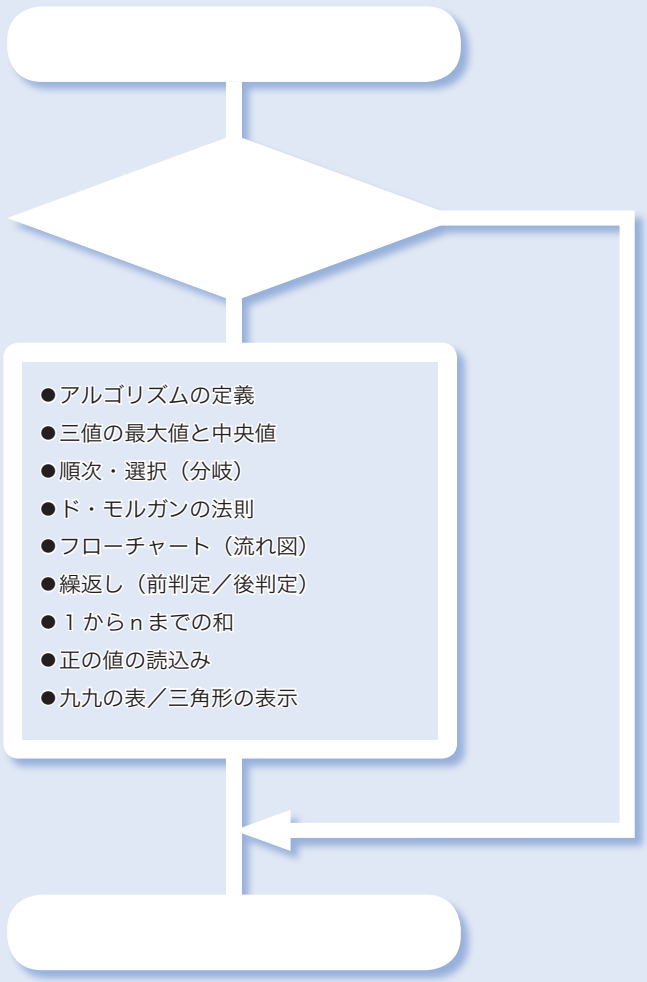
第１章

基本的なアルゴリズム



1–1 アルゴリズムとは

本節では、短く単純なプログラムを題材として、《アルゴリズム》とは何かを理解するとともに、その定義などを学習します。

三値の最大値

まず最初に、**アルゴリズム**（*algorithm*）とは何かを、短く単純なプログラムを例にとって考えていくことにします。その題材として取り上げるのは、三つの値の《最大値》を求めるプログラムです。

そのプログラムが **List 1-1** です。変数 a，b，c には、キーボードから読み込んだ値が入れられます。そして、それら三値の最大値を変数 *max* に求めて表示します。

まずは、プログラムを実行して、動作を確認してみましょう。

List 1-1 chap01/max3.c

```
/* 三つの整数値を読み込んで最大値を求める */

#include <stdio.h>

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

    printf("三つの整数の最大値を求めます。\n");
    printf("aの値：");    scanf("%d", &a);
    printf("bの値：");    scanf("%d", &b);
    printf("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;
}
```

a, b, cの最大値を求めてmaxに代入

実行例

```
三つの整数の最大値を求めます。
aの値：1↵
bの値：3↵
cの値：2↵
最大値は3です。
```

変数 a，b，c の最大値を *max* として求めるのが、プログラムの網かけ部です。最大値を求める手順は、以下のようになっています。

① *max* に a の値を入れる。
② b の値が *max* よりも大きければ、*max* に b の値を入れる。
③ c の値が *max* よりも大きければ、*max* に c の値を入れる。

三つの文が並んでおり、これらの文が順番に実行されます。すなわち**順次**（*concatination*）構造です。なお、①は単純な代入ですが、②と③は **if** 文です。**if** 文は ( ) の中の式を評価した値に応じてプログラム実行の流れを制御する**選択**（*selection*）構造です。

Column 1-1　演算子とオペランド／式と評価

■演算子とオペランド

プログラミング言語の + や > などの演算を行う記号は**演算子**（*operator*）と呼ばれ、演算の対象となる式は**オペランド**（*operand*）と呼ばれます。たとえば、a と b の値の大小関係の判定を行うための式 a > b において、演算子は > であって、オペランドは a と b の二つです。

演算子 > のように、二つのオペランドをもつ演算子を**２項演算子**（*binary operator*）と呼びます。Ｃ言語には、２項演算子のほかに、オペランドが一つの**単項演算子**（*unary operator*）と、オペランドが三つの**３項演算子**（*ternary operator*）があります。

■式と評価

プログラムの実行時に、式は評価されます。

▪ 式

厳密な定義ではないのですが、**式**（*expression*）とは、以下のものの総称です。

▫ 変数　　▫ 定数　　▫ 変数や定数を演算子で結合したもの

ここで、式 $x = n + 135$ を考えましょう（変数 x と n は `int` 型であるとします）。この式において、x，n，135，$n + 135$，$x = n + 135$ のいずれもが式です。

なお、○○演算子とオペランドとが結合された式は、○○式と呼ばれます。たとえば、代入演算子によって x と $n + 135$ が結び付けられた式 $x = n + 135$ は、**代入式**（*assignment expression*）です。

演算の対象となる式はオペランドと呼ばれるのでした。たとえば、$n + 135$ では、演算子 + のオペランドは n と 135 であり、$x = n + 135$ では、演算子 = のオペランドは x と $n + 135$ です。

▪ 式の評価

原則として、すべての式に値があります（特別な型である `void` 型の式だけは、例外的に値がありません）。その値は、プログラム実行時に調べられます。式の値を調べることを**評価**（*evaluation*）といいます。

評価のイメージの具体例を示したのが、**Fig.1C-1** です（この図は、`int` 型変数 n の値が 52 であると仮定しています）。

変数 n の値が 52 ですから、n，135，$n + 135$ の各式を評価した値は 52，135，187 となります。もちろん、三つの値の型はいずれも `int` 型です。

このように、本書では、ディジタル温度計のような図で評価値を示すことにします。左側の小さな文字が《型》で、右側の大きな文字が《値》です。

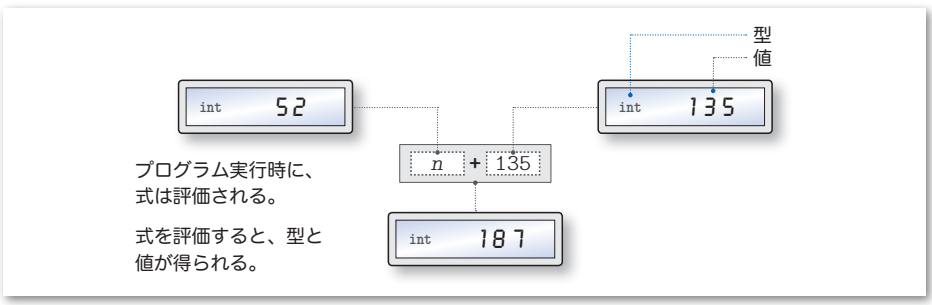


Fig.1C-1　式の評価（int 型＋int 型）

三値の最大値を求める手続きを**流れ図＝フローチャート**（*flowchart*）として図式化すると、プログラムの流れが理解しやすくなります。**Fig.1-1** に示すのが、そのフローチャートです。

▶ フローチャートの記号は p.24 で学習します。

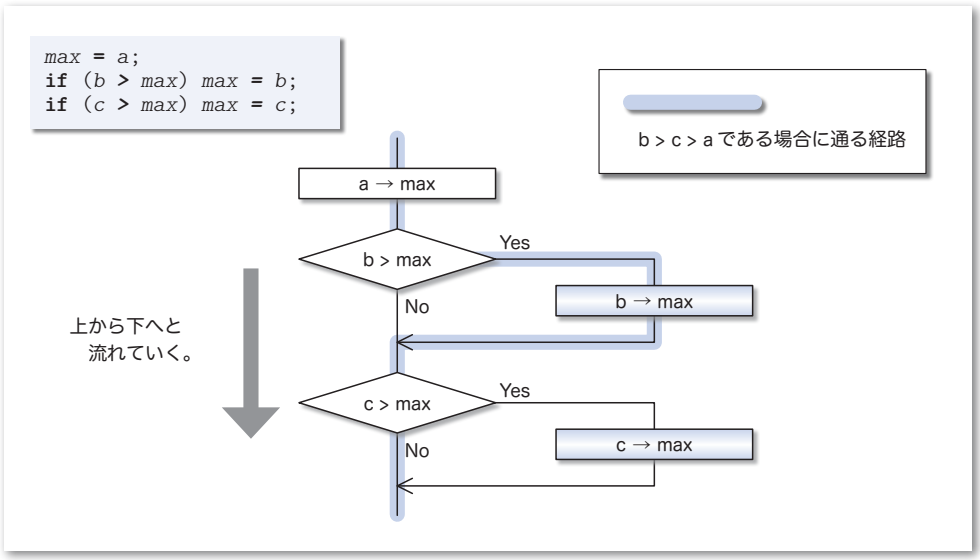


Fig.1-1 三値の最大値を求めるアルゴリズムの流れ図

プログラムの流れは、黒線 ── に沿って上から下へと向かい、その過程で［　　］内の処理が実行されます。

ただし、◇を通過する際は、その中に記された《条件》を評価した結果に応じて、Yes と No のいずれか一方をたどります。したがって、条件 *b* > *max* や *c* > *max* が成立すれば（式 *b* > *max* や式 *c* > *max* を評価した値が 1 であれば）、Yes と書かれた**右側**に進み、そうでなければ No と書かれた**下側**に進みます。

▶ **if** 文や **while** 文などの条件判定のために置かれる ( ) 中の式は、**制御式**と呼ばれます。

二つに分岐するプログラムの流れの一方を通るわけですから、**if** 文によるプログラムの流れの分岐は、**双岐選択**と呼ばれます。

なお、［　　］内の矢印記号 → は、値の代入を表します。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。

▶ この後で学習する **List 1-2**（p.18）の宣言 “**int** *max* = a;” で行われるのは、変数を作る際に値を入れる《**初期化**》で、本プログラムの “*max* = a;” で行われるのは、既に作られている変数に値を入れる《**代入**》です。初期化と代入は異なるものですが、本書の解説では、厳密に区別する必要がない文脈に限り、両者をまとめて “代入” と呼ぶことにします。

p.14 に示した実行例のように、変数 a，b，c に対して 1，3，2 を入力すると、プログラムの流れはフローチャート上の**青い線** の経路をたどります。

それでは、他の値を想定して、フローチャートをなぞってみましょう。

変数 a，b，c の値が、1，2，3 や 3，2，1 であっても、最大値を求められます。また、三つの値が 5，5，5 とすべて等しかったり、1，3，1 と二つが等しくても、正しく最大値を求められます（**Fig.1-2**）。

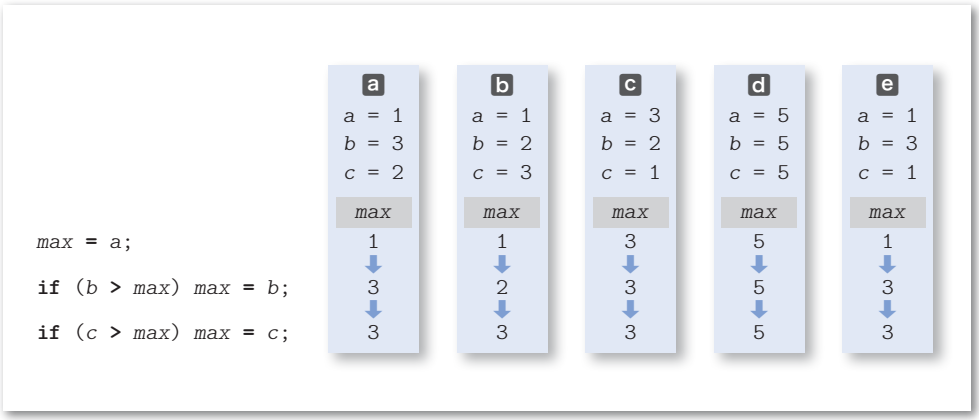


Fig.1-2　三値の最大値を求める過程における変数 max の値の変化

三つの変数 a，b，c の値が、6，10，7 や -10，100，10 であっても、フローチャート内の**青い線**をたどります。すなわち、b ＞ c ＞ a であれば、**同じ経路をたどります**。

Column 1-2　関係演算子と等価演算子

左右のオペランドの大小関係を判定する関係演算子 **<**，**<=**，**>**，**>=** と等値関係を判定する等価演算子 **==**，**!=** は、大小関係や等値関係の判定が成立すれば（真であれば）**int** 型の 1 を、成立しなければ（偽であれば）**int** 型の 0 を生成します。

いくつかの例を **Fig.1C-2** に示しています。たとえば、式 5 **>** 3 を評価して得られる値は **int** 型の 1 で（図a）、5 **==** 3 を評価して得られる値は **int** 型の 0 です（図b）。なお、図cのように、オペランドが **int** 型でない場合でも、関係式や等価式を評価して得られる値の型は **int** 型です。

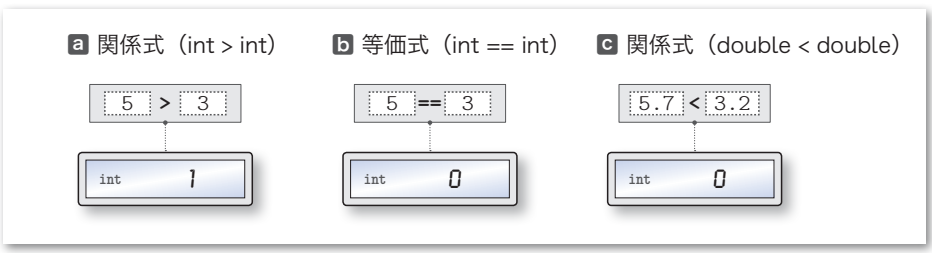


Fig.1C-2　関係式と等価式の評価

三値の**具体的な値**ではなく、すべての**大小関係**に対して、最大値を正しく求められるかどうかを確認することにしましょう。確認を手作業で行うのは大変ですから、プログラムによって行うことにします。それが**List 1-2**に示すプログラムです。

List 1-2 chap01/max3comb.c

```
/* 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] a>b>c */
    printf("max3(%d,%d,%d) = %d\n", 3, 2, 2, max3(3, 2, 2));    /* [B] a>b=c */
    printf("max3(%d,%d,%d) = %d\n", 3, 1, 2, max3(3, 1, 2));    /* [C] a>c>b */
    printf("max3(%d,%d,%d) = %d\n", 3, 2, 3, max3(3, 2, 3));    /* [D] a=c>b */
    printf("max3(%d,%d,%d) = %d\n", 2, 1, 3, max3(2, 1, 3));    /* [E] c>a>b */
    printf("max3(%d,%d,%d) = %d\n", 3, 3, 2, max3(3, 3, 2));    /* [F] a=b>c */
    printf("max3(%d,%d,%d) = %d\n", 3, 3, 3, max3(3, 3, 3));    /* [G] a=b=c */
    printf("max3(%d,%d,%d) = %d\n", 2, 2, 3, max3(2, 2, 3));    /* [H] c>a=b */
    printf("max3(%d,%d,%d) = %d\n", 2, 3, 1, max3(2, 3, 1));    /* [I] b>a>c */
    printf("max3(%d,%d,%d) = %d\n", 2, 3, 2, max3(2, 3, 2));    /* [J] b>a=c */
    printf("max3(%d,%d,%d) = %d\n", 1, 3, 2, max3(1, 3, 2));    /* [K] b>c>a */
    printf("max3(%d,%d,%d) = %d\n", 2, 3, 3, max3(2, 3, 3));    /* [L] b=c>a */
    printf("max3(%d,%d,%d) = %d\n", 1, 2, 3, max3(1, 2, 3));    /* [M] c>b>a */

    return 0;
}
```

求めた最大値を呼出し元に返却

実行結果

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

▶ コメントの[A]，[B]，…，[M]は、**Fig.1C-4**（p.20）のA，B，…，Mに対応します。

最大値を求める部分は、何度も繰り返して利用されるため、本プログラムでは、独立した**関数**（*function*）として実現されています。網かけ部の*max3*は、受け取った三つの**int**型仮引数a，b，cの最大値を求めて、それを**int**型の値として返却する関数です。

関数*max3*は、**main**関数から13回呼び出されています。**main**関数では、関数*max3*に対して三つの値を実引数として渡して呼び出し、関数の返却値を表示します（**Column 1-3**）。

*

計算結果が正しいかどうかを確認しやすくするために、本プログラムでは、すべての呼出しにおいて、最大値が3となるように組み合わせた値を渡しています。

プログラムを実行してみましょう。13種類すべての組合せに対して3と表示され、最大値を正しく求めていることが確認できます。

▶ 大小関係が全部で13種類であることについては、**Column 1-4**（p.20）で学習します。

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

四値の最大値を求める関数 *max4* を作成せよ。

```
int max4(int a, int b, int c, int d);
```

作成した関数をテストするための **main** 関数などを含んだプログラムを作成すること。以降の問題でも、同様である。

演習 1–2

三値の最小値を求める関数 *min3* を作成せよ。

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

演習 1–3

四値の最小値を求める関数 *max4* を作成せよ。

```
int min4(int a, int b, int c, int d);
```

Column 1-3　関数の返却値と関数呼出し式の評価

返却値型が **void** でない関数は、**return** 文によって呼出し元に値を返却します。関数 *max3* の場合、返却値型は **int** 型であり、関数の末尾で変数 *max* の値を返却します。

返却された値は、関数呼出し式の評価によって得られます。たとえば、*max*(3, 2, 1) と呼び出した場合、**Fig.1C-3** に示すように、関数呼出し式 *max*(3, 2, 1) を評価した値が、**int** 型の 3 となります。

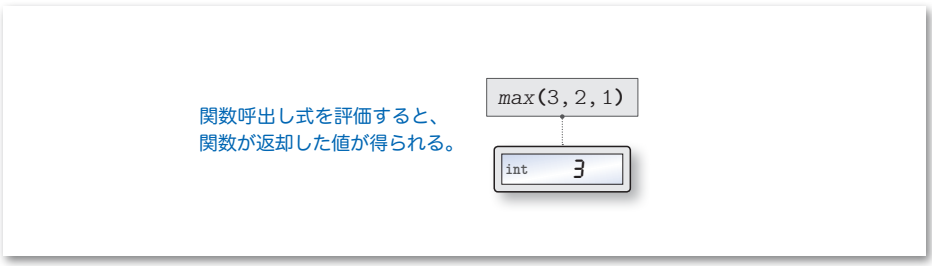


Fig.1C-3　関数呼出し式の評価

Column 1-4 三値の大小関係と中央値

三値の大小関係の組合せ 13 種類は、**Fig.1C-4** によって列挙できます。ちなみに、ここに示している図は、木の形をしていることから**決定木**（*decision tree*）と呼ばれます。

左端の枠から始めて、( ) 内の条件が成立すれば上側の黒線を、成立しなければ下側の青線をたどっていきます。

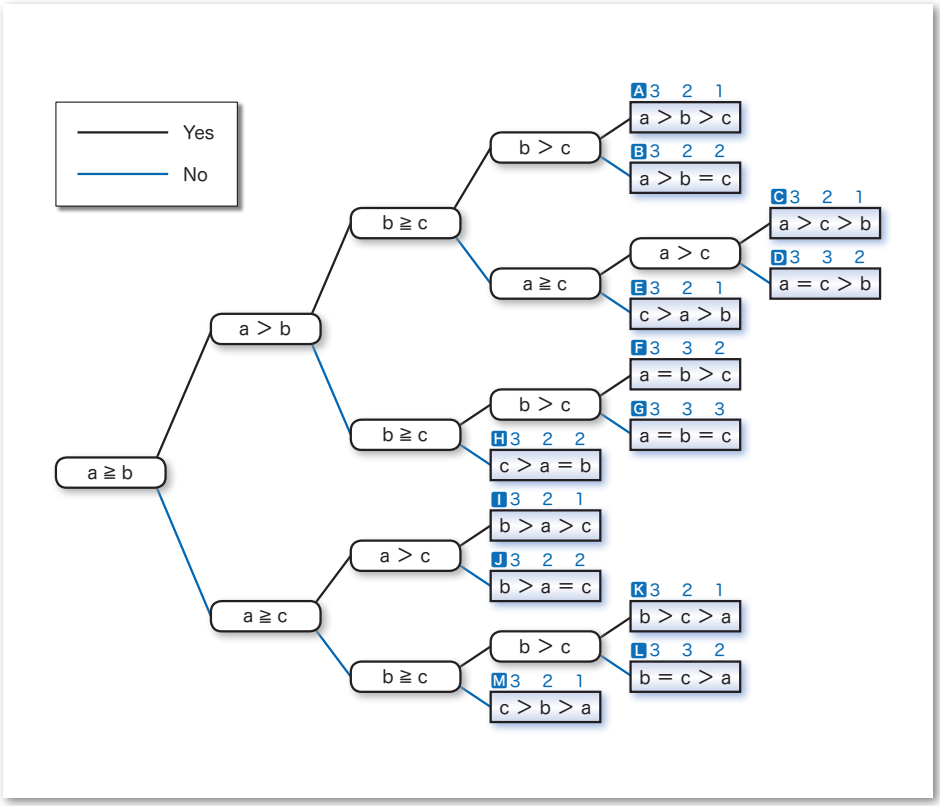


Fig.1C-4 三値 a, b, c の大小関係を列挙する決定木

右端の [ ] 内に示すのが、三つの変数 a，b，c の大小関係であり、その上に示す青い数値は、**List 1-2** のプログラムで利用した、三つの変数の値です（プログラムでは、A , B , …, M の 13 種類に対して、最大値を求めていました）。

＊

なお、最大値・最小値とは異なり、中央値を求める手続きは、非常に複雑です（そのため、数多くのアルゴリズムが考えられます）。

List 1C-1 に示すのが、プログラムの一例です。各 `return` 文の横の A , B , …, M は、**Fig.1C-4** と対応しています。

＊

なお、三値の中央値を求める手続きは、『クイックソート』の改良アルゴリズム（第 6 章）などで応用されます。

List 1C-1 chap01/med3.c

```
/* 三つの整数値を読み込んで中央値を求める */

#include <stdio.h>

/*--- a, b, cの中央値を求める ---*/
int med3(int a, int b, int c)
{
    if (a >= b)
        if (b >= c)
            return b;          ← A B F G
        else if (a <= c)
            return a;          ← D E H
        else
            return c;          ← C
    else if (a > c)
        return a;              ← I
    else if (b > c)
        return c;              ← J K
    else
        return b;              ← L M
}

int main(void)
{
    int a, b, c;

    printf("三つの整数の中央値を求めます。\n");
    printf("aの値：");   scanf("%d", &a);
    printf("bの値：");   scanf("%d", &b);
    printf("cの値：");   scanf("%d", &c);

    printf("中央値は%dです。\n", med3(a, b, c));

    return 0;
}
```

実行例

```
三つの整数の中央値を求めます。
aの値：1↵
bの値：3↵
cの値：2↵
中央値は2です。
```

演習 1–4

三値の大小関係 13 種類すべての組合せに対して中央値を求めて表示するプログラムを作成せよ。

※ヒント：**List 1-2** と **List 1C-1** を参考にして（うまく組み合わせて）作ること。

演習 1–5

中央値を求める関数は、以下のようにも実現できる。ただし、**List 1C-1** に示す *med3* と比較すると実行効率が悪い。その理由を説明せよ。

```
int med3(int a, int b, int c)
{
    if ((b >= a && c <= a) || (b <= a && c >= a))
        return a;
    else if ((a > b && c < b) || (a < b && c > b))
        return b;
    return c;
}
```

条件判定と分岐

List 1-3 は、読み込んだ整数値の符号（正／負／0）を判定・表示するプログラムです。本プログラムを通じて、プログラムの流れの分岐に対する理解を深めましょう。

List 1-3 chap01/sign.c

```
/* 読み込んだ整数値の符号（正／負／0）を判定 */
#include <stdio.h>

int main(void)
{
    int n;

    printf("整数を入力せよ：");
    scanf("%d", &n);

    if (n > 0)
        printf("それは正です。\n");  ←1
    else if (n < 0)
        printf("それは負です。\n");  ←2
    else
        printf("それは0です。\n");   ←3

    return 0;
}
```

実行例 1
```
整数を入力せよ：5↵
それは正です。
```

実行例 2
```
整数を入力せよ：-5↵
それは負です。
```

実行例 3
```
整数を入力せよ：0↵
それは0です。
```

Fig.1-3 に示すのは、網かけ部のフローチャートです。変数 n の値が正であれば 1 が実行され、負であれば 2 が実行され、0 であれば 3 が実行されます。もちろん、実行されるのは、いずれか**一つだけ**です。どれか二つが実行されたり、一つも実行されなかったり、ということはありません。というのも、プログラムの流れは**三つに分岐している**からです。

ここで、ちょっとした実験をします。プログラムの網かけ部を、右ページ リスト1 のように書きかえたプログラムを作ってみましょう。

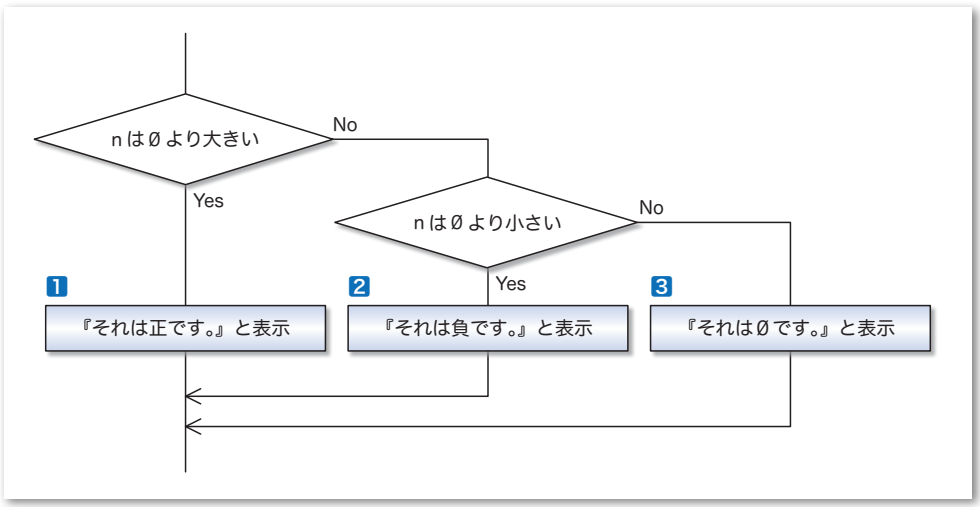


Fig.1-3 変数 n の符号の判定

リスト1

```
if (n == 1)
    printf("それは１です。\n");←1
else if (n == 2)
    printf("それは２です。\n");←2
else if (n == 3)
    printf("それは３です。\n");←3
```

n の値が 1 であれば1が、2 であれば2が、3 であれば3が実行されます。

この if 文から黒網部を削ると、構文は “if（式）文 else if（式）文 else 文” となります。これは、プログラムの流れを三つに分岐する **List 1-3** と同じ形式です。ところが、プログラムの流れの分岐の様子は異なります。右に示すように、n の値が 4 でも -8 でも、とにかく 1 と 2 以外の値であれば3が実行されてしまいます。というのも、網かけ部を削る前の リスト1 は、以下の if 文と同じ働きをしているからです。

```
整数を入力せよ：4↵
それは３です。
```

```
if (n == 1)
    printf("それは１です。\n");
else if (n == 2)
    printf("それは２です。\n");
else if (n == 3)
    printf("それは３です。\n");
else
    ;           /* 空文（実質的に何もしない文）*/
```

プログラムの流れは、実質的に**四つに分岐しています。List 1-3** の if 文とは構造が異なるため、黒網部は削れないのです。

Column 1-5　条件演算子

三つのオペランドをもつ３項演算子 ? : は、**条件演算子**（*conditional operator*）と呼ばれます。**条件式**（*conditional expression*）で行われる評価の様子をまとめたのが、**Fig.1C-5** です。

たとえば、

```
max = a < b ? a : b;
```

で変数 max に代入される値は、a が b より小さければ a の値、そうでなければ b の値です。

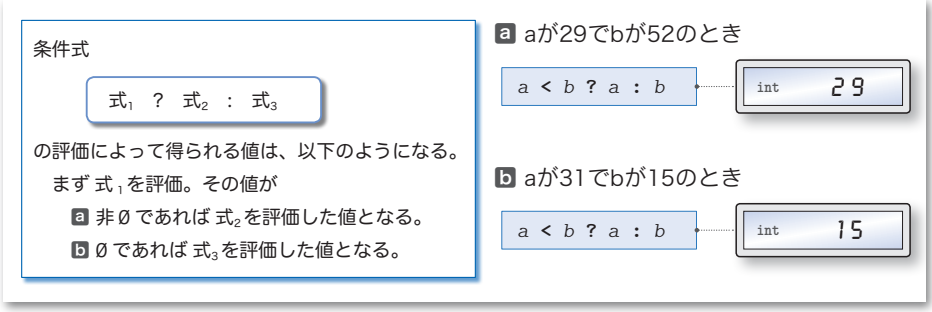


Fig.1C-5　条件式の評価

フローチャート（流れ図）の記号

問題の定義・分析・解法の図的表現である**流れ図＝フローチャート**（*flowchart*）と、その記号は、以下の規格で定義されています。

> JIS X0121『**情報処理用流れ図・プログラム網図・システム資源図記号**』

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、以下に示す記号から構成されます。

- 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（**Fig.1-4**）。

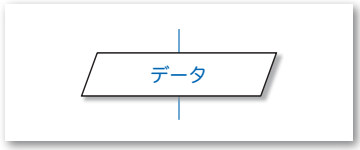


Fig.1-4 データ

処理（process）

任意の種類の処理機能を表します（**Fig.1-5**）。

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。

Fig.1-5 処理

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（**Fig.1-6**）。

Fig.1-6 定義済み処理

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（**Fig.1-7**）。

想定される評価結果は、経路を表す線の近くに書きます。

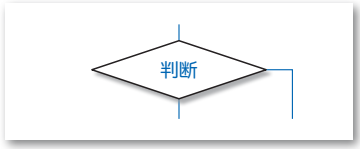


Fig.1-7 判断

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（**Fig.1-8**）。記号の二つの部分には、**同じ名前**を与えます。

Fig.1-9 に示すように、ループの**始端記号**（前判定繰返しの場合）またはループの**終端記号**（後判定繰返しの場合）の中に、初期値（初期化）・増分・終了値（終了条件）を表記します。

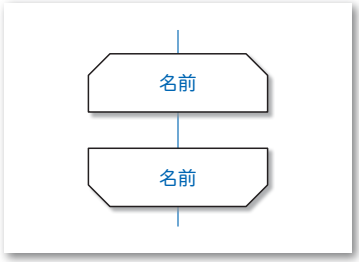


Fig.1-8 ループ端

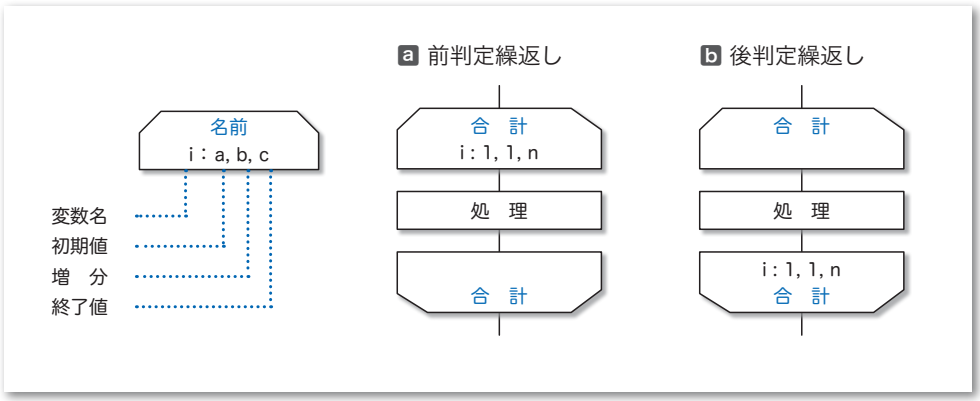


Fig.1-9 ループ端と初期値・増分・終了値

▶ 図**a**と図**b**に示すのは、変数 i の値を 1 から n まで 1 ずつ増やしながら、『処理』を n 回繰り返すフローチャートです。なお、1，1，n の代わりに、1，2，…，n という表記を用いることもあります。

線（line）

制御の流れを表します（**Fig.1-10**）。

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、明示の必要がない場合も、見やすくするために矢先を付けても構いません。

Fig.1-10 線

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（**Fig.1-11**）。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。

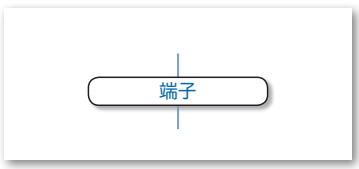


Fig.1-11 端子

1–2 繰返し

本節では、プログラムの流れを繰り返すことによって実現される、単純なアルゴリズムを学習します。

1 から n までの整数の和を求める

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。求めるのは、*n* が 2 であれば 1 + 2 で、*n* が 3 であれば 1 + 2 + 3 です。すなわち、一般的に表すと、右の式の値を求めることになります。

$$1 + 2 + \cdots + n$$

プログラムを **List 1-4** に、網かけ部のフローチャートを **Fig.1-12** に示します。

List 1-4 chap01/sum_while.c

```
/* 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;                                                    ①
    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 */     ②
    }
    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↵
1から5までの和は15です。
```

while 文による繰返し

ある条件が成立しているあいだ、処理（文または命令の集まり）が繰り返し実行されるのは、**繰返し**（*repetition*）構造であり、一般に**ループ**（*loop*）と呼ばれます。

while 文が制御するのは、繰返しを続けるかどうかを処理実行の前に判定するループである**前判定繰返し**です。

その形式は、以下のようになっています。制御式の評価によって得られる値が非 0 である限り、文が繰り返し実行されます。

```
while（制御式）文
```

なお、繰返しの対象となる**文**のことを、文法上、**ループ本体**と呼びます。

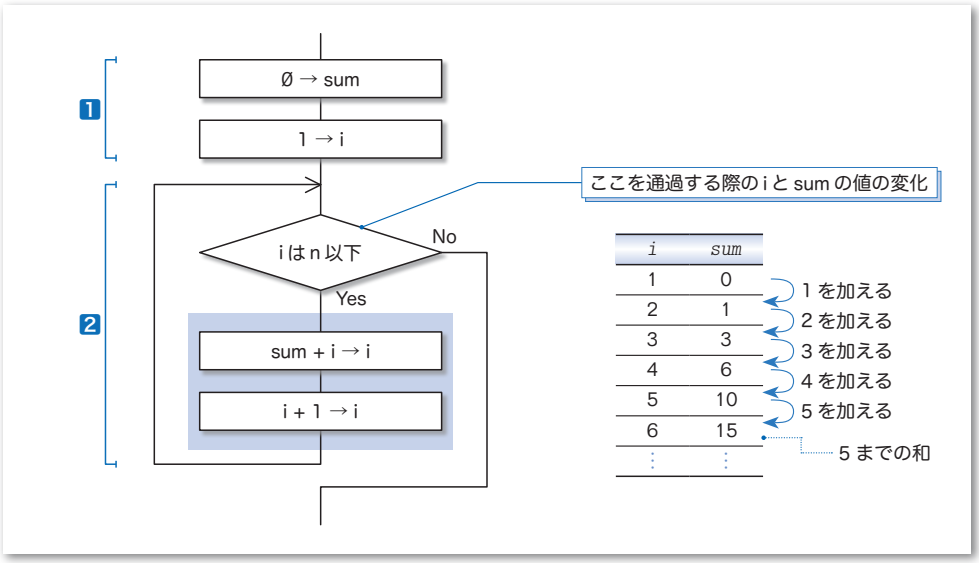


Fig.1-12 1 から n までの和を求めるフローチャートと変数の変化

プログラムとフローチャートの1と2を理解しましょう。

1 和を求めるための前準備です。和を格納するための変数 *sum* の値を Ø にして、繰返しを制御するための変数 *i* の値を 1 にします。

2 変数 *i* の値が *n* 以下であるあいだ、*i* の値を一つずつ増やしていきながら、ループ本体を繰り返し実行します。繰り返すのは *n* 回です。

▶ 2項の複合代入演算子 **+=** は、**右辺の値を左辺に加えます**。また、単項の増分演算子 **++** は、**オペランドをインクリメントします（値を一つ増やします）**。

i が *n* 以下であるかどうかを判定する制御式 `i <= n`（フローチャートの ◇ ）を通過する際の変数 *i* と *sum* の値の変化をまとめたのが、図の右側の表です。プログラムと表を見比べながら理解しましょう。

制御式を初めて通過する際の変数 *i* と *sum* の値は1で設定した 1 と Ø です。その後、繰返しが行われるたびに変数 *i* の値はインクリメントされて一つずつ増えていきます。

変数 *sum* に入っている値は『それまでの和』であり、変数 *i* に入っている値は『次に加えることになる値』です。たとえば、*i* が 5 のときの変数 *sum* の値は『1 から 4 までの和』である 10 です（すなわち変数 *i* の値である 5 が加算される前の値です）。

なお、*i* の値が *n* を超えたときに **while** 文の繰返しが終了するため、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。

演習 1–6

List 1-4 の **while** 文終了時点における変数 *i* の値が *n* + 1 となることを確認せよ（変数 *i* の値を表示するように書きかえたプログラムを作成すること）。

for 文による繰返し

単一の変数の値で制御する繰返しは、**while** 文ではなく **for** 文を用いたほうがスマートに実現できます。

1 から *n* までの整数の和を **for** 文で求めるように書きかえたプログラムが **List 1-5** です。

List 1-5 chap01/sum_for.c

```
/* 1, 2, …, nの和を求める（for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;
    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：5↵
1から5までの和は15です。
```

和を求める網かけ部のフローチャートを **Fig.1-13** に示します。

六角形の**ループ端**（*loop limit*）は、繰返しの**開始点**と**終了点**を指示する記号です。同じ名前をもったループ始端とループ終端とで囲まれた部分が繰り返されます。

したがって、変数 *i* の値を 1，2，3，… と、1 から *n* まで 1 ずつ増やしながら、ループ本体内の文 *sum* += *i*; を実行することになります。

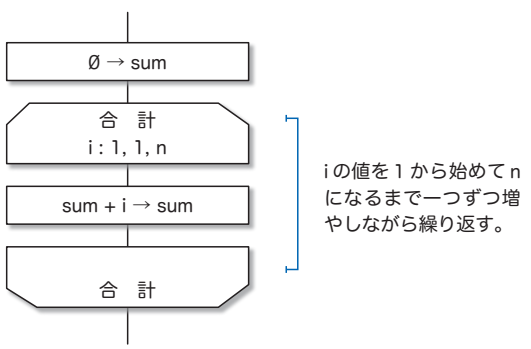


Fig.1-13　1 から n までの和を求めるフローチャート

for 文は、以下に示す形式です。

for（式$_1$; 式$_2$; 式$_3$）文

式$_1$ は、最初に（すなわち繰返しが行われる前に）一度だけ評価・実行されます。その後、**制御式**と呼ばれる式$_2$ を評価した値が非 0 である限り、ループ本体である**文**が繰り返し実行されます。その際、**文**を実行した直後に式$_3$ が評価・実行されることになっています。

▶ すなわち、以下に示す **for** 文と **while** 文は等価です。

```
/*--- for文 ---*/
for（式1; 式2; 式3）
    文
```

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

演習 1–7

List 1-5 のプログラムをもとにして、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』と表示するプログラムを作成せよ。

演習 1–8

たとえば、1 から 10 までの和は（1 + 10）* 5 によって求められる。ガウスの方法と呼ばれる、この方法を用いて、1 から *n* までの整数の和を求めるプログラムを作成せよ。

演習 1–9

整数 a，b を含め、そのあいだ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);
```

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 を求めること。

Column 1-6 非ゼロは真でありゼロは偽である

Column 1-2（p.17）では、関係演算子と等価演算子が、大小関係や等値関係の判定が成立すれば（真であれば）**int** 型の 1 を、成立しなければ（偽であれば）**int** 型の 0 を生成することを学習しました。

C 言語では、値 0 を偽とみなし、0 でないすべての値を真とみなすことを覚えておきましょう。1 でも 100 でも、とにかく 0 でなければ真です。したがって、

```
if（a）printf("ABC");
```

を実行すると、変数 a の値が 0 でなければ（1 でも 100 でも -2 でも）「ABC」と表示されます。

正の値の読込み

List 1-5 のプログラム（p.28）を実行して、変数 *n* に対して負の値である -5 を入力してみましょう。次のように表示されます。

```
1 から -5 までの和は 0 です。
```

これは、数学的に不正である以前に、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 1-6** に示します。

List 1-6 chap01/sum_for_pos.c

```
/* 1, 2, …, nの和を求める（do文によって正の整数値のみをnに読み込む）*/

#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++) {          /* 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：-6
nの値：0
nの値：10
1から10までの和は55です。
```

nが0より大きくなるまで繰り返す

実行例に示すように、*n* の値として 0 以下の値が入力されると、再び「n の値：」と表示して再入力を促します。

その実現のために利用しているのが、以下の構文をもつ do 文です。

```
do 文 while（制御式）;
```

▶ while 文や for 文とは異なり、この構文の末尾にはセミコロン ; が付きます。

do 文は、処理を行った後に、繰返しを続けるかどうかの判断を行う**後判定繰返し**を実現する文です。( ) の中の制御式を評価した値が非ゼロである限り、ループ本体である**文**が繰り返し実行されます。

したがって、プログラム網かけ部のフローチャートは **Fig.1-14** となります。

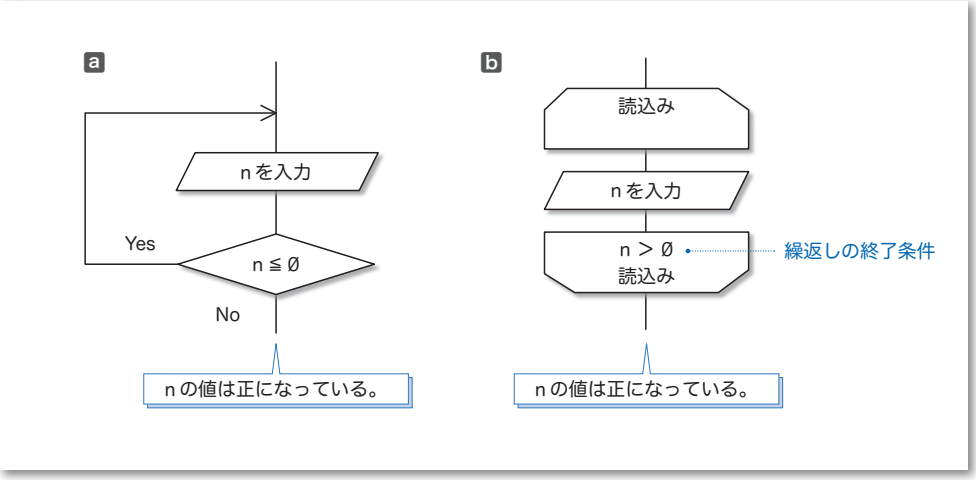


Fig.1-14 正の値の読込み

図aと図bのフローチャートは、本質的には同じです。もっとも、繰返しの終了条件を下側のループ端に書く図bは、**前判定繰返しとの見分けがつきにくい**ため、図aの書き方が好まれるようです。

*

さて、この do 文では、変数 n に読み込まれた値が 0 以下である限り、ループ本体の実行が繰り返されます。そのため、do 文終了時の n の値は必ず正になります。

*

前判定繰返しを行う while 文と for 文では、最初に制御式を評価した結果が 0 であれば、ループ本体は一度も実行されません。一方、後判定繰返しを行う do 文では、ループ本体が必ず一度は実行されます。これが、前判定繰返しと後判定繰返しの大きな違いです。

演習 1–10

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、変数 b の値を再入力させること。

```
aの値：6↵
bの値：6↵
aより大きな値を入力せよ！
bの値：8↵
b - aは2です。
```

演習 1–11

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。

構造化プログラミング

単一の入り口点と単一の出口点とをもつ構成要素だけを用いて、階層的に配置してプログラムを構成する手法を、**構造化プログラミング**（*structured programming*）といいます。構造化プログラミングでは、順次、選択、繰返しの３種類の制御の流れを利用します。

▶ 構造化プログラミングは、**整構造プログラミング**とも呼ばれます。

Column 1-7 論理演算とド・モルガンの法則

p.30で学習した**List 1-6**は、キーボードから読み込む値を《正値》に限定するプログラムでした。**List 1C-2** に示すのは、読み込む値を《２桁の正の整数値》に限定するプログラムです。

List 1C-2 chap01/dbl_digits.c

```
/* ２桁の正の整数値（10～99）を読み込む */
#include <stdio.h>

int main(void)
{
    int no;

    printf("２桁の整数値を入力してください。\n");

    do {
        printf("値は：");
        scanf("%d", &no);
    } while (no < 10 || no > 99);

    printf("変数noの値は%dになりました。\n", no);

    return 0;
}
```

実行例

```
２桁の整数値を入力してください。
値は：5
値は：105
値は：57
変数noの値は57になりました。
```

読み込む値に制限を設けるために do 文を利用している点は、**List 1-6** と同じです。ただし、本プログラムでは、網かけ部の制御式によって、変数 *no* に読み込んだ値が 10 より小さいか、もしくは 99 より大きければ、ループ本体を繰り返すようになっています。

ここで利用している || は、論理和を求める論理和演算子です。そして、論理演算を行う、もう一つの演算子が、論理積を求める論理積演算子 && です。

これらの演算子の働きをまとめたのが、**Fig.1C-6** です。

a 論理積　両方とも真であれば真

x	*y*	*x* && *y*
非０	非０	1
非０	０	０
０	非０	０
０	０	０

b 論理和　一方でも真であれば真

x	*y*	*x* || *y*
非０	非０	1
非０	０	1
０	非０	1
０	０	０

Fig.1C-6 論理積演算子と論理和演算子

▪ 論理演算子の短絡評価

no に読み込んだ値が 5 であったとします。その場合、式 *no* < 10 を評価した値は 1 となりますので、右オペランドの *no* > 99 を判定するまでもなく、制御式 *no* < 10 || *no* > 99 の評価値が 1 となることが分かります（左オペランド *x* と右オペランド *y* の一方でも非 0 であれば、論理式 *x* || *y* の評価値が 1 となるからです）。

そのため、|| 演算子の左オペランドを評価した値が 1 であれば、**右オペランドの評価は行われない**ことになっています。

同様に、&& 演算子の場合は、左オペランドを評価した値が 0 であれば、**右オペランドの評価は行われない**ことになっています（もし一方でも 0 であれば、式全体が 0 となることが明確になるからです）。

このように、論理演算の式全体の評価結果が、左オペランドの評価の結果のみで明確になる場合に、右オペランドの評価が行われないことを**短絡評価**（*short circuit evaluation*）と呼びます。

▪ ド・モルガンの法則

プログラムに戻りましょう。網かけ部の制御式を、論理否定演算子 ! を用いて書きかえると、以下のようになります（論理否定演算子は、オペランドが非 0 であれば 0 を生成し、オペランドが 0 であれば 1 を生成する、単項演算子です）。

```
!(no >= 10 && no <= 99)
```

『"各条件の否定をとって、論理積・論理和を入れかえた式" の否定』が、もとの条件と同じになることを、**ド・モルガンの法則**（*De Morgan's laws*）といいます。この法則を一般的に示すと、以下のようになります。

① *x* && *y* と !(!*x* || !*y*) は等しい。
② *x* || *y* と !(!*x* && !*y*) は等しい。

プログラムの制御式 *no* < 10 || *no* > 99 が、繰返しを続けるための《**継続条件**》であるのに対し、上記の式 !(*no* >= 10 && *no* <= 99) は、繰返しを終了するための《**終了条件**》の**否定**です。

すなわち、**Fig.1C-7** に示すイメージです。

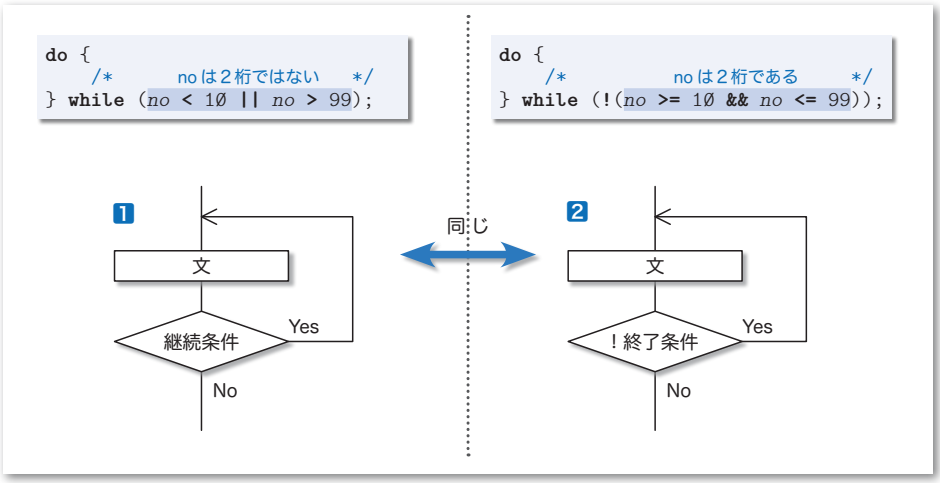


Fig.1C-7 繰返しの継続条件と終了条件

多重ループ

ここまでのプログラムは、単純な繰返しを行うものでした。繰返しの中で繰返しを行うこともできます。そのような繰返しは、ループの入れ子の深さに応じて、**二重ループ**、**三重ループ**、… と呼ばれます。もちろん、その総称は、**多重ループ**です。

九九の表

二重ループを用いたアルゴリズムの例として、《九九の表》を表示するプログラムを **List 1-7** に示します。

List 1-7 chap01/multi99table.c

```
/* 九九の表を表示 */

#include <stdio.h>

int main(void)
{
    int i, j;

    printf("----- 九九の表 -----\n");

    for (i = 1; i <= 9; i++) {
        for (j = 1; j <= 9; j++)
            printf("%3d", i * j);
        putchar('\n');
    }

    return 0;
}
```

実行結果

```
----- 九九の表 -----
  1  2  3  4  5  6  7  8  9
  2  4  6  8 10 12 14 16 18
  3  6  9 12 15 18 21 24 27
  4  8 12 16 20 24 28 32 36
  5 10 15 20 25 30 35 40 45
  6 12 18 24 30 36 42 48 54
  7 14 21 28 35 42 49 56 63
  8 16 24 32 40 48 56 64 72
  9 18 27 36 45 54 63 72 81
```

九九の表の表示を行う網かけ部のフローチャートを **Fig.1-15** に示しています。右側の図は、変数 i と j の値の変化を●と🔵で表したものです。

外側の for 文（行ループ）は、変数 i の値を 1 から 9 までインクリメントします。各繰返しは、表の 1 行目、2 行目、…、9 行目に対応します。すなわち、**縦方向の繰返し**です。

その各行で実行される内側の for 文（列ループ）は、変数 j の値を 1 から 9 までインクリメントします。これは、各行における**横方向の繰返し**です。

変数 i の値を 1 から 9 まで増やす《行ループ》は 9 回繰り返されます。その各繰返しで、変数 j の値を 1 から 9 まで増やす《列ループ》が 9 回繰り返されます。《列ループ》終了後の改行の出力は、次の行へと進むための準備です。

したがって、この二重ループでは、次のように処理が行われることになります。

- i が 1 のとき：j を 1 ⇨ 9 とインクリメントしながら 1 * j を表示。そして改行。
- i が 2 のとき：j を 1 ⇨ 9 とインクリメントしながら 2 * j を表示。そして改行。
- i が 3 のとき：j を 1 ⇨ 9 とインクリメントしながら 3 * j を表示。そして改行。

  … 中略 …

- i が 9 のとき：j を 1 ⇨ 9 とインクリメントしながら 9 * j を表示。そして改行。

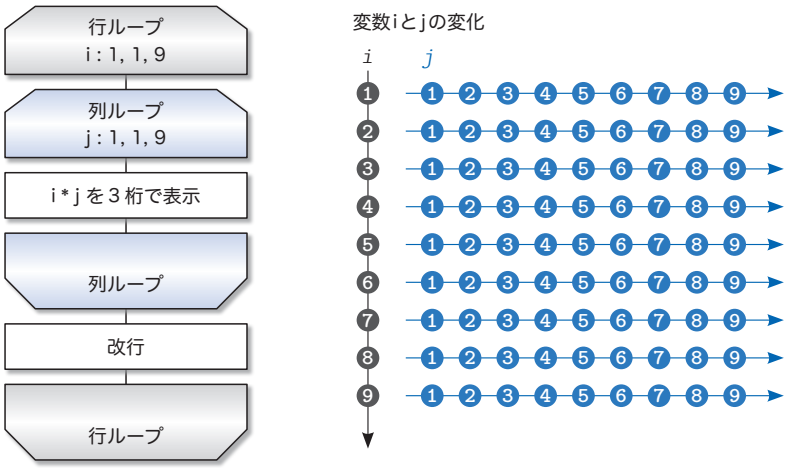


Fig.1-15 九九の表を示すフローチャート

演習 1–12

右のように、上と左に掛(か)ける数が付いた九九の表を表示するプログラムを作成せよ。

表示には、縦線記号文字 '|'、マイナス記号文字 '−'、プラス記号文字 '+' を用いること。

```
   |  1  2  3  4  5  6  7  8  9
---+--------------------------
 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
```

演習 1–13

九九の掛け算ではなく**足し算**を行う表を表示するプログラムを作成せよ。前問と同様に、表の上と左に足す数を表示すること。

演習 1–14

右のように、読み込んだ段数を一辺としてもつ正方形を * 記号で表示するプログラムを作成せよ。

```
正方形を表示します。
段数は：4↵
****
****
****
****
```

演習 1–15

右のように、読み込んだ高さと横幅をもつ長方形を * 記号で表示するプログラムを作成せよ。

```
長方形を表示します。
高さは：3↵
横幅は：7↵
*******
*******
*******
```

直角二等辺三角形の表示

二重ループを応用すると、記号文字を並べて三角形や四角形などの図形を表示することができます。**List 1-8** に示すのは、左下側が直角の二等辺三角形を表示するプログラムです。

▶ 黒網部の do 文の働きで、変数 n に読み込む値を正の値のみに限定しています。

List 1-8 chap01/triangleLB.c

```
/* 左下側が直角の二等辺三角形を表示 */
#include <stdio.h>

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

    do {
        printf("何段の三角形ですか：");
        scanf("%d", &n);
    } while (n <= 0);

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

    return 0;
}
```

段数としては正値を読み込む

実行例

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

直角二等辺三角形の表示を行う青網部のフローチャートが **Fig.1-16** です。右側の図は、変数 i と j の変化を表したものです。

実行例のように、n の値が 5 である場合を例にとって、どのように処理が行われるかを考えましょう。

外側の **for** 文（行ループ）では、変数 i の値を 1 から n すなわち 5 までインクリメントします。これは、三角形の各行に対応する**縦方向の繰返し**です。

内側の **for** 文（列ループ）は、変数 j の値を 1 から i までインクリメントしながら表示を行います。これは、各行における**横方向の繰返し**です。

*

したがって、この二重ループは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 ⇨ 1 とインクリメントしながら * を表示。そして改行。 *
- i が 2 のとき：j を 1 ⇨ 2 とインクリメントしながら * を表示。そして改行。 **
- i が 3 のとき：j を 1 ⇨ 3 とインクリメントしながら * を表示。そして改行。 ***
- i が 4 のとき：j を 1 ⇨ 4 とインクリメントしながら * を表示。そして改行。 ****
- i が 5 のとき：j を 1 ⇨ 5 とインクリメントしながら * を表示。そして改行。 *****

すなわち、三角形を上から第 1 行〜第 n 行と数えると、第 i 行目に i 個の記号文字 '*' を表示して、最終行である第 n 行目には n 個の記号文字 '*' を表示することになります。

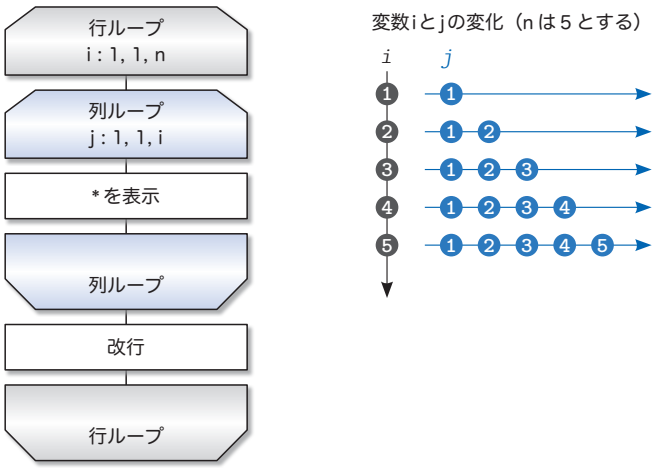


Fig.1-16 左下側が直角の二等辺三角形を表示するフローチャート

演習 1–16

直角二等辺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 triangleLB(int n);        /* 左下側が直角の二等辺三角形を表示 */
```

さらに、直角が左上側、右上側、右下側の二等辺三角形を表示する関数を作成せよ。

```
void triangleLU(int n);        /* 左上側が直角の二等辺三角形を表示 */
void triangleRU(int n);        /* 右上側が直角の二等辺三角形を表示 */
void triangleRB(int n);        /* 右下側が直角の二等辺三角形を表示 */
```

演習 1–17

n 段のピラミッドを表示する関数を作成せよ（右は 4 段の例）。

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

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

第 *i* 行目には（*i* - 1）* 2 + 1 個の記号文字 '*' を表示すること（そのため、最終行の第 *n* 行目には（*n* - 1）* 2 + 1 個の記号文字 '*' を表示することになる）。

演習 1–18

右のように、下を向いた *n* 段の数字ピラミッドを表示する関数を作成せよ。

```
1111111
22222
333
4
```

```
void nrpira(int n);
```

第 *i* 行目に表示する数字は *i* % 10 によって求めること。

章末問題

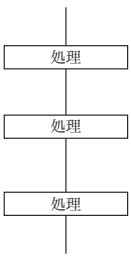
各章の章末に示しているのは、基本情報技術者試験（旧・第２種情報処理技術者試験）で出題された問題の一部です。章末問題の解答は、付録１（p.413 ～）にまとめています。

なお、章末問題を含め、平成６年度秋期～平成 22 年度秋期の基本情報技術者試験の問題と、その解答および解説を付属ディスクに収録しています（p.426 をご覧ください）。

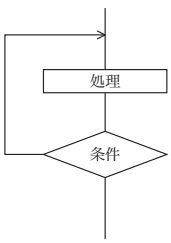
▪ 平成9年度（1997年度）秋期 午前 問37

次のプログラムの制御構造のうち、選択構造はどれか。

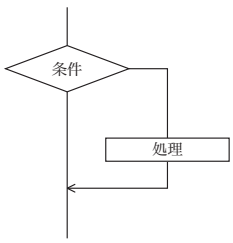
ア



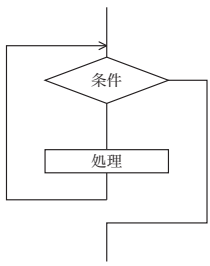
イ



ウ



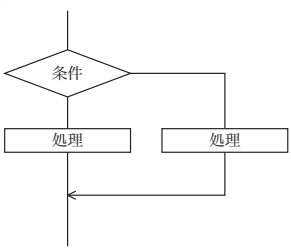
エ



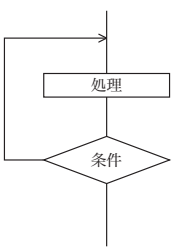
▪ 平成18年度（2006年度）春期 午前 問36

プログラムの制御構造のうち、while 型の繰返し構造はどれか。

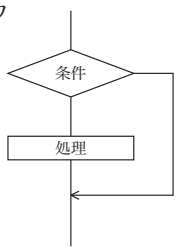
ア



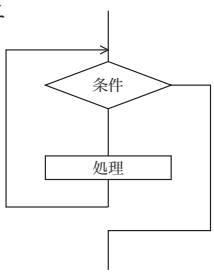
イ



ウ



エ



▪ 平成16年度（2004年度）秋期 午前 問41

プログラムの制御構造に関する記述のうち、適切なものはどれか。

ア “後判定繰返し”は、繰返し処理の先頭で終了条件の判定を行う。

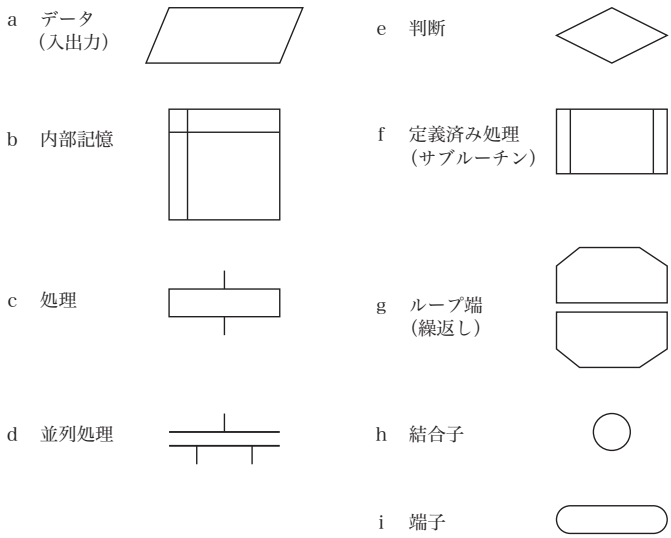
イ “双岐選択”は、前の処理に戻るか、次の処理に進むかを選択する。

ウ “多岐選択”は、二つ以上の処理を並列に行う。

エ “前判定繰返し”は、繰返し処理の本体を１回も実行しないことがある。

▪ 平成6年度（1994年度）秋期 午前 問41

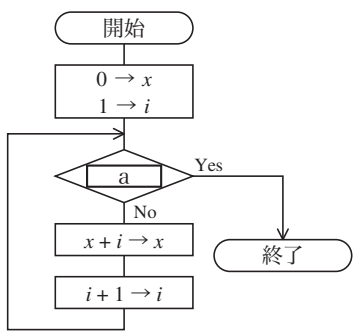
整構造プログラミング（構造化プログラミング）における基本３構造と呼ばれるものに、最も密接な関係のある流れ図記号の組合せはどれか。



ア a，b，c　　イ a，b，d　　ウ c，e，g
エ d，e，g　　オ f，h，i

▪ 平成12年度（2000年度）春期 午前 問16

次の流れ図は、1 から N（$N \geqq 1$）までの整数の総和（$1 + 2 + \cdots + N$）を求め、結果を変数 x に入れるアルゴリズムを示している。流れ図中の a に当てはまる式はどれか。



ア $i = N$　　イ $i < N$　　ウ $i > N$　　エ $x > N$

▶ 章末問題に示しているのは、基本情報技術者試験の過去問題の一部です。必ず、付属ディスクも併用して学習を進めてください。