第 6 章

複雑な繰り返し処理

ここでは繰り返し処理のより高度な使い方を学びます。複数の繰り返しを入れ子にしたり、途中で終了したり、途中から次の繰り返しへ飛ばす制御が可能です。

世の中の問題は、数学のように美しく解けるものばかりではありません。愚直に総当たりで数えたり、条件に合うものを探し出したりする場面があるかもしれません。そのような問題に対して、繰り返し処理を使いこなせば、強力な武器となるでしょう。

キーワード

- 2 重ループ
- 無限ループ
- break・continue
- ある?ない?

6.1　2 重ループ

for や while のループ処理のブロックの中では、入れ子（ネスト）にして、さらに for や while のループ処理が行えます。

```
for (int i=1; i<=3; i++) {      // 外側のループ (1,2,3)
    for (int j=5; j<=7; j++) { // 内側のループ (5,6,7)
        printf("i=%d, j=%d\n", i, j);
    }
}
```

入れ子になった 2 つのループを **2 重ループ** (nested loops) といい、最初のループを「外側のループ」、ループの中のループを「内側のループ」などと呼びます。ループ変数は**ループ毎に別のもの**を用意する必要があります。

右の実行例を見ると、2 重ループの中で i と j の値の組合せ（ここでは 9 通り）がすべて網羅されることがわかります。そしてループ変数は**内側のほうが早く変化する**ことも理解しておきましょう。

```
i=1, j=5
i=1, j=6
i=1, j=7
i=2, j=5
i=2, j=6
i=2, j=7
i=3, j=5
i=3, j=6
i=3, j=7
```

もしも 2 重ループでループ変数を共有すると、ループの条件判定がおかしくなってしまいます。次の例を見てみましょう。

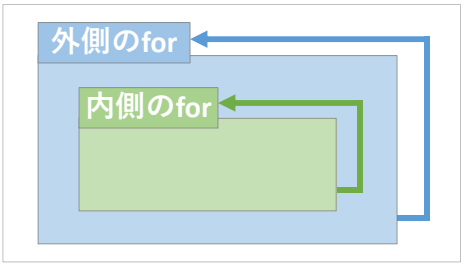
```
int i; // 共通のループ変数!?
for (i=1; i<=3; i++) {      // 外側のループ (1,2,3)
    for (i=5; i<=7; i++) { // 内側のループ (5,6,7)
        printf("i=%d, i=%d\n", i, i);
    }
}
```

右の実行例では、外側のループが実行されてないように見えます。外側のループ変数が、内側のループで上書きされています。

```
i=5, i=5
i=6, i=6
i=7, i=7
```

このように、内側のループでは新しいループ変数を用意する必要がありますが、重複しても**エラーにも警告にもなりません**。初めの例のような、for の初期化式に変数定義も含めるスタイルが安心[*1]です。

[*1] 安心の理由は、（残念ながら）変数名の重複を検査できるわけではなく、（驚いたことに）重複しても正しく 2 重ループの動作をすることにあります。重複してもよい理由は**??**項でわかります。



6.1.1 for のループ変数のスコープ

すでに活用していますが、C99 からの新しい機能で、for の初期化式で変数定義を行えるようになりました。この変数のスコープは、for ループのブロック終了まで[*2]です。

```
for (int i=0; i<n; i++) { // 初期化式でiを定義      iのスコープ
    ...;
}
// ここでは上のiは無効
for (int i=0; i<n; i++) { // iを繰り返し使える      iのスコープ
    ...;
}
```

このおかげで、別の for ループを続けて書くときに、同じループ変数名を繰り返して使えます。「スコープは狭く」の精神（☞**??**ページのコラム）にも合致しますし、C++ や Java といった多くの言語でも同様に実現されている機能なので、ぜひ活用しましょう。

コラム： 多重ループ

2 重ループと同じようにして、3 重, 4 重, . . . のような、**多重ループ** (multiple nested loops) も実現可能です。ただし動作がわかりにくくなるので、実際には 4 重, 5 重のような深いループはあまり使われません。もしそのような動作をさせたくなれば、見通しがよくなるよう、例えば内側の 2 重ループ部分を関数に分離して、3 重ループにあたるところでは関数を呼び出す、などの工夫を行って、表面的には深い多重ループにならないようにするとよいでしょう。

[*2] このスコープには別の考え方もあります。過去にはある有名 C++ コンパイラが、「その for ループの一つ外側のブロックまで通用する」としたことがありました。確かに 6.3.4 節のような場面でメリットはあったのですが、続けて書く for のループ変数名は異なるものを使う必要があります。このデメリットは大きく、後に撤回され、今では C99 と同様になっています。

6.2 ループを抜ける・次のループへ切り上げる

ループを制御する文が 2 つあります。

continue 文 ループ中の処理を途中で切り上げて、次のループにとりかかります。

break 文 ループを抜け出します。

どちらも for や while のいずれのループでも使えます。多重ループの場合は、まだ抜けていない一番内側のループに作用します。

```
for (int i=0; i<1000; i++) {  // 結果的に1000には意味がない...
    if (i <= 2) { continue; } // i=0,1,2 でループの先頭に戻る
    if (i == 5) { break; }    // i=5 でループを抜け出す
    printf("%d\n", i);        // ここに到達するのは i=3,4 のみ
}
```

continue も break も便利そうな機能に思えますが、ブロック中の実行の流れをジャンプさせるため、**動作がわかりにくくなる**という弊害もあります。そのため、多用すべきでないという考え方が主流です。使用は、例外的な事象に、**最小限に留める**のがよいでしょう。

6.2.1 無限ループ

ループ処理は、ちょっとしたミスで終了しなくなることがありますが、そうではなくて、わざと終了しないループ＝**無限ループ** (infinite loop) を作ってみましょう。for でも while でも実現できます。

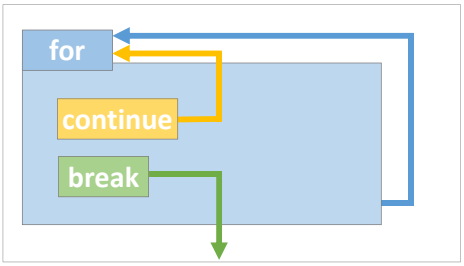
```
for (;;) {
    ...;
}
```

```
while (1) {
    ...;
}
```

```
while (TRUE) {
    ...;
}
```

for 条件式（カッコ内の 2 番目）は省略できるので、省略するだけで無限ループになるのですが、カッコ内のほかの式も不要になるので、通常は式をどれも省略して、**セミコロン 2 個だけ**を残します。

while 何かしらの条件式を書く必要があるので、**常に成立する比較式**として 0==0 を書くこともあります。もっとも、この比較式の演算結果は int 型の 1 になるので、最初から 1（あるいはマクロで定義した TRUE）と書くことが多いです。




```
int i = 0;
while (1) {
    if (i == 10) break;
    ...; // 10回ループ
    i++;
}
```

無限ループと言っても、break で中断できますから、左のようにすると 10 回ループを実現できます。ただし、メリットは感じられません。

無限ループが役立つのは、**ループの条件判定が単純には書けず**、例えば変数へ代入してからようやく判断できる、というような場合です。典型例は、外部からの入力を受け取るときです。詳しくは 6.4.3 項で紹介します。

6.2.2 continue の活用例

ここではインデントの**深さを節約**する continue 文の使い方を紹介しましょう。ループ処理の範囲は、視覚的にとらえられるようにブロック全体をインデントしますが、そのループ処理の中に条件文などが入れ子になってくると、どんどんとインデントが深くなって読みにくくなります。

例えば左下の例のように、**条件 A** が成り立ったときに**処理 X** を行い、それ以外に行うべき処理がないとしましょう。**処理 X** が長く、この中でも分岐があったりするとますます読みにくくなります。

そこで右下のように、**条件 A** が成り立たなかった場合に continue して、それ以外の場合に**処理 X** を行う、と書き換えてみましょう。else 節を書く必要がないので、**処理 X** のインデントを 1 つ浅くできます。

元々の if の後に処理すべきものがまったくないことが大前提ですが、このように continue で処理を飛ばしたい場面は、現実によく現れます。

```
while (...) {
    if (条件A) {
        処理X; // インデント深い
    }
    // ここに処理がないとする
}
```

```
while (...) {
    if (!(条件A)) continue;
    処理X; // インデント浅い


}
```

6.3　繰り返し処理にまつわる話題

TODO: 0 回ループ

6.3.1　ループ処理が空

ソースコード**??**を for で実現しなおしてみます。ループ内での処理がなくなってしまいましたが、ブロックをあらわす { } （あるいはセミコロン）は残す必要があります。そして、書き忘れではないことを示すのにコメントを書いておきましょう。

ソースコード 6.1　x を越える最小の自乗数（for 版）

```
1 int min_square(int x) {
2     int k; // スコープを広げるためにここで定義する必要がある
3     for (k=1; k*k<=x; k++) {
4         // 何もしない
5     }
6     return k*k;
7 }
```

for ループの後にもループ変数 k を使いたいので、k のスコープを広げるために、for ループの前で定義しています。

6.3.2　少なくとも 1 回は実行するループ

for も while も、ループの条件式を最初から満たさなければ、ループの処理部分をまったく行わないことになります。しかし状況によっては、少なくとも 1 回はループ処理を実行して、**ループを続けるかどうかは後から考えたい**、ということもあります。その場合に適しているのが do-while 文です。ループを続けるかどうかを、ループ処理の最後で判断します。

```
do {
    処理;
} while (条件式);
```

現実のプログラムに現れるループの内訳でいえば、for が圧倒的に多数、while がその次、do-while はたまに目にする、といった具合です。（本書でも 1 度だけ、6.4.3 項で利用します。）でも「少なくとも 1 回は実行する」という場面では do-while のことを思い出してあげてください。

鋭意作成中

6.3.3 逆順ループ

10, 9, 8, . . . , 1 と、ループ変数の値の減っていく、逆順ループも、for と while のどちらでも実現することはできます。

```
for (int i=10; i>0; i--) {
    printf("%d ", i);
}
```

```
int i = 10;
while (i > 0) {
    printf("%d ", i);
    i--;
}
```

しかし、条件に = をつけるかどうかがわかりにくいので、普段どおりのループにしておいて、表示する値を 10-i のように加工するほうがわかりやすいことも多々あります。

6.3.4 break したかどうかを後から判定する

for ループを、break で終了したのか、それともループを最後まで回りきったのか、ループの直後に知りたい場合があります。もちろん論理型変数を作って記憶させればわかることですから、多用すべき手法ではありませんが、ループ変数の動きをよく理解していれば、ループ後の値を検査することでも区別がつきます。ただし、変数定義を for のカッコ内で行うと、ループのブロックの外でその変数が使えないことに注意してください。

```
int i; // スコープを広げるためにここで定義する必要がある
for (i=0; i<10; i++) {
    if (...) break;
}
if (i < 10) { printf("breakしました\n"); }  // forの条件と同じ
```

この例では、ループを回りきったのならループ条件の i<10 を満たさなくなっているので、i は 10 になっているでしょう。逆に break していればループ条件をまだ満たしています。つまり、もう一度**ループ条件と同じ検査**をすると break したとわかります。

MEMO: 消してもよい

コラム：終わる（と思われている）繰り返し

コンピュータができて 70 年以上、計算理論という分野が出てきて 100 年以上が経っていますが、不思議なことに未だに未解決の問題がたくさんあります。

右の関数は、引数として与えられた正の整数から始めて、**偶数なら半分に、奇数なら 3 倍して 1 を加える**、という計算を繰り返します。1 になると、1 → 4 → 2 → 1 を繰り返すので、1 で終了とします。

さて、この簡単に書けてしまう関数はどんな正の整数に対しても終了するのでしょうか？これはコラッツ予想[a]と呼ばれる問題で、実は未だに決着がついていません。

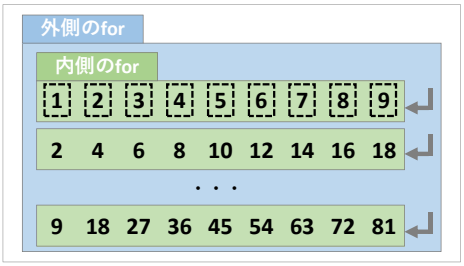
ソースコード 6.2　コラッツ予想

```
1  void print_collaz(int n) {
2      while (n > 1) {
3          printf("%d -> ", n);
4          if (n % 2 == 0) {
5              n = n / 2;
6          } else {
7              n = 3 * n + 1;
8          }
9      }
10     printf("1 -> ...\n");
11 }
```

様々な値で試した結果としては、万、億、兆の範囲ではすべて終了することが確認されています。予想としてはどんな値でも終了すると考えられていますが、証明はされていませんし、反例も見つかっていません。単純に書かれたものは、簡単に見えてもそうとは限らないという一つの例でしょう。

折角なので、この関数を for 文で次々呼び出して 50 くらいまでの結果を見てみるのも良いでしょう。10000 くらいまで試すと表示が多すぎて見えないと思いますが、まだまだ終了することが分かります。値によっては大きくなった結果、オーバーフローする場合がありますので、注意しましょう。例えば int 型が 32 ビットの場合でも 113383 から始まる系列が表示できないようです。開始時点の数値からするとずいぶん大きくなるようですね。これもどのような動きをしているか出力してみると面白いでしょう。

[a] 他にも様々な名前があります。日本人の名前を使う場合もあり、角谷（かくたに）の問題、米田の予想、とも呼びます。



6.4 より高度なループ処理の実例

6.4.1 九九の表

次のような九九の表を表示してみましょう。2 重ループで実現できそうです。

ソースコード 6.3 の実行結果

```
 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
....
 9 18 27 36 45 54 63 72 81
```

ソースコード 6.3 九九の表

```
1  #include <stdio.h>
2
3  int main(void) {
4      for (int i=1; i<=9; i++) {      // 縦に9行
5          for (int j=1; j<=9; j++) { // 横に9列
6              printf("%2d ", i*j);   // 積の表示（81回）
7          }
8          printf("\n");      // 改行して左端に戻る（9回）
9      }
10     return 0;
11 }
```

1 回　9 回　81 回

ここで注意したいのが、ソースコード 6.3 の 8 行目の改行の動作です。printf で表示した文字は、通常の文字なら左から右に進んでつながっていきます。**改行文字**（"\n"）を表示すると、表示位置が次の行（つまり下方）に進んで左端に戻ります。画面には何も表示されず、**表示位置のみが変化**します。これを 1 行の表示の最後に行っています。

この 8 行目の printf は、内側のループには含まれず、外側のループに入っているので、**9 回だけ**実行されることを理解しておきましょう。内側のループは、ループ自体が 9 回実行されます。6 行目の printf は内側のループに入っているので、**のべ 81 回**実行されます。

6.4.2　コンマで区切って列挙

1 から n までの数字を「1, 2, 3, . . . , n」のように、コンマで区切って表示してみましょう。ちょっと考えてみてください、ループで簡単に実現できるでしょうか。数字は n 個、コンマは (n-1) 個と、**個数が一致しない**ので、何かしら例外処理が必要になります。

(A) 先頭の 1 を無条件に表示して、残りを",%d"で表示する。

(B) (n-1) までを"%d,"で表示して、最後の n を無条件に表示する。

(C) 数値とコンマを常に別々に表示する。数値を"%d"で表示してから、最後以外で","を表示する、を繰り返す。

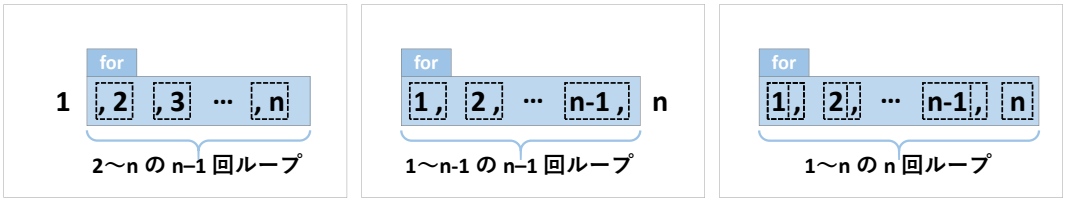
ソースコード 6.4　1 から n までの数をコンマ区切りで表示

```
1  void join1(int n) {  // (A)
2      printf("%d", 1); // 先頭を特別扱い
3      for (int i=2; i<=n; i++) {
4          printf(",%d", i);
5      }
6      printf("\n");
7  }
8
9  void join2(int n) {  // (B)
10     for (int i=1; i<=n-1; i++) {
11         printf("%d,", i);
12     }
13     printf("%d\n", n); // 末尾を特別扱い
14 }
15
16 void join3(int n) {  // (C)
17     for (int i=1; i<=n; i++) {
18         printf("%d", i);
19         if (i != n) { printf(","); } // ループの最後以外で,を表示
20     }
21     printf("\n");
22 }
```

このいずれの方法でも、ソースコード 6.4 の join1(), join2(), join3() のように実現可能です。しかしここで注意したいのは、例外的に行っている動作が、極端な条件下でどうなるか、です。

実は n の値の範囲をまだ明確にはしてませんが、例えば n=0 にしたときの動作はどうなればよいでしょうか。「想定外なので、どんな動きをしても知りません」という作り手の立場もあるでしょう。使い手の立場で言えば「0 個の数字が表示される＝数字は表示されない」と思ってるかもしれません。

実際、join1(0) と join2(0) は数字が表示されます。表示されないように修正するのを練習問題としておきます。



話は変わりますが、ソースコード 6.4 の 10 行目は**良いループ**（☞**??**節）でしょうか。

- `for (int i=1; i<=n-1; i++)` は**良いループ**です。1 から始まる (n-1) 回ループです。i=n に例外がありそうなことが、この表記からも読み取れます。
- `for (int i=1; i<n; i++)` でも動作は同じですが、ここには**不適切**です。0 オリジンで i=0 に例外がありそうにも見えます。

6.4.3 キーボードから正しい値を受け取るまで繰り返す

キーボードから入力された数値が、想定している範囲に入ってなかったとします。正しい値を入力してもらうよう、再入力を促しましょう。

??項で紹介するような手段で、`input_int()` がキーボードから int の数値を受け取るものとします。

少なくとも 1 回はキーボードから値を受け取りますので、do-while（☞6.3.2 項）の出番です。

```
int a;
do {
    printf("0以上10未満の数を入力してください => ");
    a = input_int();
} while (!(0 <= a && a < 10));
```

確かに、以下の実行例のように、範囲外の値（「100」と「-1」）を入力すると、また入力の場面に戻るのですが、利用者からすると、同じメッセージが表示されるだけなので、再入力を促されてるのかどうかわかりにくいです。

実行例：「100」と「-1」を入力した場合

```
0以上10未満の数を入力してください => 100
0以上10未満の数を入力してください => -1
0以上10未満の数を入力してください =>
```

そこで、入力された数値が範囲外であれば、そのことを表示してみます。

```
int a;
do {
    printf("0以上10未満の数を入力してください => ");
    a = input_int();
    if (a < 0) {
        printf("%dは0未満です。再入力してください。\n", a);
    }
    if (a >= 10) {
        printf("%dは10以上です。再入力してください。\n", a);
    }
} while (!(0 <= a && a < 10));
```

実行例：「-1」と「100」を入力した場合

```
0以上10未満の数を入力してください => -1
-1は0未満です。再入力してください。
0以上10未満の数を入力してください => 100
100は10以上です。再入力してください。
0以上10未満の数を入力してください =>
```

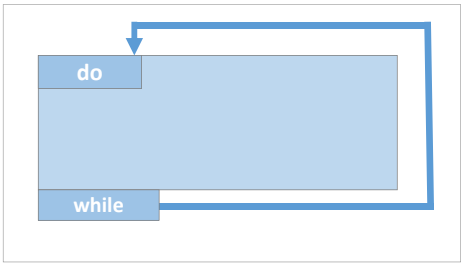
親切になったのですが、このプログラムには欠点があります。上限値の 10 を変更したくなったら、注意深く 2 ヵ所の条件式（に加えてメッセージの内容）を書き直す必要があります。つまり**メッセージを表示する条件**と、**ループを継続する条件**が別々になっています。これは崖っぷちでバランスをとってるようなもので、バグの温床になります。

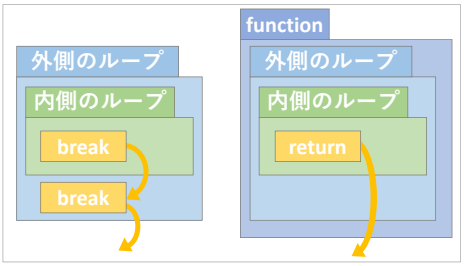
それでは、メッセージ表示とループ継続が、必ず**セットになる条件分岐**にしましょう。こうなると、もう do-while は使えず、無限ループの出番です。以下のようにすると、どちらの if が成立しても、メッセージを表示した上で break しません。

```
int a;
for (;;) {
    printf("0以上10未満の数を入力してください => ");
    a = input_int();
    if (a < 0) {
        printf("%dは0未満です。再入力してください。\n", a);
    } else if (a >= 10) {
        printf("%dは10以上です。再入力してください。\n", a);
    } else {
        break; // 正しく入力された
    }
}
```

break はエラーなどの**例外的**なものに対して使うことが多いので、正しい入力のときに break で中断するのは気持ちが悪いのですが、ここではいたしかたありません。

このように、人間の不規則な入力に対処するためのエラー処理は複雑になりがちです。人間に親切にすればするほど、長いプログラムになっていきます。





6.4.4 2 重ループの中断

break によるループの中断は、直近のループにだけ作用します[*3]。そのため、2 重ループを抜け出すには工夫が必要です。

まずは**論理型の変数**（フラグ）を使う方法です。下の例では、内側のループを break で抜ける前に、変数 finished を真にします。抜けた直後に finished を検査して、真であればもう一度 break します。これで外側のループから抜け出せます。

```
int finished = FALSE;
for (int i=0; i<10; i++) {
    for (int j=0; j<10; j++) {
        if (...) {
            finished = TRUE;
            break; // 内側ループ用
        }
    }
    if (finished) break;  // 外側ループ用
}
```

もうひとつの方法は、2 重ループを関数に入れておいて、**return で関数ごと抜け出す**というものです。

```
void func(void) {
    for (int i=0; i<10; i++) {
        for (int j=0; j<10; j++) {
            if (...) return; // ループはもちろん、関数ごと抜け出す
        }
    }
}
```

関数に分離すればインデントを浅くとどめられるという効果もあるので、実用的な手法だと思います。

[*3] 言語によっては、どのループまで抜け出すかを指定できるものもあります。

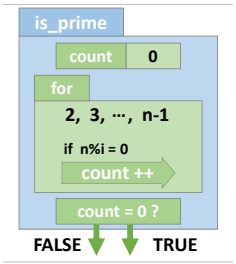
6.4.5　素数判定

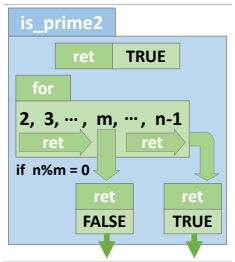
与えられた 2 以上の整数 n が**素数** (prime number) かどうかを判定してみましょう。数学の定義によると、1 と n 自身を除いて**約数のないものが素数**です。これから作る関数は、素数なら論理型の TRUE、そうでなければ FALSE を返すことにします。例外処理を省くために、与えられる n は 2 以上であることを前提にします。

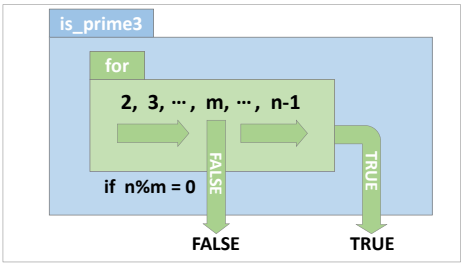
ソースコード 6.5 を見ていきましょう。3 つの関数がありますが、**下線部は共通**なので、下線部以外の違いをよく理解してください。

ソースコード 6.5　素数判定

```
1  #define FALSE 0
2  #define TRUE  1
3
4  /* n(>=2)が素数なら TRUE を返す */
5  int is_prime(int n) {              // (A)
6      int count = 0;                 // 約数の数
7      for (int i=2; i<=n-1; i++) {   // 1とnを除いてループする
8          if (n % i == 0) {
9              count++;               // 約数が見つかったら1増やす
10         }
11     }
12     if (count == 0) return TRUE; // 約数が0個なら素数
13     else            return FALSE;
14 }
15
16 int is_prime2(int n) {            // (B)
17     int ret = TRUE;                // 素数であることを前提にスタート
18     for (int i=2; i<=n-1; i++) {
19         if (n % i == 0) {
20             ret = FALSE;           // 約数が見つかったら素数ではない
21             break;                 // ループを続ける必要もない
22         }
23     }
24     return ret;
25 }
26
27 int is_prime3(int n) {            // (C)
28     for (int i=2; i<=n-1; i++) {
29         if (n % i == 0) {
30             return FALSE;          // 約数が見つかったら偽を返す
31         }
32     }
33     return TRUE;          // 最後まで約数が見つからなければ素数
34 }
```







コラム： ある?ない?

約数に限らず、「ない」と判定するのは、難しいことです。逆に「ある」と判断するのは簡単です。それを見つければ動かぬ証拠になります。

ループ処理中の if で、見つけたいものを探しているとします。if が成り立てば（見つかれば）「ある」と即断できます。逆に、ループ中に 1 回ぐらい if が成り立たなかったからといって、すぐに「ない」とは決められません。「ない」と判断できるのは、ループが終わってからです。そしてループ中では、いくつ見つかったのか、個数を数えておくとうまく判断できます。個数の 0 は「ない」ことを示します。

(A) まず is_prime() 関数では、定義どおりに 2 から $(n-1)$ までのループ処理で、約数の個数を数えてみます（7–11 行目）。その結果、**0 個であれば素数**だとわかるので TRUE を返します（12 行目）。そうでなければ FALSE を返します（13 行目）。

(B) しかしよく考えると、約数が 1 つでも見つかると、もっとたくさん見つかっても、素数でない（FALSE を返す）ことには変わりありません。つまり**約数の個数は重要ではない**ので、見つかったかどうかを論理型で覚えておけば十分です。そこで is_prime2() 関数では、変数 ret を用意して、ひとまず TRUE にします。（17 行目）。約数が見つかると ret を FALSE にします（20 行目）。こうすることで、ret 変数が関数の戻り値としてそのまま使えます（24 行目）。

ところで、ループ処理を続けて約数をいくつも見つけると、何度も ret に FALSE を代入することになりますが、**FALSE は一度代入すれば十分**ですので、そこでループを中断しても結果は同じです（21 行目）。これで計算時間を節約できます。

(C) ここまで、ループ処理中の計算結果を変数に反映してきましたが、関数の中での完結した処理なので、変数を使わなくても呼び出し元に影響はありません。そこで is_prime3() 関数では、素数ではないという結果が出れば、その場で FALSE を返して（30 行目）、ループはもちろん関数ごと抜け出すことにします。**結果を保存する変数が不要**になり、見通しよくなります。ループの最後に到達すると、素数だと判明するので TRUE を返します（33 行目）。

6.5 練習問題

1. **[3 重ループ or 1 重ループ + 桁分解（☞6.1 節または??章の練習問題??.）]**
   153 は特別な数である。各桁の数の 3 乗の和 $1^3 + 5^3 + 3^3 = 1 + 125 + 27 = 153$ が、元の数に一致する。このような 3 桁の整数を、総当たりですべて求めて表示せよ。
2. **[逆ループ or 計算（☞6.3.3 項）]**
   9 ページのソースコード 6.3 の九九の表を、上下反転して表示せよ。
3. **[2 重ループ、回数の変化するループ（☞6.4.1 項）]**
   2 重ループを用いて「*」の文字を並べて、縦横 n 文字の長方形を表示せよ。また同様に、縦横 n 文字の直角三角形を表示せよ。右は $n = 5$ の出力例である。

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

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

4. **[例外処理]**
   10 ページのソースコード 6.4 の join3(0) では数字が表示されない。しかし join1(0) と join2(0) では、1 個の数字が表示される。数字が表示されないように修正せよ。
5. **[複雑なループ]**
   ??ページのソースコード**??**の cbrt_eps() におけるループ回数に上限を設けよ。 (案 1) break を使用する (案 2) while の条件を厳しくする (案 3) for に置き換える
6. **[ある?ない?]**
   14 ページのソースコード 6.5 の is_prime3() 関数を利用して、次の関数を作れ。
   - `int is_sum_of_2primes(int n)` は、n が 2 つの素数の和で表せれば論理型の TRUE、そうでなければ FALSE を返す。（例：6 は 3+3 と表せるので TRUE）

   そして main() で、2 から 100 までの整数のうち、素数でもなく、2 つの素数の和でも表せないものを、すべて求めて表示せよ。（27 から 95 までに、9 個見つかる。）
   ヒント：n が素数でないことは「`if (!is_prime3(n))`」と検査する。（☞**??**項）
7. **[数列の和]**
   以下の数列 $\{a_k\}$ を、a_k と a_{k-1} の関係に着目して $a_0, a_1, \ldots, a_9$ を求め、表示せよ。

$$a_k = \frac{1}{k!} \qquad (k = 0, 1, 2, \ldots)$$

   そして、数列 $\{a_k\}$ の和 S_n を以下で定義する。$S_0, S_1, \ldots, S_9$ も同時に表示せよ。

$$S_n = \sum_{k=0}^{n} a_k = \sum_{k=0}^{n} \frac{1}{k!} = \frac{1}{0!} + \frac{1}{1!} + \frac{1}{2!} + \ldots + \frac{1}{n!} \qquad (n = 0, 1, 2, \ldots)$$

   - $0! = 1$ である。計算方法によっては $k = 0$ に例外処理が現れる。
   - S_∞ はネイピア数 e (= 2.71828$\cdots$) に収束すると知られている。
   - 書式文字列には`"%.10f"`などを指定して、表示桁数を増やすとよい。

8. **[キー入力に対するループ]** ☞**??**項

索引