第 4 章

条件分岐と論理型

これまで書いたプログラムは、どんな値を与えても同じ手続きで計算するものでした。それでは、例えば絶対値を返す関数を作るにはどうすればよいでしょうか。0 以上の数値であればそのまま返して構いませんが、負の値の場合は符号を反転させなければなりません。符号を反転させるには、例えば変数 a に対しては -a と書けば良いです。正の値の場合はこのマイナスは要りませんから、式を使い分ける必要があります。

二次方程式の解の個数は、判別式の値によって判定できますが、その個数を出すにはどうすれば良いでしょうか。

駐車場の自動精算機は、硬貨の中にも、1 円や 5 円のように、取り扱わないものもあります。駐車料金以上のお金を受け取ると、出庫できるようにします。どれも、状況によって行なうことが違います。

この章では、「ある条件を満たすときに限って実行する文」（条件分岐）について説明します。今までは実行時に関数を呼び出して別の場所に飛ぶことはありましたが、今回は「制御構文」と呼ばれる、実行の流れを変える方法に触れます。

キーワード

- 条件分岐，多重分岐，入れ子
- if ... else if ... else
- 条件式, 論理演算子
- 論理型（boolean），TRUE / FALSE
- ド・モルガンの法則
- 短絡評価

4.1 条件分岐の if-else と論理型

まずは簡単な**条件分岐** (conditional branch) です。

`if` 文は、最初に**条件式** (conditional expression) を検査します。成立していれば、そのすぐ後のブロック（**then 節**[*1]）を実行します。不成立であれば、`else` の後ろのブロック（**else 節**）を実行します。（else 節が省略されていれば何もしません。）

```
/* 文法 */
if (条件式) {
    成立時に実行する文; ...
} else {      // else 以降は省略可
    不成立時に実行する文; ...
}
```

```
/* 実例 */
if (a >= 10) {
    printf("10以上です\n");
} else {
    printf("10以上ではない\n");
}
```

単純な条件式として、2 つの値を比較するものがあります。例えば変数 a と 10 を比較する if 文は右上の例のようになります。比較に使う不等号は**比較演算子** (comparison operator) あるいは**関係演算子** (relational operator) と呼ばれ、次の 6 通りあります。

`==`	等しい	`>=`	以上	`>`	より大きい，超えて
`!=`	異なる	`<=`	以下	`<`	より小さい，未満

- 2 文字の演算子は、いずれも **2 文字めが =** です。
- 2 文字の演算子は、間に空白を入れて 1 文字ずつに分割してはいけません。

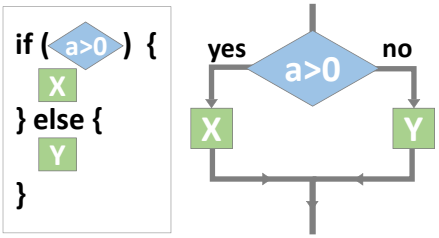
条件式の結果は、**真** (true) と**偽** (false) の 2 通りの値で表されます。比較演算子の大小関係が成立していれば真、そうでなければ偽です。このようなデータ型を、一般に**論理型** (logical datatype) とか**ブーリアン型** (boolean datatype) と呼びます[*2]。

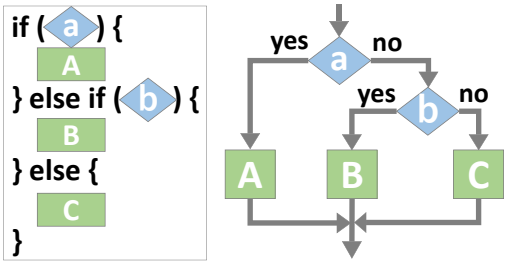
`0 == 0` → 真	`0 != 0` → 偽	`0 =! 0` → 代入（???）
`1 < 5` → 真	`1 >= 5` → 偽	`1 => 5` → コンパイルエラー

数学の表記では、不等号で変数の取りうる範囲を限定したり、場合分けの条件を示すことがあります。しかし C 言語（をはじめとする多くの手続き型言語）の不等号は、その時点の変数の値を用いて、大小関係が成立しているかどうかを**検査**して、**真偽を決定**するばかりです。「`x >= 10`」と書いたところで、x が 10 以上の値に限定されるわけではないことに注意してください。

[*1] C 言語では then のキーワードは使いませんが、言語によっては if の条件と実行文の境目に then を挿入するので、「then 節」の用語を採用しました。

[*2] C 言語で、本物の論理型が導入されたのは C99 のことで、それまでは長らく int を流用してきました。（☞13 ページのコラム）





4.1.1 多重分岐

次は、分岐先が 3 つ以上の場合の書き方です。`else if` を繰り返します。右下の例は、年齢 x によって年代を分類しています。

```
/* 文法 */
if (条件1) {
    文1;
} else if (条件2) {
    文2;
} else if (条件3) {
    文3;
...             // 繰り返してよい
} else {        // else 以降は省略可
    文;
}
```

```
/* 実例 */
if (x <= 19) {
    printf("10代以下\n");
} else if (x <= 29) {
    printf("20代\n");
} else if (x <= 39) {
    printf("30代\n");
} else {
    printf("40代以上\n");
}
```

条件 1 が成り立てば、文 1 を実行します。それ以外で、条件 2 が成り立てば、文 2 を実行します。さらにそれ以外で、条件 3 が成り立てば、文 3 を実行します。この調子で、else if は好きなだけ続けて構いません。どの条件も成り立たなければ、最後の else 節を実行することになります。このように if ... else if ... else の構文は、**どれか 1 つの文を実行**することを保証します。（最後の else 節がなければ、最大で1 つの文を実行します。）

コラム： ブロックの範囲を見やすくするインデント

インデント（字下げ）は、関数の範囲を見分けるのに役立ってきました。これからは、if のブロックの範囲を見分けるのにも活用します。

{ の次の行から、インデントを 1 段深くします。

} の行で、深さを回復します。

```
int func(int x) {
    if (x % 2 == 0) {
        printf("xは偶数\n");
        return x / 2;
    } else {
        printf("xは奇数\n");
        return x + 1;
    }
}
```

4.2 if の入れ子と論理演算

2 次元平面上の (x, y) 座標が、**原点である**ことを判定しましょう。条件を、「$x = 0$」で、しかも「$y = 0$」と考えます。

```
/* ifの入れ子 */
if (x == 0) {
    if (y == 0) {
        printf("原点である\n");
    }
}
```

```
/* 論理積 */
if (x == 0 && y == 0) {
    printf("原点である\n");
}
```

- 左上の例では、2 つの if を組み合わせました。このように、if のブロックに、さらに if を入れても構いません。（このような「あるものの中に、同じ種類のものがもう一度含まれている構造」を**入れ子** (nest) といいます。）
- 右上の例では、2 つの条件を組み合わせるのに**論理積** (and) 演算子 && を用いました。&& で結ばれた 2 つの条件の両方が成立したときに限って、全体として成立したことになる（日本語の「かつ」に相当します）ので、if は 1 つですみます。

それでは**原点ではない**ことはどのように判定すればよいでしょうか。条件は、「x != 0」と「y != 0」の、どちらが成り立っても原点ではありません。

```
/* 多重分岐 */
if (x != 0) {
    printf("原点ではない\n");
} else if (y != 0) {
    printf("原点ではない\n");
}
```

```
/* 論理和 */
if (x != 0 || y != 0) {
    printf("原点ではない\n");
}
```

- 左上の例のように、多重分岐にする[*3]と、まったく**同じ printf() を 2 回**書くことになります。これはあまりうまい方法とは言えません。
- 右上の例のように、**論理和** (or) 演算子 || はうまく働きます。どちらか片方の条件でも成り立てば、全体として成り立ったことになる（日本語の「または」に相当します）ので、1 つの if で表現できて、**printf() も 1 回**ですみます。

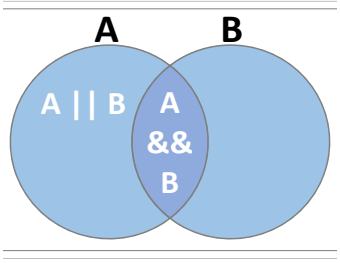
論理否定 (not) 演算子 ！ は真偽を反転するので、「〜ではない」の条件を直接的に表現できます。原点の条件の直前に ！ を書きます。演算の優先順位が高いので、否定される条件式全体をカッコで囲います。

```
/* 論理否定 */
if (!(x == 0 && y == 0)) {
    printf("原点ではない\n");
}
```

[*3] 単純に 2 つの if を並べるだけだと、printf() が 2 回とも実行される場合があるので、else if は必須です。

表 4.1 論理演算子の真偽値

入力 A	入力 B	演算結果 (積) A && B	演算結果 (和) A || B	演算結果 (否定) !A
真	真	真	真	偽
真	偽	偽		
偽	真			真
偽	偽		偽	



論理演算（積・和・否定）の働きをまとめると、表 4.1 のようになります。

頻出ミス

比較の a==1 は、a=1 とよく書き間違えますが、間違えると**代入**になります。しかも C 言語では、代入も演算子なので、文法上は if の条件式として正当です。

```
if (a == 1) { // ○比較
    printf("aは1でした");
}
```

```
if (a = 1) { // ×代入
    printf("aを1にしました!?");
}
```

比較の != を =! と書き間違えると、これも**代入**と解釈されます。（4.5 節で述べるように、!1 は 1 の否定ですから、0 です。）

```
if (a != 1) { // ○比較
    printf("aは1以外でした");
}
```

```
if (a =! 1) { // ×代入
   printf("aを!1にしました!?");
}
```

論理演算の && や || は 2 文字ですが、1 文字の & や | にすると**ビット演算**になって、エラーにならずに微妙に動作が変わります。（☞4.6 節）
このように、文法上はエラーにならず、気づきにくい間違いなので要注意です。
gcc なら、-Wall のオプションで以下のような警告も出るので、活用しましょう。

```
source.c:9:5: 警告: 真偽値として使われる代入のまわりでは、
丸括弧の使用をお勧めします [-Wparentheses]
     if (a = 1) {
     ^
```

他言語に目を向けると、Java では、if の条件式を boolean（論理型）に限ることで、このような間違いをコンパイルエラーにしています。

Pascal 言語では、代入が a:=1、比較が a=1 です。代入は右辺から左辺へという向きが重要ですから、コロンで向きを表現していると思えば、納得の文法です。

4.3 論理演算の組合せと優先順位

数学でよく目にする「$0 \leq x < 10$」の表記は、残念ながら C 言語では通用しません。C 言語の比較演算子は 2 つの値だけを比較するので、「$0 \leq x$」と「$x < 10$」に分解した上で、両方とも同時に成立して欲しいので、論理積で連結して `0<=x && x<10` とします。2 つの条件が目立つように `(0<=x) && (x<10)` と、カッコを活用するのもよいでしょう。

頻出ミス

`0 <= x < 10` は、エラーにならずに、動作だけがおかしくなります[a]。気づきにくい間違いですが、各条件をカッコで囲えばこの間違いを防げます。

```
if (0 <= x && x < 10) { // ○
    printf("xは1桁の整数です");
}
```

```
if (0 <= x < 10) { // ×
    printf("常に成立します!?");
}
```

Java ではコンパイルエラーになります。Python では、何と正常に動作します。

[a] `(0<=x)<10` と同じです。`(0<=x)` の演算結果は、4.5 節で説明するように、int 型の 0 か 1 になります。全体としては `0<10` あるいは `1<10` を判定することになり、どちらであっても成立です。

それでは、x が「偶数か、あるいは 0 以上 10 未満」の条件は、どう記述すればよいでしょうか。「または」と「かつ」の両方が出てくるので**優先順位**が問題になります。

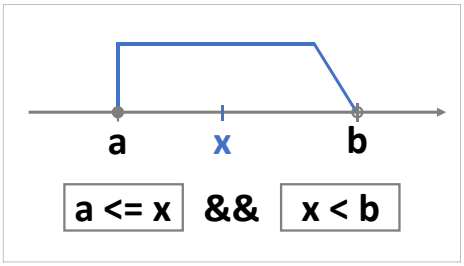
```
if ((x % 2 == 0) ||  (0 <= x) && (x < 10) ) {...} // && が || より優先
if ((x % 2 == 0) || ((0 <= x) && (x < 10))) {...} // 同じ動作（推奨）
```

文法としては、（積と和というだけあって）論理**積が先**に計算されるので、上の 2 行は同じ動作をします。それでも、優先順位を混同する人が多いためか、覚えていなくても理解できるよう、カッコを使用することが推奨されています。

論理否定の ! は、（すでに述べていますが）論理和や論理積よりも優先度が高いので、下の例のように「0 以上 10 未満」を否定するのにカッコが必要です。もしカッコを忘れて「`!(0 <= x) && ...`」とするとどうなるかは、4.5 節を参考に考えてみてください。

```
if (!( (0 <= x) && (x < 10) )) {...} // 0 <= x < 10 の否定
```

論理否定で if の条件式の真偽を反転すると、then 節と else 節のブロックを入れ替えたのと同じ効果があります。これを利用して、then 節に行なうべき処理がなければ（then 節自体は省略できないので）、論理否定で真偽を反転して else 節を省略する、という使い方をよく見かけます。**MEMO:** 書いてみたが、目立っていない。コラムにするか。




```
if ( !(条件) ) {
  成立時;
} else {
  不成立時;
}
```

4.3.1 ド・モルガンの法則

論理積や論理和に否定が組み合わさった論理式を変形するのに、便利な法則があります。

```
!(A && B)  ⟺  (!A) || (!B)
!(A || B)  ⟺  (!A) && (!B)
```

全体の真偽を反転する代わりに、個別の真偽を反転させた上で、論理積と論理和を入れ替えれば同じ、ということです。この規則を**ド・モルガンの法則** (De Morgan's laws) といいます。これを覚えておくと、機械的な書き換えができます。例えば「`x==0 && y==0`」の否定は「`x!=0 || y!=0`」とすぐにわかります。

コラム： 不等号の向き

条件式の「$x > 0$」と「$0 < x$」は、どちらもまったく同じ意味です。このような不等号の向きをどちらにするのがよいか、考え方が何通りかあります。

```
if (x > 0 && y < 0) ...
// 変数 [不等号] 定数
```

```
if (0 <= x && x < 10) ...
// 小 [不等号] 大
```

変数・定数 左上の例のように、変数と定数の比較の場合、変数を左辺にします。**左辺のほうが主体**というか、「検査対象である」という雰囲気が感じられます。両方とも変数なら、変化の頻度の多い変数を左辺にするでしょう。

数直線 右上の例のように、数直線上の大小関係を思い浮かべると理解しやすい場面では、**不等号を < と <= に限定**して、定数を小さいものから並べます。

どちらが常によいともいえないので、場面ごとに使い分ければよいでしょう。

4.4 if を羅列する弊害

いくつも分岐があるときに、同じ意味の条件式を（真偽の反対のものも含めて）何度も冗長に羅列していると、**プログラムの制御構造**が理解しにくくなります。else を適切に用いて、制御構造を簡潔に表現しましょう。いくつかある文のうち「どれか 1 つを実行する」のか「複数個を実行する可能性がある」のかは、**else のありなし**で表現できます。条件式を吟味して、ようやく動きがわかるようではバグの温床になってしまいます。

4.4.1 条件の網羅

x の**絶対値を表示**することを考えます。

```
/* if-else */    // ○
if (x > 0) {
    printf("%d", x);
} else {
    printf("%d", -x);
}
```

```
/* if の羅列 */   // ×失敗
if (x > 0) {
    printf("%d", x);
}
if (x < 0) { // !(x > 0) なら△
    printf("%d", -x);
}
```

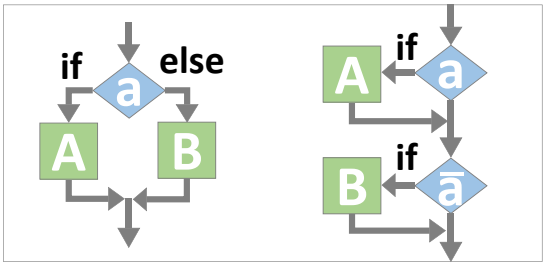
- 左上の if-else の例では、正ならば x、そうでなければ $-x$ を表示することで、絶対値を実現しています。2 つの printf() は else で振り分けられているので、どちらか片方が必ず実行されることがわかります。つまり、（何かしらの）値が 1 回だけ表示されることが、**プログラムの制御構造**から読み取れます。
- 右上の例は、else を使わずに 2 つの if を並べました。しかし問題が 2 つあります。
  (a) どちらか片方の printf() が必ず実行されるとは、**プログラムの制御構造**からは読み取れないので、動作を理解するには、条件式を吟味する必要があります。
  (b) 2 つめの if に真偽逆の条件を書いたつもりでも、$x = 0$ の場合が抜けています。

  結局のところ、冗長な記述のために条件を網羅しきれず、「$x = 0$ のときに何も表示されない」というバグを生んでしまいました。

4.4.2 条件の網羅と関数

x の**絶対値を返す関数**を作ってみます。`abs_if_else(x)` はもちろん正しく動作します。

`abs_if_if(x)` は、先ほどと同じ条件分岐にしたので、$x = 0$ のときに return に到達しません。このため、Java であればコンパイルエラーになります。C 言語ではエラーにならず（コンパイルオプションにによっては警告してくれる場合はあります）、return に到達しないときの戻り値は**不定**です。（初期化忘れの変数と同じです。）



それでは、$x = 0$ の場合が抜けないように、例えば 2 つめの条件を `!(x > 0)` としたら正しく動作するでしょうか。確かに、動作は正しくなるのですが、Java ならこれでもコンパイルエラー、C 言語でも警告は消えないでしょう。この関数が必ず return に到達するとは、**プログラムの制御構造**からは判断できないからです。

```
int abs_if_else(int x) { // ○
    if (x > 0) {
        return x;
    } else {
        return -x;
    }
}
```

```
int abs_if_if(int x) { // ×失敗
    if (x > 0) {
        return x;
    }
    if (x < 0) { // !(x > 0)なら△
        return -x;
    }
}
```

ちなみに右上の例では、2 つめの if は書かずに、「`return -x;`」だけにしてしまえば、簡単に正しいプログラムになります。というのも、1 つめの if が成立すれば return で関数ごと抜け出すので、その後の部分はそもそも else 節に入っているのと同じだからです。

4.4.3 条件に影響のある操作

x が負なら符号反転、そうでなければ 2 を加える、という操作をしてみましょう。この操作では x が変化するので、後続の条件判定に影響を及ぼすことに注意してください。

```
/* if-else */     // ○
if (x < 0) {      // xが負なら
    x = -x;       // 符号反転
} else {          // そうでなければ
    x = x + 2;    // 2を加える
}
```

```
/* if の羅列 */   // ×失敗
if (x < 0) {      // xが負なら
    x = -x;       // 符号反転
}
if (x >= 0) {     // 元が負でも
    x = x + 2;    // 2を加える
}
```

左上の if-else の例は、もちろん正しく動作します。しかし、右上の if の羅列の例は、負のときに、符号反転の上に 2 まで加えてしまいます。一度の if で動作を確定しないために、両方の if が成立してしまうことになりました。

4.4.4　if の羅列と多重分岐

年度初めの年齢 a 才によって、身分を次の表に従って 3 通りに分類してみましょう。以下の 3 つのプログラムは、どれも同じ動作をしますが、どれがよいでしょうか。

年齢	6 才～	9 才～	12 才～	15 才～
身分	小学生低学年	小学生高学年	中学生	(義務教育修了)

(A) if の羅列は、規則的で見通しは悪くないのですが、やはり弊害があります。「成立する if は最大でもひとつ」であることが**プログラムの制御構造**からは読み取れません。そして、境界の年齢を変更する（あるいは、間違いに気づいて修正する）ときには **2 ヶ所の数値**を直す必要があるため、どちらかを直し忘れそうです。

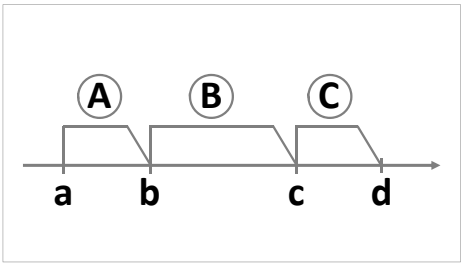
```
/* (A) ifの羅列 */
if (12 <= a && a < 15) {
    printf("中学生");
}
if ( 9 <= a && a < 12) {
    printf("小学生高学年");
}
if ( 6 <= a && a < 9) {
    printf("小学生低学年");
}
```

(B) if の入れ子ではその点は改善されていて、境界の年齢の記述が 1 ヶ所です。そして、else でつながっていることから「成立する if は最大でもひとつ」という構造も（何とか）読み解けます。（例えば「小学生高学年」の if が成立するときには、「中学生」が除外されていることに注意してください。）しかし、**インデントが深く**なり、見通しが悪くなりました。分類が増えれば、インデントは更に深くなります。

```
/* (B) ifの入れ子 */
if (a < 15) {
   if (12 <= a) {
      printf("中学生");
   } else {
      if ( 9 <= a) {
         printf("小学生高学年");
      } else {
         if ( 6 <= a) {
            printf("小学生低学年");
         }
      }
   }
}
```

(C) 多重分岐は一番うまく処理できます。`else if` の繰り返しでインデントの深さが 1 段に揃います。**同じ構文の繰り返し**であることがすぐにわかりますし、分類が増えても複雑になりません。ただし、15 才以上のときに何もしないために、ブロックの中身を空にしたところは、少し技巧的です。書き忘れでないことを示すために、コメントを残しておきました。

```
/* (C) 多重分岐 */
if        (15 <= a) {
    // 何もしない（義務教育修了）
} else if (12 <= a) {
    printf("中学生");
} else if ( 9 <= a) {
    printf("小学生高学年");
} else if ( 6 <= a) {
    printf("小学生低学年");
}
```



コラム： 常にブロックを書くべし

これまで、if や else の後にはブロックを記述すると述べてきました。本当は、ブロックの中身がたった 1 つの文（＝**単文**、つまりセミコロン (;) が 1 つだけ）であれば、{ と } を省略して、単文にできますが、お薦めしません。実際のプログラム作成の現場では、作っているプログラムが刻々と姿を変えていきます。単文にしていると、後から処理を加えるときに { } で囲うのを忘れ、インデントに惑わされて制御構造をおかしくする、という**事故**が簡単に起こります。

```
/* 元のプログラム */
if (x < 0)
    x = -x;  // △ if成立時のみ

printf("絶対値は%d\n", x);
```

```
/* printf()を書き加えると... */
if (x < 0)
    printf("符号を反転します\n");
    x = -x;  // × ifと無関係
printf("絶対値は%d\n", x);
```

右上の例は、左上の if に printf() を書き加えたのですが、{ } で囲うのを忘れました。誤ったインデントに惑わされずに「`x = -x;`」が **if と無関係に実行される**ことを見抜けるでしょうか?

```
// 単文でもブロックにする
if (x < 0) { x = -x; } // ○
```

```
// 単文では改行しない
if (x < 0) x = -x; // ○
```

最初から**単文でもブロック**にしておけば[a]、このような事態を防げます。少なくとも**単文では改行しない**のなら、ブロックが必要だと思い出せるでしょう。

ところで、C 言語の else if は、else 節を単文の if にして実現しているだけで、特別な文法があるわけではありません[b]。それでも、多重分岐の構文は「else if」と覚えるのが簡便でしょう。そしてこの if は、else 節の単文にあたりますから、「単文でもブロック」の例外とも言えます。

[a] Perl のように、そもそも単文が許されない言語もあります。
[b] Perl や Ruby のように、言語によっては専用の構文（elsif）が用意されます。

4.5　論理型の変数と関数

+ や / などの四則演算の演算子は、演算の結果を変数に代入できました。同じように、< や >= などの比較演算子も、結果を変数に代入できそうです。これまで、条件式の結果は**論理型**の「真」と「偽」のどちらかだと説明してきましたが、実際には、C 言語ではこの値を **int 型の「1」と「0」**にしています。ですから、代入する変数は int 型にします。

```
double x = 2.5, y = 3.6;
double c = x + y;// 四則演算を代入
```

```
double x = 2.5, y = 3.6;
int    c = x < y;// 比較演算を代入
```

上のプログラムはどちらも正当です。ただ、= と < が連続して出てくるのに違和感のある人もいるでしょう。書かなくてもよいカッコを書いて「`c = (x < y);`」として、条件式の結果を代入することを示す、という使い方もあります。

この性質を理解すると、次の 2 つのプログラムが、まったく同じ動作をするとわかります。**条件式を一旦 int 変数に代入**しても、動きは同じということです。

```
/* 直接の条件式 */

if (x > 0) {
    printf("xは正です\n");
}
```

```
/* 変数経由の条件式 */
int c = (x > 0);
if (c) {
    printf("xは正です\n");
}
```

関数でも、変数と同じように、条件式を返せます。return に書いた式が、そのまま関数呼び出し部分に置き換わると理解しましょう。やはり右の `is_plus()` の return のように、書かなくてもよいカッコを書くことがあります。

条件式には、論理演算が入っても大丈夫ですから、右の `is_in_range()` のような関数も作れます。

```
int is_plus(double x) {
    return (x > 0);
}

if (is_plus(x)) { // (x>0) と同じ
    printf("xは正です\n");
}
```

```
int is_in_range(int a) {
    return (0 <= a && a < 10);
}
```

このように、論理型の変数や関数は **if の条件式に単独で現れて**、比較演算のない、風変わりな表記になります。そこで、真偽値だとすぐにわかるよう、変数名や関数名に目印をつけます。つまり、is_+(形容詞) や has_+(過去分詞)、has_+(名詞) のような名前にします。この習慣を知っていれば、例えば <ctype.h> の `isalpha()` など[*4]が論理型であるとわかります。(☞4.5.2 項)

[*4] 正確には、単語の区切りの _（アンダースコア）が省略されています。標準ライブラリ関数は、昔の名残りで、極端に文字数を節約しています。

偽	FALSE(=0)
真	TRUE(=1), 2, 3, ...

4.5.1 論理型の定数

関数の処理が複雑になると、真を返したい場面と、偽を返したい場面が、別々になることがあります。変数にどちらかの値を**代入**したい場合もあります。このとき、真を表すためには「1」ではなく、右上のようにマクロで定義[*5]した「TRUE」を用いるのが習慣です。偽なら「0」ではなく「FALSE」です。これも、論理型であることを示す工夫です。（**注意：比較には用いません。**）

```
#define FALSE 0
#define TRUE  1
```

```
int is_plus2(int x) {
    if (x > 0) { // 正なら表示する
        printf("正です\n");
        return TRUE;
    }
    return FALSE;
}
```

コラム： 条件式の型は int 型

C 言語のように、条件式の型に int を流用するのは、古い言語ではよくあることでした。その C 言語でも、C99 からは 2 値に制限された bool 型が用意され、true と false の定数が使えるようになりました。既に C++ では bool 型、Java 言語では boolean 型が用意されていたので、C 言語もその流れに乗った形です。

本文では bool 型を紹介していません。int 型を流用する関数はいくつもあるので、この説明は外せません。加えて <stdbool.h> ヘッダの読み込みなど、ture false キーワードの互換性維持の仕掛け（☞??節）まで一度に学ぶのは重荷でしょう。言語の全体像がわかれば難しくないので、後から学んでも大丈夫です。

なお、一つの式で bool 型の要素を int 型の要素と混ぜて用いると、int 型に統一され、真が 1、偽が 0 に変換されます。これはこれで使い道もあるのですが、5 ページの頻出ミスのように、予期しない比較演算になることも多々あるため、Java のような混在防止の仕組みにならなかったのは残念です。

[*5] C99 より前は、真偽の定数が用意されてなかったので、プログラムで定義する必要がありました。ここでは #define の方法を紹介しましたが、列挙型 (enum) による方法もあります。

4.5.2　if に現れる論理型

ところで、論理型の変数や関数は、比較演算の結果だと理解してもらえたでしょうから、if の条件式の中で、これらの値を**もう一度 TRUE や FALSE と比較するのはおかしい**ことにも気づいてもらえるでしょう。おかしいだけでなく、（条件式の型に int を流用しているため）TRUE と比較すると害があります。なぜなら、int には TRUE(=1), FALSE(=0) 以外にもとりうる値がたくさんあって、条件式で **0 以外はすべて真**と扱われることに決まっているからです。TRUE 一つだけと比較したのでは網羅しきれません。逆に、偽であることを判定したければ[*6]、直前に否定演算子！を置いて真偽を反転させるのが慣用句です。

```
if (isalpha(c)) {   // ○ 真の判定
    printf("英字です\n");
}

if (! isalpha(c)) { // ○ 偽の判定
    printf("英字ではない\n");
}
```

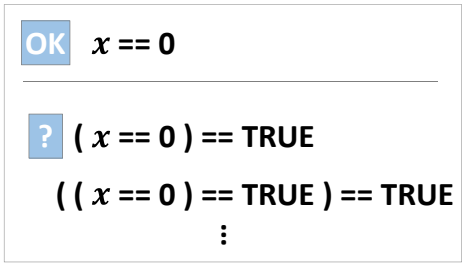
```
if (isalpha(c) == TRUE) {  // ×
    printf("英字です\n");
} // ↑どんなcでも実行されない

if (isalpha(c) == FALSE) { // △
    printf("英字ではない\n");
} // ↑動作は正しい（非推奨）
```

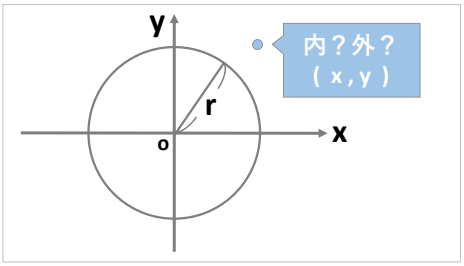
<ctype.h>（☞??節）の **is で始まる**関数のほとんどは、真として **1 以外**を返すことで有名です[*7]。左上の例とソースコード 4.1 が正しい使い方です。

ソースコード 4.1　文字の種類を見分ける

```
1  #include <stdio.h>
2  #include <ctype.h>
3
4  /* 真を返す関数を表示する */
5  void print_true_func(int c) {
6      if (isalnum(c)) { printf("isalnum('%c') ", c); } // 英字か数字
7      if (isalpha(c)) { printf("isalpha('%c') ", c); } // 英字
8      if (islower(c)) { printf("islower('%c') ", c); } // 英字の小文字
9      if (isupper(c)) { printf("isupper('%c') ", c); } // 英字の大文字
10     if (isdigit(c)) { printf("isdigit('%c') ", c); } // 数字
11     if (isspace(c)) { printf("isspace('%c') ", c); } // 空白や改行等
12     printf("\n");
13 }
14
15 int main(void) {           // 以下のものが真になる
16     print_true_func('A'); // isalnum('A') isalpha('A') isupper('A')
17     print_true_func('z'); // isalnum('z') isalpha('z') islower('z')
18     print_true_func('0'); // isalnum('0') isdigit('0')
19     print_true_func(' '); // isspace(' ')
20 }
```

[*6] FALSE との比較は、FLASE の値が 1 種類だけであるため、（非推奨ながらも）正しく動作します。
[*7] テーブル参照とビット演算を組み合わせた、効率のよい実装が知られています。





ソースコード 4.2 は、円の内側判定を行います。原点を中心とする半径 r の円が、点 (x, y) を含むなら、関数 is_in_circle(r, x, y) は論理型の TRUE を返します。この関数には **if 文がない**ことに注目してください。

ソースコード 4.2 円の内側判定

```
#include <stdio.h>

/* 原点が中心で半径rの円が、点(x,y)を含むなら論理型のTRUE、
   そうでなければFALSEを返す */
int is_in_circle(double r, double x, double y) {
    return (x * x + y * y <= r * r);
}

void print_is_in_circle(double r, double x, double y) {
    printf("原点を中心とする半径 %g の円は、点(%g,%g)を", r, x, y);
    if (is_in_circle(r,x,y)) {
        printf("含む\n");
    } else {
        printf("含まない\n");
    }
}

int main(void) {
    print_is_in_circle(1.4143, 1.0, 1.0); // 含む
    print_is_in_circle(1.4142, 1.0, 1.0); // 含まない
    return 0;
}
```

コラム： 自分に厳しく、他人に優しく

TRUE は「1」と定義するのが慣用句ですが、「2」と定義しても構いません。（正確に言うと、2 と定義して動作が変わるなら TRUE の**使い方が間違って**います。）

C コンパイラは、比較演算で論理値を得るときには、自分に厳しく 0 / 1 に限定しますが、if で評価する際には、他人に優しく **0 以外はすべて真**だと受け入れます。真を 2 で示す、甘えた (?) 変数や関数も許容されます。

4.6　論理積と論理和の短絡評価（発展的内容）

整数 m, n (≥ 0) について、m が n の倍数かどうかを判定してください。

普通に考えると、m を n で割った余りが 0 かどうかで判断できそうです。しかし、$n = 0$ のときは、割り算が実行できません（☞**??**節）。このため、剰余計算の前に除数 n が 0 でないことを確かめる必要があります。この部分の処理を 2 通り書いてみました。

```
/* ifの入れ子 */
if (n != 0) {
    if (m % n == 0) { // OK
        printf("倍数です\n");
    }
}
```

```
/* 論理積 */
if (n != 0 && m % n == 0) { // OK
    printf("倍数です\n");
}
```

左上の if の入れ子の例は、もちろん思い通りの動きをします。$n = 0$ ならば、2 つめの if は実行されないので、条件式の 0 の割り算を回避します。では、右上の論理積の例はどうでしょうか。驚いたことに、これも同じ動作をします。どういう仕組みになってるのでしょうか。

C 言語の論理和と論理積は、全体としての結果が判明した時点で、それより後の式を検査しません。これを**短絡評価** (short-circuit evaluation) といいます。具体的には次のような動作をします。

A && B　A が**偽**であれば、B を検査することなく、即座に全体として**偽**に決まります。
　　A が**真**であれば、B を検査して、その結果がそのまま全体の結果になります。

A || B　A が**真**であれば、B を検査することなく、即座に全体として**真**に決まります。
　　A が**偽**であれば、B を検査して、その結果がそのまま全体の結果になります。

「0 の倍数は 0 だけ」であることを加味すると、プログラムの全体はソースコード 4.3 のようになるでしょう。

(A) `print_multi_nest()` は if の入れ子で、条件に無駄がありません。すべての場合分けがプログラム上に現れているので、例えば「倍数ではありません」を表示させる改造も簡単です。しかし、インデントが深くなって、行数も多いです。（論点が変わりますが、最初の if の条件に否定がありながら else 節もあるので、条件を反転して then 節と else 節を入れ替えたい人もいるでしょう。）

(B) `print_multi_and()` は n の比較を 2 回行なうものの、論理積のおかげで行数が短く、しかも直後に起こる問題を回避しているというのが読み取りやすいです。ただし、一体化しすぎてて「倍数ではありません」を表示させるには手間がかかります。

FALSE && B ⇔ FALSE
TRUE && B ⇔ B
TRUE || B ⇔ TRUE
FALSE || B ⇔ B

このように、どちらにも利点欠点があるので、必ずこうすべき、とは考えないでください。

なお、似た演算にビット演算の & や | があります。これらは短絡評価をせず、必ず両辺の値を得てから、ビットごとの積や和を求めます。つまり、if の条件式で && を & あるいは || を | と書き間違えると、この点で動作が異なります。

ソースコード 4.3 if の入れ子と論理積の短絡評価

```c
#include <stdio.h>

void print_multi_nest(int m, int n) { // (A) if の入れ子
    if (n != 0) {
        if (m % n == 0) {
            printf("%d は %d の倍数です\n", m, n);
        }
    } else { // n == 0 の場合
        if (m == 0) { // 0の倍数は0だけ
            printf("%d は %d の倍数です\n", m, n);
        }
    }
}

void print_multi_and(int m, int n) { // (B) 論理積
    if (n != 0 && m % n == 0) {
        printf("%d は %d の倍数です\n", m, n);
    }
    if (n == 0 && m == 0) { // 0の倍数は0だけ
        printf("%d は %d の倍数です\n", m, n);
    }
}

int main(void) {
    print_multi_nest(10, 5); print_multi_and(10, 5);
    print_multi_nest(2, 3);  print_multi_and(2, 3);
    print_multi_nest(7, 0);  print_multi_and(7, 0);
    print_multi_nest(0, 0);  print_multi_and(0, 0);
    return 0;
}
```

コラム： ブロックのスタイル

ブロックのスタイルには、いくつもの流儀があります。自動的に整形してくれるツールもあるので、それほどこだわる必要もないのですが、意外なところに落とし穴があることを指摘しておきます。

本書では、下の abs1() の例のように、ブロック開始の { **を行末**に配置するスタイルを採用しています。書籍の K&R[?][?] や、Java 公式のコーディング規約もこのタイプです。行数が節約できるので、大きさの限られたディスプレイ（あるいは紙面）で読むのに好都合です。ただし、ブロックの開始・終了の対応を探すのに少し慣れが必要なので、熟練者向けと思う人もいるでしょう。

abs2() のように、{ **を行頭**に配置するスタイルもあります。ブロックの開始・終了の対応がわかりやすいので、初学者向けとも思えるのですが、授業を長年サポートしてきた経験からいえば、このほうが**致命的な問題**を引き起こします。

```
int abs1(int x) {   // {が行末
    if (x < 0) {
        return -x;
    } else {
        return x;
    }
}
```

```
int abs2(int x)     // {が行頭
{
    if (x < 0)
    {
        return -x;
    }
    else
    {
        return x;
    }
}
```

右下の abs3() は、abs1(), abs2() と同じく、絶対値を求めようとしているのですが、失敗しています。初学者はもちろん、指導役の**熟練者までもが間違いを見抜けず**、困り果てる姿を何度も見てきました。そしていつも、行末スタイルに慣れてもらえばよいのに、と思います。

答えは行末スタイルに書き換えれば、すぐにわかるでしょう。

```
int abs3(int x) // ×絶対値失敗
{
    if (x < 0);
    {
        x = -x;
    }
    return x;
}
```

4.7 練習問題

指示された関数以外にも、main() を作って動作を確かめよ。

1. **[1 要素による分岐（☞4.1 節）]**
   ある美術館の入館料は 1 人 1000 円であるが、20 人以上の団体の場合は、10% 割り引かれて 1 人 900 円になる。次の関数を作れ。
   - `int total_charge(int n)` は、n 人の団体の入館料の総額を返す。

   ヒント：複数の if を用いると、return に到達しないことがある。（☞4.4.2 項）

2. **[2 要素による分岐（☞4.2 節）]**
   あるお化け屋敷は、6 才以上の子供なら 1 人で入場できるが、そうでなければ 1 人以上の大人の付き添いが必要である。いま、a 才の子供に n 人の大人が付き添っている（$a \geq 0, n \geq 0$）。入場できるかどうかを、if と else と論理和（または論理積）を各 1 回ずつ用いて判定して表示せよ。論理積と論理和は、どちらか片方だけ用いてよい。

n人＼aオ	0 ～ 5	6 ～
0	×	?
1 ～	?	○

   ヒント：「?」を埋めよ。

3. **[多重分岐（☞4.1.1 項、4.2 節）]**
   点 (x, y) を、「原点」「X 軸上」「Y 軸上」「軸上ではない」の 4 通りに分類して表示せよ。if と else を各 3 回ずつ用いよ。「X 軸上」「Y 軸上」には、原点を含めないものとする。論理積を使わない方法と、if を入れ子にしない方法の、2 通り作れるとよい。

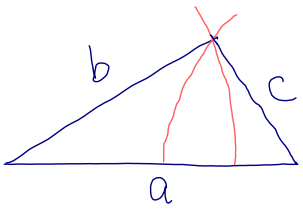
y ＼ x	x == 0	x != 0
y == 0	原点	?
y != 0	y軸上	?

   ヒント：「?」を埋めよ。

4. **[多重分岐（☞4.1.1 項、4.4.4 項）]**
   4.4.4 項の **(C) 多重分岐**の例では、else if を繰り返して、高学年から判定しているが、低学年から判定するよう**判定順序と条件**を変更し、次の関数に独立させよ。
   - `void print_position(int a)` は、年度初めの年齢 a 才に応じて「小学生低学年」「小学生高学年」「中学生」と表示し、これら以外は何も表示しない。

   main() で 5, 6, 8, 9, 11, 12, 14, 15 才の動作を確かめよ。（☞20 ページのコラム）

5. **[論理型の関数（☞4.5.2 項、ソースコード 4.2）]**
   3 辺の長さが a, b, c (> 0) の三角形は、「$a < b + c$ かつ $b < c + a$ かつ $c < a + b$」のときに存在し、そうでなければ存在しない。これを判定する、次の関数を **if 文を用いずに**作れ。簡単のため $a, b, c > 0$ を前提とする。



   - `int exist_triangle(double a, double b, double c)` は、3 辺の長さが a, b, c の三角形が存在すれば論理型の TRUE、そうでなければ FALSE を返す。

6. **[0 以外]**

x 円を何枚かの硬貨で払いたい。次の関数を作り、main() から繰り返し呼び出すことで、右の**実行結果**を実現せよ。

- `void print_coins(int x)` は、合計が x 円になる硬貨の、各枚数を表示する。
  - x は 10 以上 990 以下の 10 の倍数であることを前提とする（呼び出し側の条件）
  - 使用する硬貨は 500 円，100 円，50 円，10 円の 4 種類
  - 各硬貨の枚数は十分にある
  - 硬貨の合計枚数が最小となる払い方を選ぶ
  - 0 枚のものは表示しない

実行結果

```
40 円を支払います。
 10円硬貨 4 枚

50 円を支払います。
 50円硬貨 1 枚

90 円を支払います。
 50円硬貨 1 枚
 10円硬貨 4 枚

100 円を支払います。
 100円硬貨 1 枚

490 円を支払います。
 100円硬貨 4 枚
 50円硬貨 1 枚
 10円硬貨 4 枚

500 円を支払います。
 500円硬貨 1 枚
```

（ヒント）

- 高額の硬貨から枚数を決定すれば、自動的に合計枚数が最小になる。
- ある硬貨の枚数を決定したら、x からその金額を引く。残った x 円を、次の硬貨で支払うことにする。これを、4 通りの硬貨で繰り返せばよい。
- 硬貨の金額を変数にすると、4 回の処理が（金額を除いて）完全に同じになる。

コラム： 境界値分析

プログラムの間違いを探すための効率的な手法の一つに、**境界値分析** (boundary value analysis) というものがあります。ある条件の成立する（あるいは成立しない）境界近くのギリギリの数値を重点的に試すというものです。if で使われているであろう条件式のイコールの有り無しが正しいかをあぶり出すわけです。

例えば 4. の問題であれば、6 才、9 才、12 才、15 才と、その 1 才下の年齢を試すと効果的です。

6. の問題では、例えば 490 円と 500 円では、使用する硬貨がすっかり切り替わります。このような切り替わりの境界をしっかりと試しておきましょう。

索引