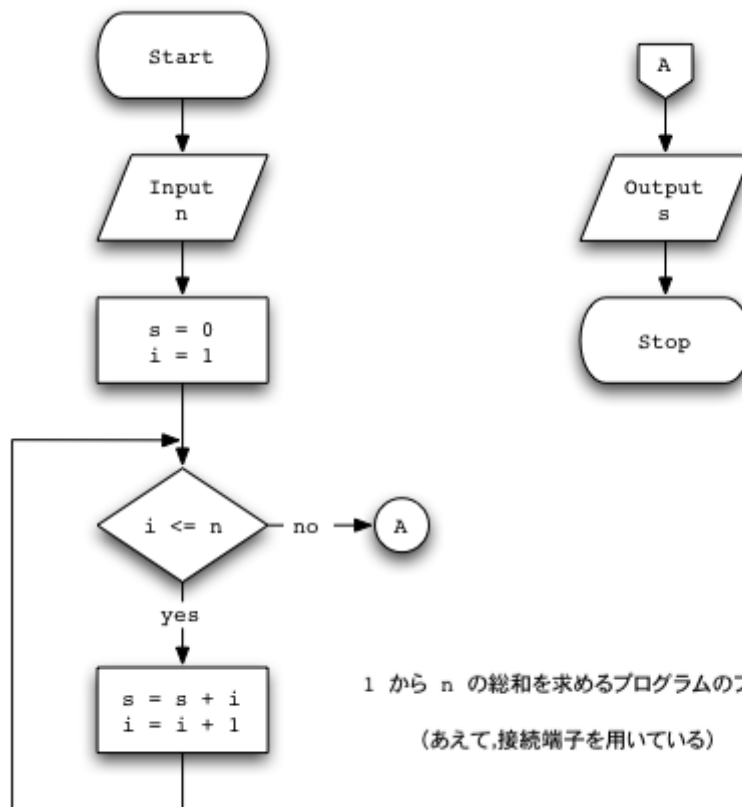
1 から n までの整数の和を求めるプログラムと、そのフローチャートとを見比べてみる。

```
#include <stdio.h>

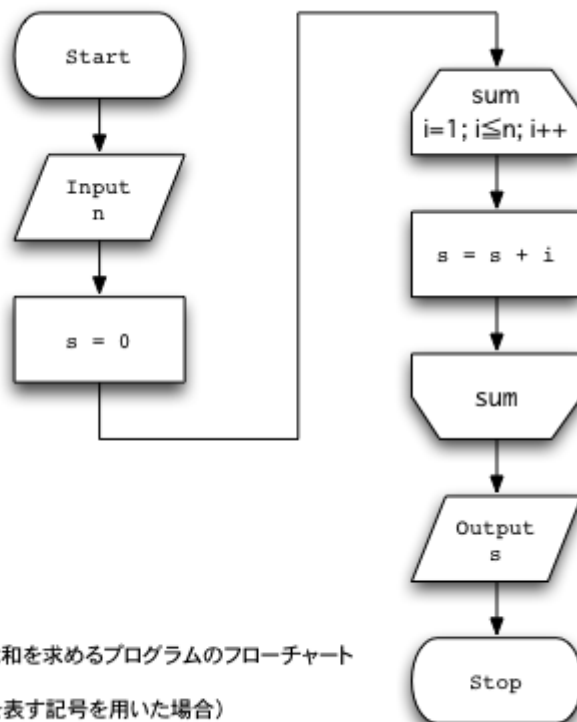
int main(void)
{
    int n, i, s;

    scanf("%d", &n);
    s = 0;
    for (i = 1; i <= n; i++) {
        s += i;
    }
}
```

```
}  
printf("%d\n", s);  
}
```



1 から n の総和を求めるプログラムのフローチャート  
(あえて, 接続端子を用いている)



1 から n の総和を求めるプログラムのフローチャート  
(ループを表す記号を用いた場合)

## 条件式

制御されたプログラムの流れは, 条件式が成り立つかどうかで決まる.

C言語では条件式が必要な主な場所は次の通り.

if (条件式)

for (...; 条件式; ...)

while (条件式)

do { ... } while (条件式)

## 条件式の書き方

条件式には、一致 `==`、不一致 `!=`、大小比較 `<`, `>`, `<=`, `>=` を用いることができる。（`!=` や `=<` や `=>` は間違いである。）

```
if ( a == b ) ...
```

```
if ( a >= b ) ...
```

“かつ `&&`” や “または `||`” を含む条件式も書ける。

```
if ( ( a == b ) && ( b > c ) ) ...
```

`&&`, `||` を用いるときには、間違いがないよう、括弧 `()` で括って書くようにする。

```
if ( (( a == b ) || ( b > c )) && ( c == d ) ) ...
```

間違いやすいことなので注意しておくが、C言語では `a < b < c` や `a == b <= c` のような書き方はできない。（コンパイルエラーにはならないが、期待した結果には決してならない）

```
if ( a < b < c ) ...   間違い
```

このようなものは、2つに分けて次のように書くべきである。

```
if ( ( a < b ) && ( b < c ) ) ...
```

条件式 `A` の結果を否定したものを、条件式とすることもできる。その場合には、条件式 `A` の先頭に `!` を付けて `!(A)` とする。

```
if ( ! ( a >= b ) ) ...
```

## 真偽値

条件式を判断した結果を真偽値という。多くの高級言語には真偽値を表すための特別な型 (boolean 型) があるが、C言語ではそのための特別な型はなく真偽値は int 型で表される。0でない整数値が“真”を意味し、整数値0が“偽”を意味する。

1, 2, 3, ..., -1, -2, ...	= 真
0	= 偽

極端な話、次の if 文の判断文は常に真となり、また次の while 文は終わることのない無限ループとなる。

```
if (-2) ...  
while (1) ...
```

整数値そのものを条件式とすることもできる。(以下の例では、a は整数値(あるいはint型変数)とする)

```
if ( a ) ...  
                                     これは次と同じ  
if ( a != 0 )  
if ( !a ) ...  
                                     これは次と同じ  
if ( a == 0 )
```

条件式の結果は整数値であるということを、次のプログラムで確認してみよう。

```
main()  
{  
    printf("(1 < 2) == %d\n", 1 < 2);  
    printf("(1 > 2) == %d\n", 1 > 2);  
}
```

```
(1 < 2) == 1  
(1 > 2) == 0
```

一致の条件式 `a == b` を書くべき所に間違えて `a = b` と書いたときに、`a, b` の型が整数型であればコンパイルエラーとはならない(警告は出るかも知れない)。それは、`a` に `b` の値を代入して、その代入した値(整数値)を真偽値とみなして、判断がなされる。すなわち `0` でなければ真であり `0` であれば偽となる。

```
a = 1;
b = 2;
if (a = b) {
    ....
}
```

上の `if` 文では、まず `a` に `b` の値が代入される。そしてその値は `2` であるので、条件式が真となり、`....` の部分が実行される。

`a != b` と書くべき時に間違えて `a =! b` と書いたときも、`a, b` が整数型であればコンパイルエラーとはならない(警告は出るかも知れない)。それは `a = !b` と解釈される。

```
a = 1;
b = 2;
if (a =! b) {
    ....
}
```

`a` に `!b` の値、すなわち `b` の値が `0` ではないから(真)その否定(偽)を表す `0` が代入される。`if` 文の条件式の値は `0` (偽)となり、`....` の部分は実行されない。

```
if ( (a < b) && (b < c) ) ....
```

と書くべきなのに、次のように書いてしまう間違いが多い。

```
if ( (a < b < c) ) ....
```

このような誤った書き方をした場合でも、コンパイルエラーにはならない。そして、次のように解釈される。

```
if ( (a < b) < c ) ....
```

その意味は、「まず  $a < b$  を判断してその結果(値は 0 か 1 か)が  $c$  の値よりも小さいかどうかを判断する」ということになる。これは明らかにプログラマーが意図するものとは違う。

## 演習問題

閏年は、西暦年が 4 で割り切れてしかも 100 では割り切れないか、あるいは 400 で割り切れる年である。

西暦  $y$  年が閏年であるかどうかの真偽値を返す関数 `int urudosi(int y)` を作れ。(閏年のときは 1 を、閏年でないときは 0 を返すこととする。)

ヒント:  $a$  を  $b$  で割った余りは  $a \% b$  である。

```
int urudosi(int y)
{
    ....
}

int main()
{
    int y;

    y = 2000;
    printf("%d %d\n", y, urudosi(y));

    y = 2001;
    printf("%d %d\n", y, urudosi(y));

    y = 2004;
    printf("%d %d\n", y, urudosi(y));

    y = 2100;
    printf("%d %d\n", y, urudosi(y));
}
```



```
return 0;  
}
```

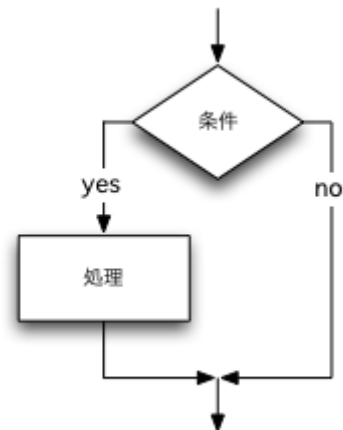
```
2000 1  
2001 0  
2004 1  
2100 0
```

## if 文

条件が満たされているかいないかで、処理を場合分けするときに用いる。

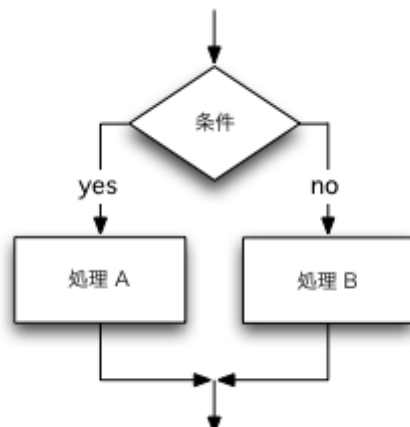
条件が満たされているときだけに、処理をしたいとき。

```
if (条件式) {  
    条件が成立するときの処理  
}  
if (a < 10) {  
    b = c + d;  
}
```



条件が満たされているときと満たされていないときとで、違った処理がしたいとき。

```
if (条件式) {  
    条件が成立するときの処理 A  
} else {  
    条件が成立しないときの処理 B  
}  
if (a < 10) {
```



```

        b = c + d;
    } else {
        b = c - d;
    }

```

if 文の中で行う処理が 1 行で終わるときには, {} で括らずに, 次のような書き方もできる.

```

if (a < 10)
    b = c + d;      推奨しない
if (a < 10)
    b = c + d;      推奨しない
else
    b = c - d;      推奨しない

```

しかし, この書き方はバグの誘因となるので, 止めておくべきである.

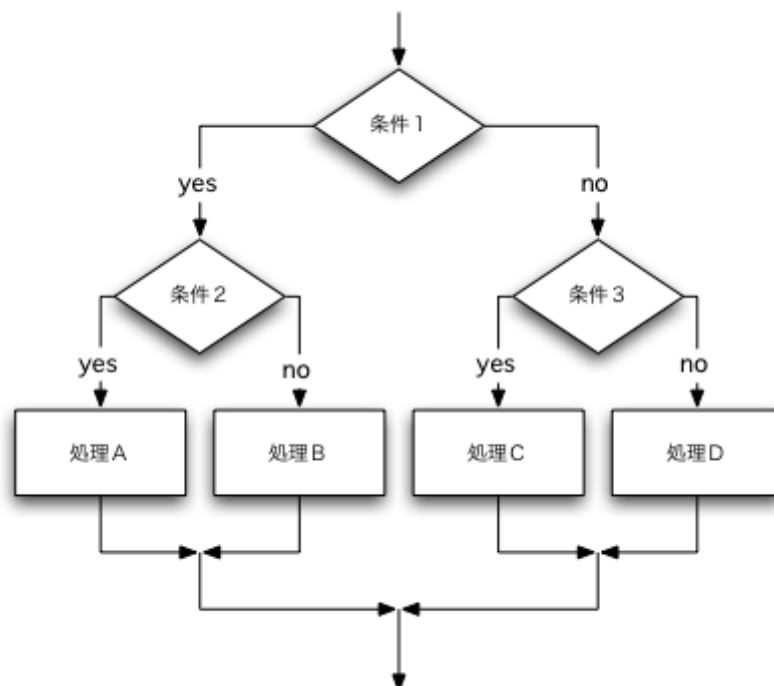
## 多重 if 文

if 文の中に, さらに if 文を含めることもできる.

```

if (条件式 1)
{
    if (条件式 2) {
        処理 A
    } else {
        処理 B
    }
} else {

```



```

    if (条件式 3) {
        処理 C
    } else {
        処理 D
    }
}

```

多重 if 文を書くときには、処理が 1 行しか無くても {} で括って、さらにきちんと段下げ(行頭を右に数文字ずらすこと)を行うべきである。 そうしないと、プログラムが大変読みにくなる。

多重 if 文の特殊な形として、else if の繰り返しがある。

「この場合はこう、そうでなくてこの場合はこう、そうでなくてこの場合はこう、..., そうでない場合はこう」という場合分けに用いる

```

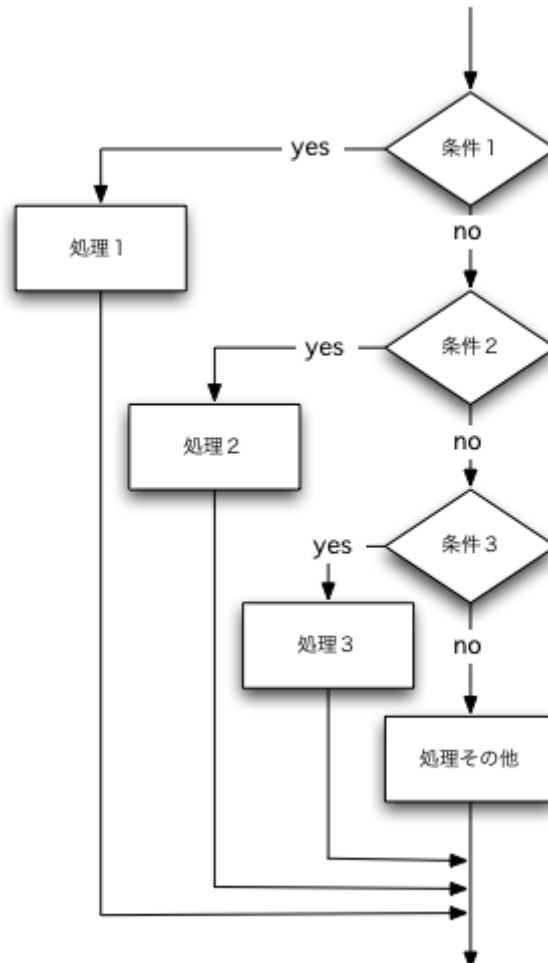
if (条件式 1) {
    処理 1
} else if (条件式 2) {
    処理 2
} else if (条件式 3) {
    処理 3
} else {
    処理その他
}

int main()
{
    double x;

    x = 1.5;

    if (x < 0) {
        printf("%f < 0\n", x);
    }
}

```



```

    } else if (x < 1) {
        printf("0 <= %f < 1¥n", x);
    } else if (x < 2) {
        printf("1 <= %f < 2¥n", x);
    } else {
        printf("2 <= %f¥n", x);
    }

    return 0;
}

```

1 <= 1.500000 < 2

## switch 文

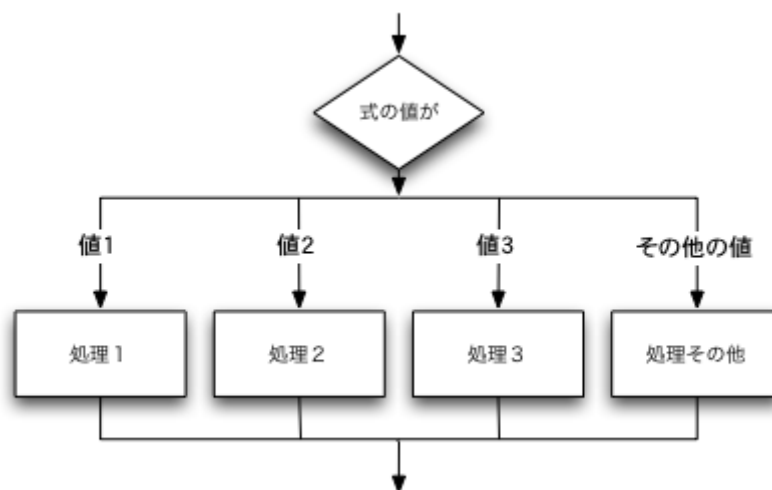
一つの式の値を場合分けのキーとして、多数の場合に分けて処理を行うときには、switch 文を用いる。

次のような switch 文の場合、値1、値2、値3の中で「式」の値と一致するものがあれば、そこに書かれている処理が行われる。値1、値2、値3の中には「式」の値と一致するものがなければ、default のところに書かれている処理が行われる。

```

switch (式) {
case 値1:
    処理1
    break;
case 値2:
    処理2
    break;
case 値3:
    処理3
    break;
default:
    処理その他
}

```



```
    break;
}
```

“case 値:” を case ラベルと呼ぶ。

case ラベルに続く各処理の最後には break を付ける必要がある。break はそこで switch 文を終了するという意味になる。(break を付け忘れると、その処理が終わった後 switch 文を終了せずその次の case ラベルにある処理に入ってしまう。)

```
int main()
{
    int i;

    for (i = 1; i <= 5; i++) {

        switch (i) {
        case 1:
            printf("case 1¥n");
            break;
        case 3:
            printf("case 3¥n");
            break;
        case 5:
            printf("case 5¥n");
            break;
        default:
            printf("other case¥n");
            break;
        }

    }

    return 0;
}
```

```
case 1
other case
```

```
case 3
other case
case 5
```

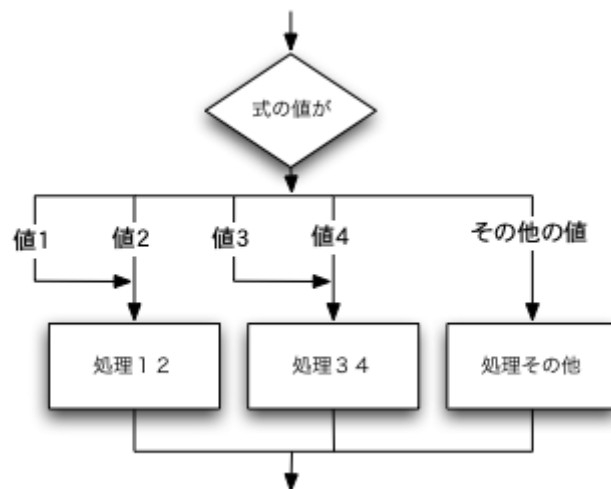
一つの処理に複数の case ラベルを付けることもできる.

```
switch (式) {
case 値 1 :
case 値 2 :
    処理 1 2
    break;
case 値 3 :
case 値 4 :
    処理 3 4
    break;
default :
    処理その他
    break;
}

int main()
{
    int i;

    for (i = 1; i <= 5; i++) {

        switch (i) {
        case 1:
        case 2:
            printf("case 1 or 2¥n");
            break;
        case 3:
        case 4:
            printf("case 3 or 4¥n");
            break;
```



```

        default:
            printf("other case¥n");
            break;
    }

}

return 0;
}

```

```

case 1 or 2
case 1 or 2
case 3 or 4
case 3 or 4
other case

```

## for 文

for 文は、パラメータを一定数ずつ増やし(減らし)ながら、決まった回数だけ処理を行いたいときに、よく使われる。

```

for ( パラメータ初期化 ; 繰り返し条件 ; パラメータの増減 ) {
    処理
}

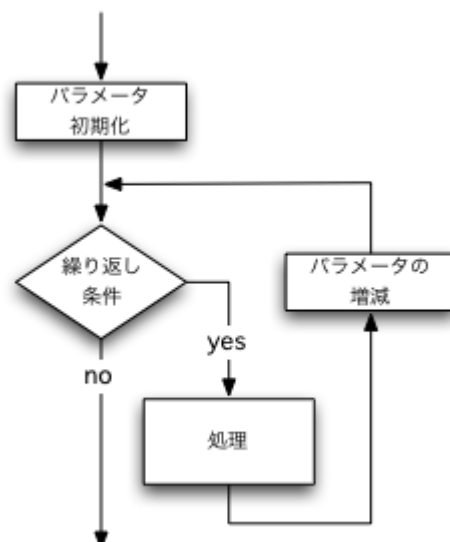
```

パラメータ初期化, 繰り返し条件, パラメータの増減, 処理は, 右図のような順序で行われることに注意しておこう。要注意点は

―― 繰り返し条件の判断は処理に先立って行われる ――

ということである。始めから繰り返し条件が満たされないときには, 処理は一度も行われない。

```
int main()
```



```

{
    int i;

    for ( i = 0 ; i < 5 ; i ++ ) {
        printf("%d¥n", i);
    }

    return 0;
}

```

```

0
1
2
3
4

```

for 文は、決まった回数だけ処理を行うためだけではなく、条件が成立する限り処理を繰り返すようなときにも使える。

また、パラメータの増減は `i++` や `i--` だけではなく、パラメータが変化するようなものならどんな式でも書ける。

さらに、パラメータ初期化、パラメータの増減の場所には、カンマ “,” で区切ることで、式を何個でも書くことができる。

```

int main()
{
    int a, b;

    for ( a = 1, b = 1024 ; a <= b ; a *= 2, b /= 2 ) {
        printf("(%d, %d)¥n", a, b);
    }

    return 0;
}

```

```

(1, 1024)

```



```
(2, 512)
(4, 256)
(8, 128)
(16, 64)
(32, 32)
```

for 文の処理が一つの実行文であるときには、処理を {} で括らず、例えば次のような形で書くことができる。

```
for ( i = 0 ; i < 10 ; i++ )
    printf("%d", i);
```

## 演習問題2

for 文を用いて、関数 total と max とを作れ。

```
int total (int a[], int n) の関数仕様
    int 型配列 a 内の, a[0] から a[n-1] の n 個の要素の総和を返す.

int max (int a[], int n) の関数仕様
    int 型配列 a 内の, a[0] から a[n-1] の n 個の要素中の最大値を返す.
```

```
int total(int a[], int n)
{
    ....
}

int max(int a[], int n)
{
    ....
}

int main()
```

```
{
    int data[10] = {3, 4, 5, 8, 2, 4, 1, 9, 3, 6};

    printf("total = %d\n", total(data, 10));
    printf("max = %d\n", max(data, 10));
}
```

total = 45

max = 9

## 多重 for ループ

for 文の処理の中に, for 文を含めることができる.

```
for (パラメータ 1 の初期化 ; 繰り返し条件 1 ; パラメータ 1 の増減 ) {
```

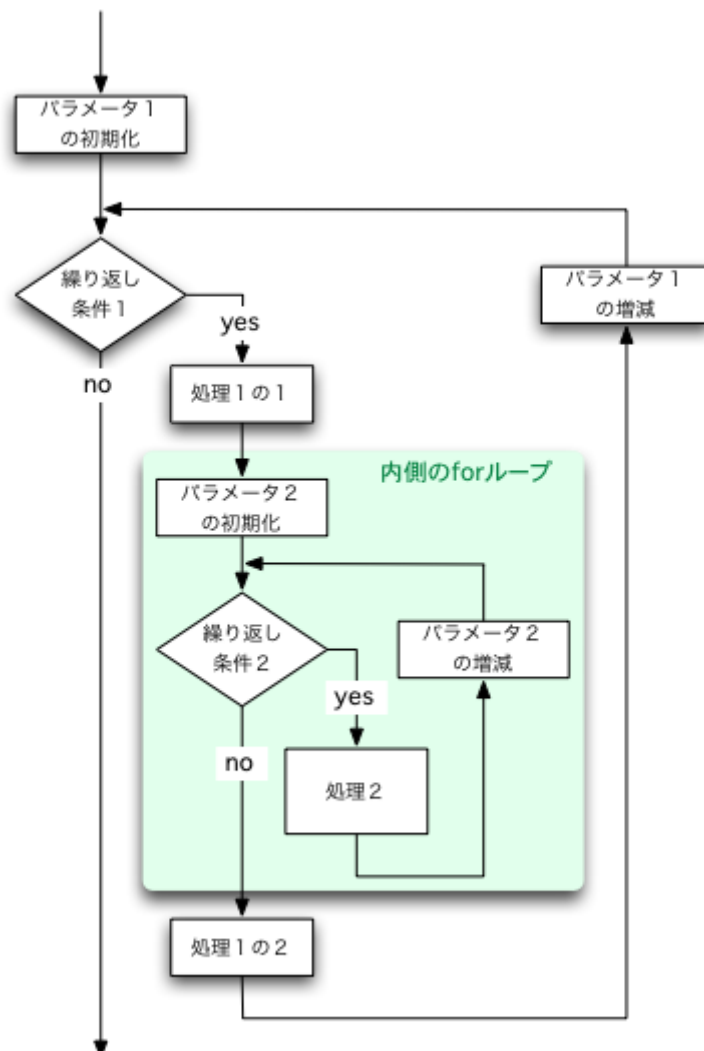
```
    処理 1 の 1
```

```
        for (パラメータ
            2 の初期化 ; 繰り返し
            条件 2 ; パラメータ 2
            の増減 ) {
                処理 2
            }
    }
```

```
    処理 1 の 2
```

```
}
int main()
{
    int i, j;

    for (i = 1; i <= 9; i++)
    {
        printf("%2d |", i);
    }
}
```



```

    for (j = 1; j <= 9; j++) {
        printf("%4d", i*j);
    }
    printf("\n");
}

return 0;
}

```

1	1	2	3	4	5	6	7	8	9
2	2	4	6	8	10	12	14	16	18
3	3	6	9	12	15	18	21	24	27
4	4	8	12	16	20	24	28	32	36
5	5	10	15	20	25	30	35	40	45
6	6	12	18	24	30	36	42	48	54
7	7	14	21	28	35	42	49	56	63
8	8	16	24	32	40	48	56	64	72
9	9	18	27	36	45	54	63	72	81

## continue 文による処理のスキップ

for 文中の処理を実行中に continue 文に行き当たったときには、その回のその後の処理がスキップされる。そして次の回の処理に入る。

```

for (パラメータの初期化 ; 繰り返し条件 ; パラメータの増減 ) {
    処理前半
    if (スキップ条件) continue;
    処理後半
}

```

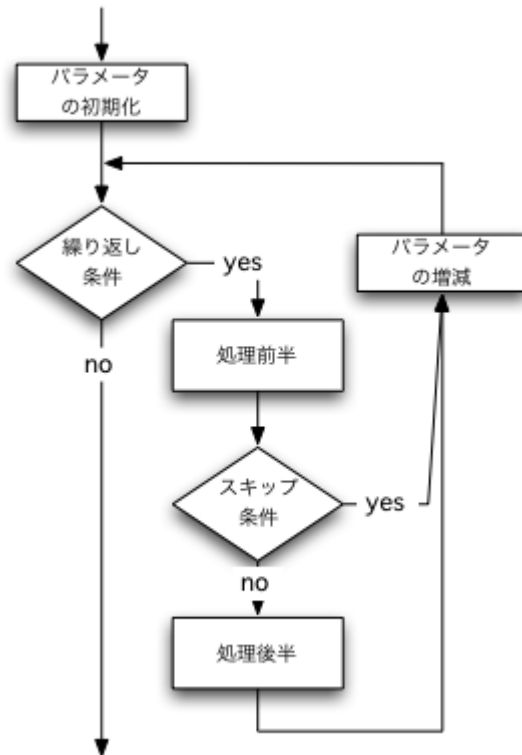
もちろん、パラメータの増減、繰り返し条件の判断を行い、それが真であれば次の回の処理に入る。

```
int main()
{
    int i;

    for (i = 1; i <= 5; i++) {
        printf("%d¥n", i);
        if (i == 3) continue;
        printf("-----¥n");
    }

    return 0;
}
```

```
1
-----
2
-----
3
4
-----
5
-----
```



多重 for ループの中で continue 文に行き当たったときには、スキップされるのはその continue 文を含む一番内側の for 文の処理だけである。

```
for ( ... ; ... ; ... ) {
    .....
    for ( ... ; ... ; ... ) {
        .....
        if (...) continue;
        .....
    }
}
```

```

    }
    .....
}

```

### 演習問題3

```

int main()
{
    int i, j;

    for (i = 1; i <= 9; i++) {
        printf("%2d |", i);
        for (j = 1; j <= 9; j++) {
            printf("%4d", i*j);
        }
        printf("\n");
    }

    return 0;
}

```

上のプログラム中に `continue` 文を二つ入れて、次のような出力になるようにせよ。（5の行と7の列とがない）

1	1	2	3	4	5	6	8	9
2	2	4	6	8	10	12	16	18
3	3	6	9	12	15	18	24	27
4	4	8	12	16	20	24	32	36
6	6	12	18	24	30	36	48	54
7	7	14	21	28	35	42	56	63
8	8	16	24	32	40	48	64	72
9	9	18	27	36	45	54	72	81

break 文による for 文の中断

for 文中の処理を実行中に break 文に行き当たったときには、直ちに for 文の処理を中断して、for 文を終了する。

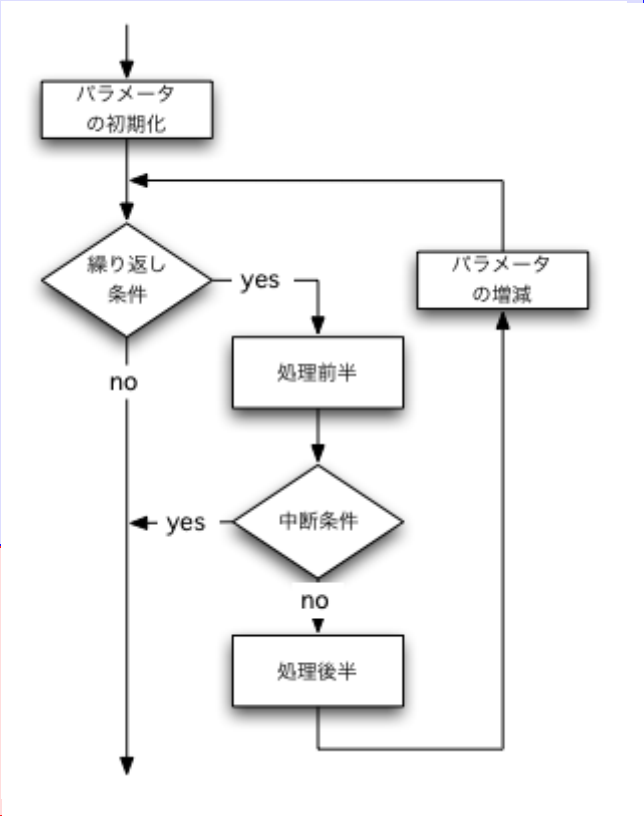
```
for (パラメータの初期化 ; 繰り返し条件 ; パラメータの増減 ) {
    処理前半
    if (中断条件) break;
    処理後半
}
```

```
int main()
{
    int i;

    for (i = 1; i <= 5; i++) {
        printf("%d¥n", i);
        if (i == 3) break;
        printf("-----¥n");
    }

    return 0;
}
```

- 1
- 2
- 3



多重 for ループの中で break 文に行き当たったときには、中断されるのはその break 文を含む一番内側の for ループである。

```
for ( ... ; ... ; ... ) {  
    .....  
    for ( ... ; ... ; ... ) {
```

```

.....
    if (...) break;
.....
}
.....
}

```

## break 文のちょっと高度な使い方

break 文は、何かを見つけるための繰り返しループにおいて、目的のものを見つけた時点でループを中断するときに、よく使われる。

```

#include <stdio.h>
#define N 10

int main()
{
    int data[N] = {1, 3, 4, 5, 7, 0, 3, 3, 2, 5};
    int i, val;

    val = 7;                                // val の値をもつ要素を探す

    for (i = 0; i < N, i++) {
        if (a[i] == val) break;             // 目的の値が見つかったら直ちに for 文を終了
    }

    if (i < N) {                             // 見つかった場合
        printf("data[%d] == %d¥n", i, val);
    } else {                                 // 見つからなかった場合
        printf("no data %d¥n", val);
    }
}

```

data[4] == 7

break を含む for 文が終了した後で、「その for 文が break 文で中断したか、ループが最後まで回って終了したか」を判断したいときには、その for 文の繰り返し条件が成立しているかどうかを改めて判断すればよい。繰り返し条件が成立している場合、それは break 文で中断している。

## while 文

while 文は、特定の条件が満たされる限り繰り返し処理を行いたいときに用いる。

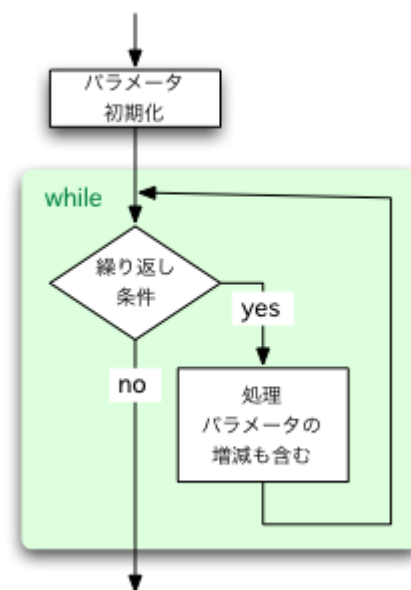
while 文には、パラメータの初期化を行う部分が含まれていない。パラメータの初期化は while に先だって行う必要がある。

また、パラメータの増減を書く位置が決まっていない。それは処理中のどこかに書かなければならない。

パラメータの初期化

```
while ( 繰り返し条件 ) {  
    処理 (パラメータの増減も含む)  
}  
i = 0;
```

```
while ( i < 10 ) {  
    printf("%d ", i);  
    i++;  
}
```



while 文では、処理の先だって繰り返し条件が判断されることに注意しよう。始めから繰り返し条件が成立していないときには、処理は一度も実行されない。

## continue 文による処理のスキップ



while 文中の処理を実行中に continue 文に行き当たったときには、その回のその後の処理がスキップされる。そして繰り返し条件の判断がなされてそれが真であれば次の回の処理に入る。

## break 文による while 文の中断

while 文中の処理を実行中に break 文に行き当たったときには、直ちに while 文の処理を中断して、while 文を終了する。

## do-while 文

do-while 文は、処理を行った後で繰り返し条件がまだ満たされていたら 同じ処理を繰り返したいときに用いる。

do-while 文には、パラメータの初期化を行う部分が含まれていない。パラメータの初期化は do-while に先だって行う必要がある。

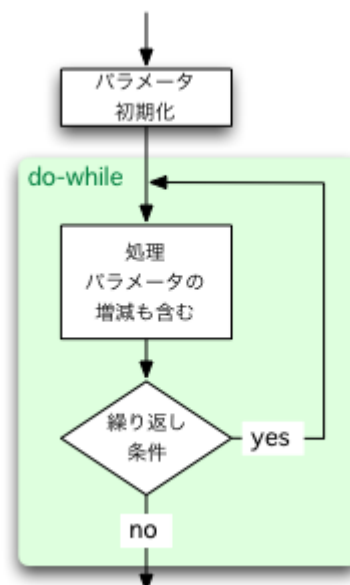
また、パラメータの増減を書く位置が決まっていない。それは処理中のどこかに書かなければならない。

パラメータの初期化

```
do {  
    処理 (パラメータの増減も含む)  
} while ( 繰り返し条件 );
```

while ( ... ) の後のセミicolon ; を忘れやすいので注意。

```
i = 0;  
  
do {  
    printf("%d ", i);  
    i++;
```



```
} while ( i < 10 );
```

do-while 文では、判断に先立って処理が行われることに注意しよう。どのような場合でも、処理は少なくとも1回は実行される。

## continue 文による処理のスキップ

do-while 文中の処理を実行中に continue 文に行き当たったときには、その回のその後の処理がスキップされる。そして繰り返し条件の判断がなされてそれが真であれば次の回の処理に入る。

## break 文による do-while 文の中断

do-while 文中の処理を実行中に break 文に行き当たったときには、直ちに do-while 文の処理を中断して、do-while 文を終了する。

## 無限ループ

プログラムには無限ループはあってはならないが、プログラムのコーディングにおいては、故意に無限ループの形にすることがある。

その形だけの無限ループは、実際には break か return か exit() によって中断される。break はループを終了させるだけだが、return なら同時に関数も終了し、exit() ならプログラムが終了する。

```
for (;;) {                                // 無限 for ループ
    ....
    if ( ... ) break;
    ....
}
while (1) {                                // 無限 while ループ
```

```
....  
if ( ... ) break;  
....  
}
```

## よくある間違い

制御文の書き方において間違いやすいことを注意しておく.

### if

```
if ( a == b ) {      正  
    c = d;  
    e = f;  
}  
if ( a == b )      誤  
    c = d;  
    e = f;
```

誤りの方の if 文は, 段下げの具合から見ると, `a == b` のときに `c = d; e = f;` という二つの代入をしたいようだが, 実際には if 文の処理は `c = d;` という一つの代入だけになる. この場合は `{ }` で括らなければならない.

```
if ( a == b )      正  
    c = d;  
if ( a == b );     誤  
    c = d;
```

誤りの方の if 文は, `if ( a == b )` の後に `;` がついているので, そこで if 文が終わっている. すなわち, `c = d;` という代入文は if 文の処理には含まれない. この場合は `;` は付けてはならない.

```

if ( a == b ) {
    c = d;
}
else {
    e = f;
}
if ( a == b ) {
    c = d;
};
else {
    e = f;
}

```

正

誤

誤りの方の if 文は、最初の {} の後に ; がついているので、そこで if 文が終わっている。したがって、else に対応する if 文がないので、コンパイルエラーとなる。else の直前の } の後ろには ; は付けてはならない。

```

if ( a == b )
    c = d;
else
    e = f;
if ( a == b )
    c = d
else
    e = f;

```

正

誤

誤りの方の if 文は、c = d の後ろに ; がついていない。ここには ; が必要である。

## for, while

```

for ( i = 0 ; i < 10 ; i++ ) {
    a = b;
    c = d;
}

```

正

```
for ( i = 0 ; i < 10 ; i++ )      誤
    a = b;
    c = d;
```

誤りの方の if 文では, for 文の処理は a = b; だけになってしまう. 処理が複数行あるときには, {} で括る必要がある.

```
for ( i = 0 ; i < 10 ; i++ )      正
    a = b;
for ( i = 0 ; i < 10 ; i++ ) ;    誤
    a = b;
```

誤りの方の if 文では, 1 行目の最後に ; が付いているので, そこで for 文が終わっている. これでは, a = b; はこの for 文の処理に含まれない. ここには ; を付けてはならない.

上の for 文についての注意事項は, すべて while 文についても当てはまる.

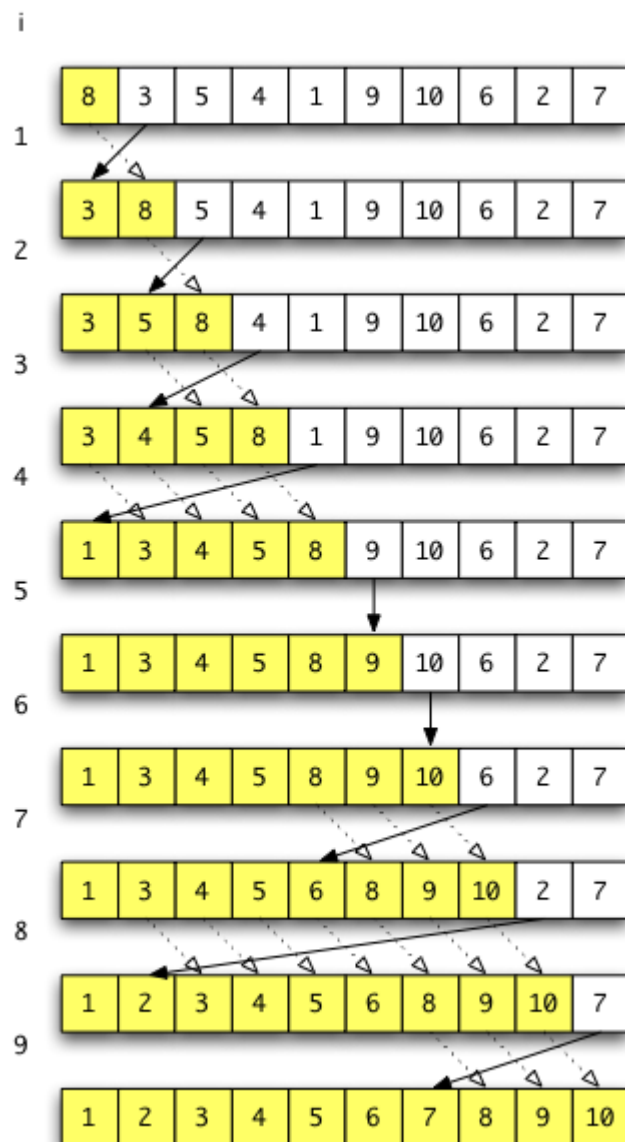
## レポート課題

挿入法によるソートを行う関数 void ins\_sort(int a[], int n) を作れ.

```
void int_sort(int a[], int n)
```

要素数 n の int 型配列 a を挿入法によりソートする.

挿入法は, 右図のように, ソートできている部分(黄色い部分)を一つずつ増やすことを続けていく方法であるが, そのとき, 新しい要素を入れる場所を



探し出して、そこに挿入することで進んでいく。さらには、挿入する場所を探し出すと同時に、そこから右にある要素を一つずつ右にずらして、挿入する場所を空ける必要がある。

挿入法の手続きの最中において、ソートできている部分(黄色い部分)の個数を  $i$  としよう。そして、 $i$  を  $n$  まで増やしていけばよい。

そこで、次のような for ループから始める。(  $i = 0$  ではなくて  $i = 1$  から始まっている理由は何故であるか、考えよう。 )

先頭の  $i$  個 ( $a[0], a[1], \dots, a[i-1]$ ) がソート済みの場合、次に注目するのは、 $a[i]$  の要素である。ループの中の処理は、その  $a[i]$  をソート済みの中の適当な場所に挿入することである。

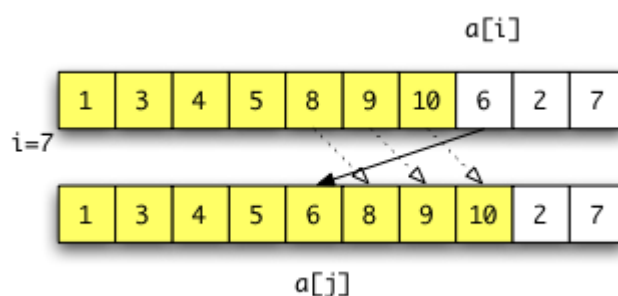
```
for ( i = 1; i < n; i++ ) {  
  
    a[i] を適当な場所に挿入する  
  
}
```

さて次は、「 $a[i]$  を適当な場所に挿入する」という部分を作る。

挿入する場所  $j$  を探していくが、その可能性は  $i, i-1, i-2, \dots, 0$  である。 $j = i$  の場合は、動かさなくてもいいというラッキーな場合だ。右図の例では  $i$  が 5, 6 のときに(9, 10 を挿入するときに)そうになっている。

挿入する場所  $j$  を探すには、ソート済みの部分を左から見ていき、最初に  $a[j-1] \leq a[i]$  となる部分を探せばよい。しかし、注意することは、そのような部分が無いかも知れないということである。その場合、挿入する場所は先頭、すなわち 0 番目である。

挿入する場所を探すと同時に、挿入の場所よりも右にあるものを、右にひとつずつずらさなければならない。たとえば、右図の場合、10, 9, 8 の値をこの順でひとつずつ右にずらす必要がある。



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

a[i] の値を何かに待避させる

```
for ( j = i; j > 0; j-- ) {  
    挿入する場所かどうか  
    そうであれば break  
    そうでなければ a[j-1] の値を a[j] に移す  
}  
  
    退避させておいた a[i] の値を, a[j] に入れる  
  
}
```

for ( j = i; j > 0; j-- ) のループは, break で終わっても通常通りに終わっても, どちらにせよ, 終了時の j の値が, a[i] の値を挿入する場所となる.

以上の考え方で作ればよい. (先ほどは, 動かさなくてもいいラッキーな場合などと言ったが, 結局その場合も動かしている.)

関数ができたら, 次のような main で動かして確かめること.

```
int main()  
{  
    int data[10] = {8, 3, 5, 4, 1, 9, 10, 6, 2, 7};  
    int i;  
  
    ins_sort(data, 10);  
  
    for ( i = 0; i < 10; i++ ) {  
        printf("%3d", data[i]);  
    }  
}
```

1 2 3 4 5 6 7 8 9 10