4.2 素数の表示

プログラム [primechk.cpp] では、入力された数が素数かどうか判定するものだった。それなら初めから素数の一覧表があると便利だろう。そこで素数の一覧を表示するプログラムを用意した。今度のプログラムでは、随所に簡潔だが分かりづらい—しかし、慣れれば分かりやすい—書き方をしている。簡潔なほど単純で分かりやすいというのは、プログラムを組む場合には当てはまらない。冗長で複雑そうに見えるほうが分かりやすい場合が多い。記述の仕方は、ある程度好みの問題を含むので一概に言えない。私はここまでに、やや冗長な書き方をしてきたが、簡潔な記述を好む人もいるだろう。旅においても、目に付いた所を悠然と訪ねる旅程もあれば、名所を効率的に訪ねる旅程もある。目に付く場所を悠然と見ていれば、そこがどんな地か分かりやすい。冗長な記述はそういうものだ。一方、名所を効率的に訪ねるには、ある程度の知識が必要になる。簡潔な記述がそれにあたる。

プログラム [primenum.cpp] では、所々簡潔な表現を用いた。何もそうする必要はなかったのだが、こんな書き方も可能だということを示しただけである。

programming list [primenum.cpp]

```
#include <iostream>
#include <math.h>
#define SUP 100

int main() {
    int n, i, flag;
    std::cout << "2 ";
    for(n = 3; n < SUP; n += 2) {
        i = 1; flag = 1;
        while((i += 2) <= sqrt(n)) {
            if(flag) {
                flag = n % i;
            } else {
                break;
            }
        }
        if(flag)
            std::cout << n << " ";
            ;
    }
    std::cout << std::endl;

    return 0;
}
```

プログラムを解説する前に、素数の調べ方について話しておこう。例えば 49 が素数かどうか判定するには、2 から 48 までの数で割ってみればよい。何ひとつ割ることができなければ 49 は素数

だ。しかし考えてみてほしい。素数は 2 を除いてすべて奇数である。奇数は偶数で割れっこない。従って偶数での割り算を試す必要はないのだ。その結果、49 が素数かどうか調べるには、3 から 47 までの奇数で割ってみるだけでよい。これで計算量を半分に減らせる。

さらに、n が素数がどうか調べるのに、$n-1$ までの数で割る必要などない。結論を言えば $\sqrt{n}$ までの数で割るだけで十分である。例えば 51 は 3 で割れる—商は 17 である—から、同時に 17 でも割れてしまう。数 n が p で割れるなら、同時に n/p でも割れているのだ。

EX. このことから n が素数かどうか調べるのに、$\sqrt{n}$ までの数で割ればよいことが結論できる。それを確認せよ。

以上の考察によって、プログラムは次の考えに基づいて組み立てられる。

a) 奇数 n だけを対象に素数かどうかを調べればよい。

b) 候補となる n を割る数は、3 から $\sqrt{n}$ までの奇数で十分である。

それでは [primenum.cpp] を見ていこう。

早速 2:行目に `#include <math.h>` という見慣れぬものが現れたね。`#include <iostream>` とは何が違うのだろう。まず、これらは**ヘッダファイル**と呼ばれるファイルである。**C++**は基本的に規模が小さいプログラム言語で、余分な機能を極力装備しないようにできている。機能が必要なら、必要だと思う人があらかじめ機能を取り込めばよいのだ。普通のプログラムは、キーボードから入力したり画面に出力する機会が多いだろう。そのような**標準入出力**の機能は `<iostream>` —これは input/output stream の略だ—によって取り込まれる仕組みなのだ。だからプログラムの初めに `#include <iostream>` が必要だったのだ。

では、`<math.h>` が何だか想像できるね。そう、これは数学的処理を必要とするときに装備される。平方根の計算は数学的処理にあたる。そこで `#include <math.h>` によって、数学処理機能を装備したのである。

3:行目で調べる数の上限を 100 とした。もっと大きな素数まで求めたければ、ここの値を変えるだけでよい。

プログラムは `main` 関数しかない。6:行目で必要な変数を用意したが、`n` には候補となる数が代入され、`i` は `n` を割り算で試す除数である。変数 `flag` は今までの変数と違う使われ方をする。候補となる数が素数でなければ `flag` に `0` が代入され、素数であれば `flag` に正の数が代入される。素数を画面に出力するかどうかは、`flag` の値で判断するのである。

7:行目は画面に “2 ” を表示する文だ。それは 2 が最初の素数だからである。あれ？　コンピュータに計算させるんじゃないの？と感じたかな。何でもかんでもコンピュータに計算させようとする

のは間違っている。自明なことは計算させることなどないのだ。もし君が、素数を 2, 3, 5, 7 まで確実に知ってると言うなら、7:行目は `std::cout << "2 3 5 7 ";` として構わない。そして 8:行目では、`n = 9` から調べればよいのだから。

このプログラムは正直に `n = 3` から調べている。それは 8:行目の `for` 文を見れば分かる。`for()` 内の最後が `n++`でないことに注意してほしい。調査は奇数に対して行われるので、2 ずつ増えるよう `n += 2` と書いている。そして、この `for` 文は 20:行目の `}` で終わっている。

9:行目で `i = 1` としたのは、次の行で `i` を 2 つ増やして、3 から割り始めるようにするためだ。また、`flag = 1` としたのは、とりあえず候補の数は素数であると仮定するためだ。素数でないことが判明し次第、`flag` には 0 が代入される。

さあ、ここからが厄介だ。10:行目は `while` 文だが、この終わりは 16:行目の `}` である。従って、`while()` の条件が正しければ`{}`内の処理が行われ、条件が正しくなければ 17:行目に処理が飛ぶのだ。

具体的には、`while` 文に入る直前は `n = 3, i = 1` である。すると条件 `(i += 2) <= sqrt(n)` は、$3 \leq \sqrt{3}$ が正しいかどうかを問うていることになる。$3 \leq \sqrt{3}$ は正しくない、すなわち偽だ。よって `while` 文は`{}`内の処理をしないで 17:行目に移る。

17:行目は `if` 文である。`if` 文も `while` 文同様、`()` 内の条件によって処理が分岐する。しかし `if(flag)`って何だ？　`flag` では条件式になっていないじゃないか。その通り。`flag` だけでは式とは言いづらい。でも、これも立派な式なのだ。実は、`()` 内に書かれる条件には 2 種類ある。ひとつは式による判定である。これは式が正しいか正しくないかで判定する。そしてもうひとつが式の値による判定だ。**C++**では一般に、式の値が 0 のときが偽、0 以外のときが**真**—すなわち正しい—と判定される。私たちが 0 を自然数の仲間に入れなかったことを覚えているだろうか。0 は不自然数というわけだ。その感覚を継承すれば、0 だけが偽の扱いになるのもうなづけるだろう。

そうであれば `if(flag)` は十分判定の材料を与えている。処理が 17:行目に来たとき、`flag` の値は 1 だったはずだ。だからこの `if` 文は真の判定をし、そのための処理をするわけだ。

ところで `if` 文の下には、18:行目に `std::cout` が、19:行目に...ん？　`;` だけがある。ここには `else` 文がないことに注意してほしい。

`if(A){X}else{Y}`のように `else` が使われている `if` 文であれば、`A` が真のときには`{X}`の処理がされ、`A` が偽のときには`{Y}`の処理がされたことと思う。つまり二者択一でどちらか一方の処理だけがされる。だが、`if(A) X Y` では注意が必要だ。この場合は、`A` が真のときには `X` から処理が始まり `Y` 以下が処理される。`A` が偽のときには `X` は飛ばされ `Y` から処理が始まるのだ。

すると今はどうなるんだろう。17:行目の `if(flag)` は `flag(1)` であるから真の判定が下される。真であれば 18:–19:行目に処理が進むはずだ。18:行目は `n` の値を出力するので画面には “3” が表示

され、さらに 19:行目も処理される。

ところで 19:行目は ; しかない。これはいささか矛盾を含むが**空文**と呼ばれ、何もしない文である。では、何のためにあるのか？　それは `if(flag)` が偽になったときのためである。`if(flag)` が偽のときは素数でないのだから、何もせず次の数へ進む必要があるのだ。

さて、プログラムは 17:行目以降に飛んでしまったが、10:行目からの `while` 文はいつ実行されるのだろうか？　それは `(i += 2) <= sqrt(n)` が真になるときだから、`n = 9` になって初めて実行される。

このとき 11:行目で `if(flag)` の判定があるが、この時点での `flag` は `1` である。従って真の場合の処理、12:行目が実施される。

12:行目の処理はちょっとうまい方法を用いている。それは `flag` に `n / i` の余りを代入していることである。`n` が素数でなければ余りは `0` だから、`flag` が `0` になるのだ。素数の可能性があれば必ず `0` 以外の余りが出るので、何度でも 12:行目が試されることになる。もし、何度試しても割り切れず、そうこうするうち `(i += 2) <= sqrt(n)` が偽になれば、`while` 文を抜けて 17:行目以降で素数と判定されるのである。

一方、12:行目で余りが `0` になれば、次の `if(flag)` の判定は偽になって、`else` 文へ処理が移る。

`else` 文は 14:行目の `break;` である。`break;` は "現在行われている`{}`内の処理" を中断し、`}` の直後へ出る命令だ。おっと、この言い方だと誤解を生じかねない。具体的に言おう。この場合、"現在行われている`{}`内の処理" というのは、`else{}`ではなく、`if(){}`全体—すなわち `while(){}` 内の処理—のことである。従ってここでは 16:行目の直後へ出るが、この場所はまだ `for` 文の中である。

しかも、`break` 文で 16:行目の直後へ出たときは、`flag` の値は `0` だから、17:–19:行目は無いに等しい。よって、`n` が `2` つ増えて、引き続き `for(){}`が繰り返される仕組みだ。

もし `break;` の出どころが不安なら、11:–15:行目は

programming list [change of line 11-15.cpp]

```
11:             if(!flag)
12:                 break;
13:                 flag = n % i;
14: //（空行）
15: //（空行）
```

とするとよい。これなら `while(){}`の外へ出ることが一目瞭然だ。記号 `!` は**否定演算子**と呼ばれ、真偽を逆にする効果がある。すなわち、`flag` が真なら`!flag` は偽となって `flag = n % i;` が実行され、`flag` が偽なら`!flag` は真となって `break;` が実行され `}` の直後に出る。理解できただろうか。正直に言って、`break;` の出どころはややこしい。

EX. 今のように `if` 文で`{}`を用いない場合、`if(flag)`、`flag = n % i;`、`break;` の順に書いたのではうまく動作しない。その理由を考えてみよ。