第1章

基本的なアルゴリズム

Algorithms and Data Structures in C

1-1　アルゴリズムとは

三値の最大値

簡単なプログラムを例に、**アルゴリズム**（*algorithm*）について考えましょう。三つの整数の最大値を求めるプログラムを **List 1-1** に示します。

List 1-1

```
/*
    三つの整数値の最大値を求める
*/

#include  <stdio.h>

int main(void)
{
    int  a, b, c;
    int  max;                  /* 最大値 */

    printf("整数ａの値：");    scanf("%d", &a);
    printf("整数ｂの値：");    scanf("%d", &b);
    printf("整数ｃの値：");    scanf("%d", &c);

    max = a;
    if (b > max) max = b;
    if (c > max) max = c;

    printf("最大値は%dです。\n", max);

    return (0);
}
```

実行例

```
整数ａの値：1↵
整数ｂの値：3↵
整数ｃの値：2↵
最大値は3です。
```

このプログラムは、まず変数 *a*，*b*，*c* に整数値を読み込み、それらの最大値を求めて変数 *max* に格納し、その値を表示します。網掛け部が最大値を求める部分であり、次のように動作します。

(1) *max* に *a* の値を代入する。
(2) *b* の値がそれよりも大きければ、*max* に *b* の値を代入する。
(3) *c* の値がそれよりも大きければ、*max* に *c* の値を代入する。

この処理の流れを表した**流れ図＝フローチャート**（*flowchart*）を **Fig.1-1** に示しています。プログラムの実行は、黒い線に沿って、上から下へと順に流れていきます。

ただし、ひし形を通過する際は、その中に書かれている条件を評価した結果に応じて、Yes あるいは No の二つの分岐のいずれかをたどります。したがって、このような分岐は、**双岐選択**と呼ばれ、C 言語では **if** 文に対応します。

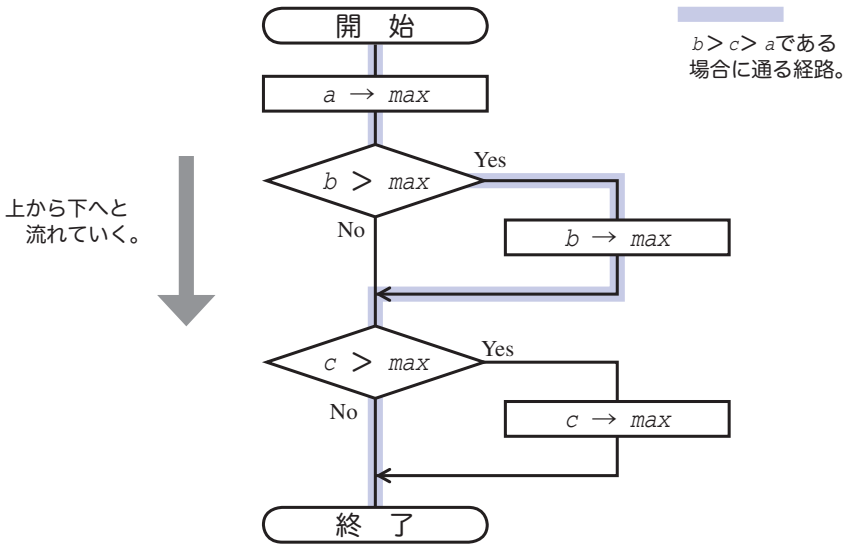


Fig.1-1　三値の最大値を求めるアルゴリズム

このフローチャートでは、*b* > *max* や *c* > *max* が成立すれば Yes と書かれている右側に進み、そうでなければ、そのまま下に進みます。

■　フローチャートの記号については、*p.22* にまとめています。

また、矢印記号 → は、代入を表します。たとえば“*a* → *max*”は、変数 *a* の値を変数 *max* に代入せよ、という指示です。

さて、左ページの＜実行例＞に示すのは、変数 *a*，*b*，*c* の値として 1，3，2 を入力した場合の実行の様子です。このとき、プログラムの流れは、フローチャート上の薄青い線で示した経路をたどります。

それでは、これ以外の値を想定して、フローチャートをなぞってみましょう。たとえば、変数 *a*，*b*，*c* の値が 1，2，3 でも 3，2，1 でも、正しく最大値を求めることができるでしょうか？　また、三つの値が 5，5，5 とすべて等しかったり、5，3，5 と二つが等しい場合でも正しく最大値を求められるでしょうか？　いろいろな値で確認してみましょう。

＊

フローチャート内の薄青い線は、*a*，*b*，*c* の値が 1，3，2 である場合を示したものと説明しました。しかし、これらの値が 6，10，7 でも、-10，100，10 でも、とにかく *b* ＞ *c* ＞ *a* であれば、同じ経路をたどります。

すなわち、三値の具体的な値ではなくて、その大小関係のあらゆる場合に対して、最大値を求められるかどうかを確認すればよさそうです。

三値の大小関係をすべて列挙した図を **Fig.1-2** に示します。丸枠の中に書かれている条件が成立すれば上側、そうでなければ下側の線をたどっていきます。右端の薄青い四角枠の中に書かれているのが三値の大小関係です。

■　このような図は**決定木**と呼ばれます。

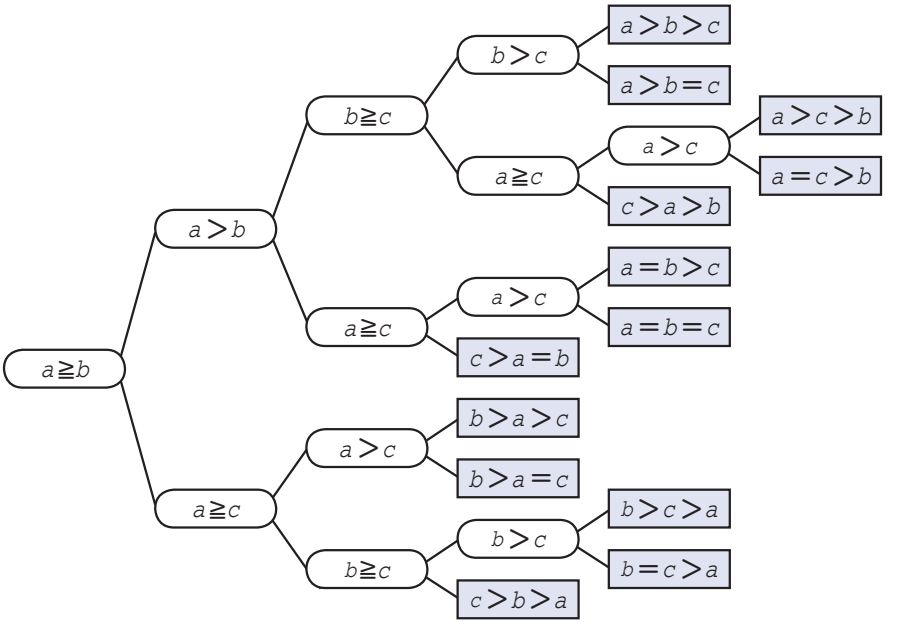


Fig.1-2　三値の大小関係の組合せ

この図から、三値の大小関係の組合せは 13 種類あることが分かります。

そこで、これらのすべてに対して、最大値を求めて表示してみましょう。そのプログラムを **List 1-2** に示します。

なお、大小関係を満たしていれば具体的な値は任意ですから、確認しやすいように、どの組合せでも最大値が 3 となるようにしています。

プログラムを実行すると、すべての組合せに対して 3 と表示され、正しく最大値を求めていることが確認できます。

なお、最大値を求める部分は繰り返し利用されますので、**関数**（*function*）として実現しています。

関数 *max3* は、受け取った三つの **int** 型の仮引数 *a*，*b*，*c* の最大値を求めて、それを **int** 型の値として返します。**main** 関数からは、関数 *max3* を 13 回呼び出しています。

関数はプログラムを構成するための便利な“部品”であることが、この例からも分かりますね。

List 1-2

```
/*
    三つの整数値の最大値を求める（すべての大小関係に対して確認）
*/

#include  <stdio.h>

/*--- a, b, cの最大値を求める ---*/
int max3(int a, int b, int c)
{
    int  max = a;                /* 最大値 */

    if (b > max) max = b;
    if (c > max) max = c;

    return (max);
}

int main(void)
{
    printf("max3(%d,%d,%d) = %d\n", 3, 2, 1, max3(3, 2, 1));    /* a>b>c */
    printf("max3(%d,%d,%d) = %d\n", 3, 2, 2, max3(3, 2, 2));    /* a>b=c */
    printf("max3(%d,%d,%d) = %d\n", 3, 1, 2, max3(3, 1, 2));    /* a>c>b */
    printf("max3(%d,%d,%d) = %d\n", 3, 2, 3, max3(3, 2, 3));    /* a=c>b */
    printf("max3(%d,%d,%d) = %d\n", 2, 1, 3, max3(2, 1, 3));    /* c>a>b */
    printf("max3(%d,%d,%d) = %d\n", 3, 3, 2, max3(3, 3, 2));    /* a=b>c */
    printf("max3(%d,%d,%d) = %d\n", 3, 3, 3, max3(3, 3, 3));    /* a=b=c */
    printf("max3(%d,%d,%d) = %d\n", 2, 2, 3, max3(2, 2, 3));    /* c>a=b */
    printf("max3(%d,%d,%d) = %d\n", 2, 3, 1, max3(2, 3, 1));    /* b>a>c */
    printf("max3(%d,%d,%d) = %d\n", 2, 3, 2, max3(2, 3, 2));    /* b>a=c */
    printf("max3(%d,%d,%d) = %d\n", 1, 3, 2, max3(1, 3, 2));    /* b>c>a */
    printf("max3(%d,%d,%d) = %d\n", 2, 3, 3, max3(2, 3, 3));    /* b=c>a */
    printf("max3(%d,%d,%d) = %d\n", 1, 2, 3, max3(1, 2, 3));    /* c>b>a */

    return (0);
}
```

実行結果

```
max3(3,2,1) = 3
max3(3,2,2) = 3
max3(3,1,2) = 3
max3(3,2,3) = 3
max3(2,1,3) = 3
max3(3,3,2) = 3
max3(3,3,3) = 3
max3(2,2,3) = 3
max3(2,3,1) = 3
max3(2,3,2) = 3
max3(1,3,2) = 3
max3(2,3,3) = 3
max3(1,2,3) = 3
```

JIS X0001 では、《アルゴリズム》は次のように定義されています。

> 問題を解くためのものであって、明確に定義され、順序付けられた有限個の規則からなる集合。

もちろん、いくら曖昧さのないように記述されていても、変数の値によって、解けたり解けなかったりするのでは、正しいアルゴリズムとはいえません。

ここでは、三値の最大値を求めるアルゴリズムが正しいことを、論理的に確認するとともに、プログラムの実行結果からも確認したわけです。

■ 演習 1-1

三値の最小値を求める以下の関数を作成せよ。

```
int min3(int a, int b, int c);
```

■ 演習 1-2

三値の中央値を求める以下の関数を作成せよ。

```
int med3(int a, int b, int c);
```

1-2　繰返し

1からnまでの整数の和を求める

次に、1 から *n* までの整数の和を求めるアルゴリズムを考えましょう。たとえば、*n* が 2 であれば 1 + 2 を、*n* が 3 であれば 1 + 2 + 3 を求めます。すなわち、より一般的には、次の値を求めます。

1 + 2 + … + *n*

この手続きは、合計した値を格納するための変数を *sum* とすると、**Fig.1-3** に示すフローチャートとなります。

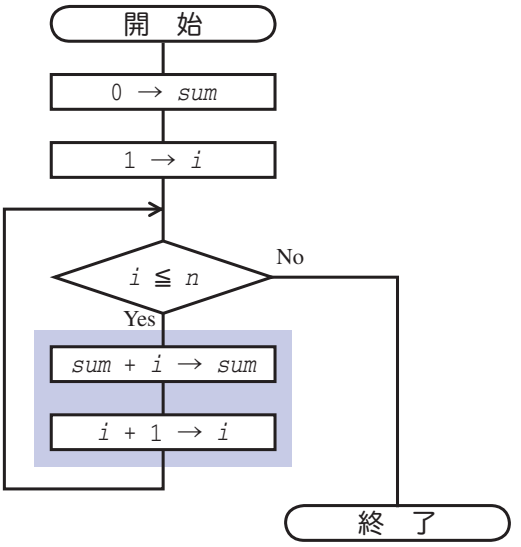


Fig.1-3　1からnまでの和を求める

■ while文による繰返し

このフローチャートに基づいたプログラムの例を **List 1-3** に示します。C 言語の **while** 文の形式は、

while（*制御式*）
　ループ本体

であり、*制御式*の評価によって得られる値が 0 でない限り、*ループ本体*を繰り返して実行します。すなわち、**前判定繰返し**の実現に適しています。

■　最初に*制御式*を評価した際に得られた値が 0 であれば、*ループ本体*は一度も実行されません。この点で、後判定繰返しを実現する **do** 文（*p.24*）と異なります。

List 1-3

```
/*
    1, 2, …, nの和を求める（while文）
*/

#include  <stdio.h>

int main(void)
{
    int  i, n;
    int  sum;                  /* 和 */

    puts("1からnまでの和を求めます。");

    printf("nの値：");
    scanf("%d", &n);

    sum = 0;
    i = 1;

    while (i <= n) {           /* iがn以下であれば繰り返す */
        sum += i;              /* sumにiを加える */
        i++;                   /* iの値をインクリメント */
    }
    printf("1から%dまでの和は%dです。\n", n, sum);

    return (0);
}
```

実行例

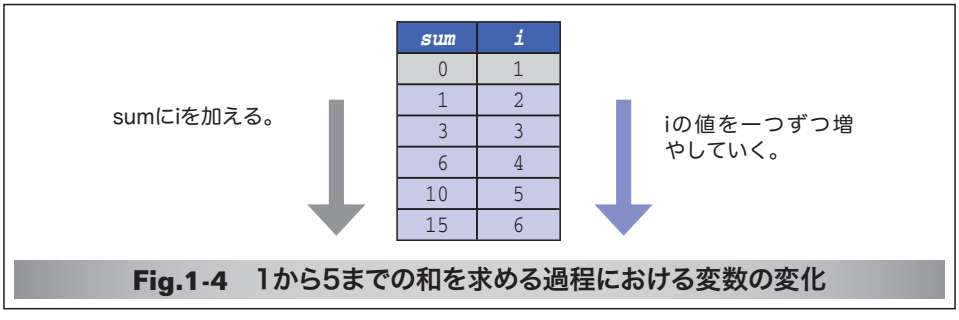
```
1からnまでの和を求めます。
nの値：10
1から10までの和は55です。
```

フローチャート内で、薄青い網のかかった部分が、プログラムの網掛け部に対応します。最初に 1 が代入されている *i* の値を、一つずつ増やしていき、その値が *n* 以下である限り、この部分が実行されます。すなわち、繰り返される回数は、都合 *n* 回です。

変数 *i* と *sum* の値の変化を **Fig.1-4** に示します。*i* の値を 1 から *n* まで増やしながら、その値を *sum* に加えることによって合計を求めていくわけですが、その過程における値の変化が分かりますね。

■　複合代入演算子 **+=** は＜右辺の値を左辺に加える＞ことを意味し、増分演算子 **++** は＜値を一つ増やす＞ことを意味します。

なお、*i* の値が *n* を超えたときに **while** 文の繰返しが終了しますので、最終的な *i* の値は、*n* ではなく *n* + 1 となることに注意しましょう。



sum	i
0	1
1	2
3	3
6	4
10	5
15	6

Fig.1-4　1から5までの和を求める過程における変数の変化

■ for文による繰返し

ある特定の変数の値で制御する繰返しは、**while** 文ではなく **for** 文を用いて実現すると、プログラムがよりスマートになります。

そのように書きかえたプログラムを **List 1-4** に示します。

List 1-4

```
/*
    1, 2, …, nの和を求める（for文を利用）
*/

#include  <stdio.h>

int main(void)
{
    int  i, n;
    int  sum;                  /* 和 */

    puts("1からnまでの和を求めます。");

    printf("nの値：");
    scanf("%d", &n);

    sum = 0;

    for (i = 1; i <= n; i++) {          /* i = 1, 2, …, n */
        sum += i;                       /* sumにiを加える */
    }

    printf("1から%dまでの和は%dです。\n", n, sum);

    return (0);
}
```

実行例

```
1からnまでの和を求めます。
nの値：10↵
1から10までの和は55です。
```

合計を求める網掛け部のフローチャートを **Fig.1-5** に示します。

台形と長方形を組み合わせた形の六角形は、**ループ端**（*loop limit*）と呼ばれ、繰返しを指示する記号です。なお、その中に書かれた、

```
i : 1, 1, n
```

という式は、“変数名：初期値，増分，終値”です。

したがって、変数 *i* の値は 1 から *n* まで、1 ずつ増えることになります。

Column 1-1　while 文と for 文

C 言語の **while** 文と **for** 文は互いに可換であり、どちらを利用しても、まったく同じことを実現できます。すなわち、以下に示す二つは、同じように働きます。

```
for (式1; 式2; 式3)
    文
```

```
式1;
while (式2) {
    文
    式3;
}
```

なお、**for** 文の式 $_1$、式 $_2$、式 $_3$ は、いずれも省略可能です。式 $_2$ を省略した場合は、整数値 1 が指定されたものとみなされます。

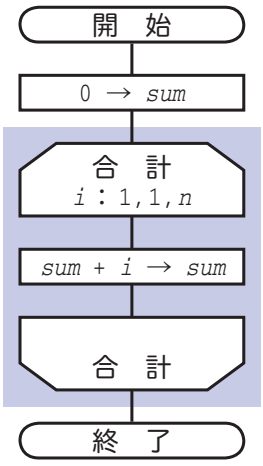


Fig.1-5　1からnまでの和を求める

■　フローチャートでは、*i* の値は最終的に *n* となります。一方、プログラムでは、*i* の値が *n* を超えたときに **for** 文による繰返しが終了しますから、その最終的な値は *n* + 1 となります。したがって、フローチャートとプログラムは完全に対応しているわけではないことに注意してください。

なお、ループ端の中に書く、繰返しを制御するための式には、いくつかの記述法があります。

たとえば、**Fig.1-6** に示すのは、変数の値をコンマで区切って並べ、省略する部分を … と記す方法です。

いずれにせよ、繰返しの開始と終了を対応させるために、二つのループ端には同じ名前をつけます。

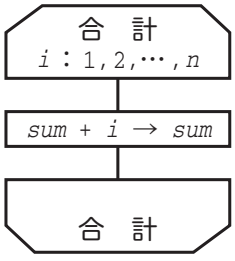


Fig.1-6　繰返し記号

■ 演習 1-3

List 1-4 のプログラムを、たとえば *n* が 7 であれば『1 から 7 までの和は 28 です。』と表示するのではなく、『1 + 2 + 3 + 4 + 5 + 6 + 7 = 28』と表示するように変更せよ。

■ 演習 1-4

整数 *a*, *b* を含め、その間の全整数の和を求めて返す以下の関数を作成せよ。

```
int sumof(int a, int b);
```

なお、*a* と *b* の大小関係に関係なく和を求めること。たとえば *a* が 3 で *b* が 5 であれば 12 を、*a* が 6 で *b* が 4 であれば 15 を返すこと。

Column 1-2　フローチャートの記号

問題の定義、分析、解法の図的表現である**フローチャート**（*flowchart*）と、その記号については、JIS X012『情報処理用流れ図・プログラム網図・システム資源図記号』で定義されています。

ここでは、代表的な用語と記号を簡単に紹介します。

プログラム流れ図（*program flowchart*）

プログラム流れ図は、以下のものから構成されます。

■実際に行う演算を示す記号。
■制御の流れを示す線記号。
■プログラム流れ図を理解し、かつ作成するのに便宜を与える特殊記号。

データ（*data*）

媒体を指定しないデータを表します。

処理（*process*）

任意の種類の処理機能を表します。たとえば、情報の値、形、位置を変えるように定義された演算もしくは演算群の実行、または、それに続くいくつかの流れの方向の一つを決定する演算もしくは演算群の実行を表します。

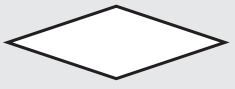
定義済み処理（*predefined process*）

サブルーチンやモジュールなど、別の場所で定義された一つ以上の演算または命令群からなる処理を表します。

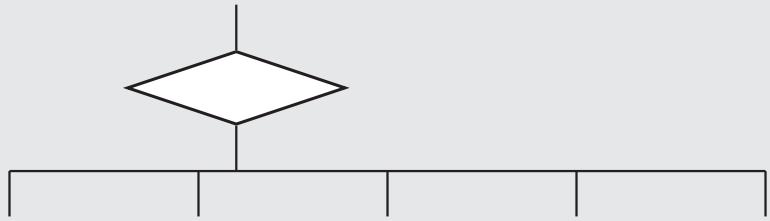
判断（*decision*）

一つの入り口といくつかの択一的な出口をもち、記号中に定義された条件の評価にしたがって、唯一の出口を選ぶ判断機能またはスイッチ形の機能を表します。

想定される評価結果は、経路を表す線の近くに書きます。



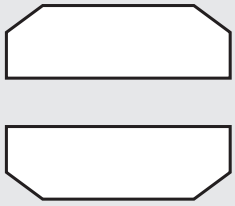
なお、C言語の **switch** 文のように、多重に分岐する場合は、たとえば以下のように表記します。



ループ端（*loop limit*）

二つの部分から構成され、ループの始まりと終わりを表します。記号の二つの部分には、同じ名前を与えます。

テスト命令の位置に応じて、ループの始端または終端の記号中に、初期化、増分、終了条件を表記します。



線（*line*）

制御の流れを表します。

流れの向きを明示する必要があるときは、矢先を付けなければなりません。なお、そうでない場合でも、見やすくするために矢先を付けても構いません。

端子（*terminator*）

外部環境への出口、または外部環境からの入り口を表します。たとえば、プログラムの流れの開始もしくは終了を表します。

この他に、並列処理、破線などの記号があります。

正の値の読込み

和を求めるプログラムを実行して、*n* に対して -5 といった負の値を入力してみましょう。そうすると、次のように表示されます。

```
1 から -5 までの合計は 0 です。
```

これは、数学的にも感覚的にもおかしいですね。

そもそも、このプログラムでは、正の値のみを *n* に読み込むべきです。そのように改良したプログラムを **List 1-5** に示します。

List 1-5

```
/*
    1, 2, …, nの和を求める（nに正の整数を読み込む）
*/

#include  <stdio.h>

int main(void)
{
    int  i, n;
    int  sum;                  /* 和 */

    puts("1からnまでの和を求めます。");

    do {
        printf("nの値：");
        scanf("%d", &n);
    } while (n <= 0);

    sum = 0;

    for (i = 1; i <= n; i++) {
        sum += i;                          /* sumにiを加える */
    }

    printf("1から%dまでの和は%dです。\n", n, sum);

    return (0);
}
```

実行例

```
1からnまでの和を求めます。
nの値：-5
nの値：8
1から8までの和は36です。
```

実行例に示すように、*n* の値に 0 以下の値を入力すると、『n の値：』と表示されて、値を再入力するように促します。

値を読み込むのが網掛け部の **do** 文です。この部分のフローチャートは **Fig.1-7** であり、**後判定繰返し**を行っているわけです。

変数 *n* に読み込まれた値が 0 以下である限り繰返しが行われますので、**do** 文が終了したときに *n* の値は必ず正となります。

フローチャートの図 **(a)** と図 **(b)** は、本質的には同じです。ただし、図 **(b)** のように、繰返しの条件を下側のループ端に書く方法は、前判定繰返しと区別しにくく紛らわしくなりますので、図 **(a)** の書き方が好まれるようです。

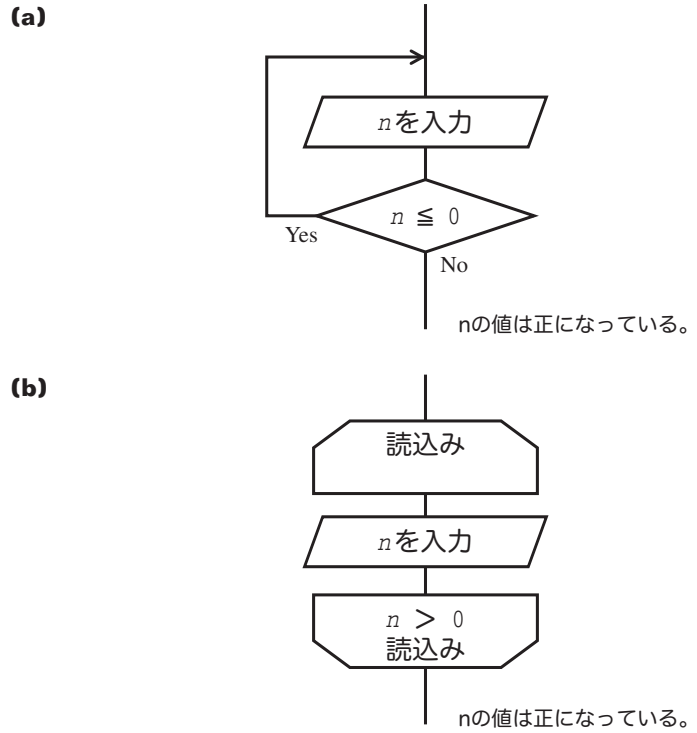


Fig.1-7　正の値を読み込む

■ 演習 1-5

正の整数値を読み込んで、その値の桁数を表示するプログラムを作成せよ。たとえば、135 を読み込んだら『その数は 3 桁です。』と表示し、1314 を読み込んだら『その数は 4 桁です。』と表示すること。

■ 演習 1-6

右に示すように、二つの変数 *a*，*b* に整数値を読み込んで *b* - *a* の値を表示するプログラムを作成せよ。

なお、変数 *b* に読み込んだ値が *a* 以下であれば再入力させること。

```
aの値：5↵
bの値：4↵
aより大きな値を入力せよ！
bの値：7↵
b - aは2です。
```

Column 1-3　関数呼出しと実引数・仮引数

関数間で情報をやり取りするために受け渡すのが引数です。**List 1-2** に示した関数 *max3* には三つの引数があります。

関数を呼び出す側が渡すのが**実引数**（*argument*）、呼び出される側が受け取るのが**仮引数**（*parameter*）であり、仮引数は実引数のコピーです。したがって、たとえ関数内で仮引数の値を変更しても、実引数に影響を与えることはありません。

1

多重ループ

ここまでは単純な繰返しでしたが、繰返しの中でも繰返しを行えます。そのような繰返しは、その深さに応じて、二重ループ、三重ループ、…と呼ばれます。なお、これらをまとめて一般に**多重ループ**と呼びます。

ここでは二重ループの例として、記号文字 * を利用して左下側が直角の三角形を表示するプログラムを **List 1-6** に示します。

List 1-6

```
/*
    左下側が直角の三角形を表示
*/

#include  <stdio.h>

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

    printf("何段の三角形ですか：");
    scanf("%d", &n);

    for (i = 1; i <= n; i++) {
        for (j = 1; j <= i; j++)
            putchar('*');
        putchar('\n');
    }

    return (0);
}
```

実行例

```
何段の三角形ですか：5
*
**
***
****
*****
```

プログラム中の網掛け部が、直角三角形の表示を行う部分であり、そのフローチャートは **Fig.1-8** となります。

それでは、実行例に示すように n の値が 5 のときに、どのような処理が行われるかを考えましょう。

外側の **for** 文は、i の値を 1 から始めて n になるまで一つずつ増やします。その際、内側の **for** 文は、j の値を 1 から始めて i まで一つずつ増やしながら表示を行います。したがって、次のように動作することになります。

```
i が 1 のとき：j を 1 から 1 まで増やして繰返し   →  *
i が 2 のとき：j を 1 から 2 まで増やして繰返し   →  **
i が 3 のとき：j を 1 から 3 まで増やして繰返し   →  ***
i が 4 のとき：j を 1 から 4 まで増やして繰返し   →  ****
i が 5 のとき：j を 1 から 5 まで増やして繰返し   →  *****
```

すなわち、三角形を第 1 行から第 n 行と考えると、第 n 行目に n 個の * 記号を表示するのです。

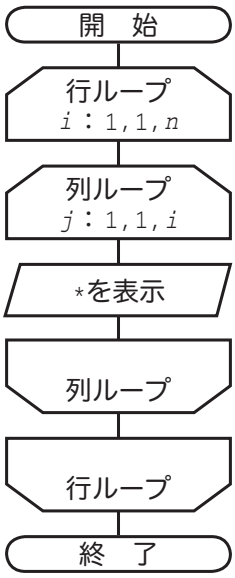


Fig.1-8　直角三角形を表示

■ 演習 1-7

直角三角形を表示する部分を独立させて以下の形式の関数として実現せよ。

```
void trilb(int n);
```

さらに、直角が左上側、右上側、右下側の三角形を表示する関数を作成せよ。

```
void trilu(int n);
void triru(int n);
void trirb(int n);
```

■ 演習 1-8

右図のように、*n* 段のピラミッドを表示する関数を作成せよ。

```
void spira(int n);
```

第 *n* 行目には、(*n* - 1) * 2 + 1 個の * 記号を表示すること。

```
   *
  ***
 *****
*******
```

■ 演習 1-9

右図のように、*n* 段の数字ピラミッドを表示する関数を作成せよ。

```
void npira(int n);
```

第 *n* 行目に表示するのは、*n* % 10 とすること。

```
   1
  222
 33333
4444444
```

■ 演習 1-10

九九の表を表示するプログラムを作成せよ。