1. . . の旅

1.1 フィボナッチ数列

数列とは、数が何らかの規則で並んでいるものを言う。

$$2,\ 4,\ 6,\ 8,\ 10,\ \ldots$$

は 2 から始まる偶数の数列だが、この数列を作る規則は『前の**項**に 2 を加える』というものである。

さて、ここに 1, 1 から始まる数列がある。次の項を作る規則は “直前の 2 項の和” である。すなわち 1, 1 の次に来る項は 2 となり、数列は 1, 1, 2 と延びる。規則を繰り返し当てはめれば、次の項は 3 で数列は 1, 1, 2, 3 と延び、さらに次の項は 5 である。これが延々となされ

$$1,\ 1,\ 2,\ 3,\ 5,\ 8,\ 13,\ 21,\ 34,\ \ldots \tag{1.1}$$

なる数列が出来上がる。このようにしてできる数列を**フィボナッチ数列**[1]と呼ぶ。

フィボナッチ数列をコンピュータで再現してみよう。

programming list [fibonacc.cpp]

```
#include <iostream>
#define SUP 40

int main() {
    int i, f1 = 1, f0 = 1;
    for(i = 3; i <= SUP; i++) {
        std::cout << f1 + f0 << " ";
        f1 = f1 + f0;
        f0 = f1 - f0;
    }
    std::cout << std::endl;

    return 0;
}
```

ここでは `#include <iostream>` の他に `#define SUP 40` が目に付くだろう。この行は、`SUP` という定数値を 40 に定めることを意味する。すると以後 `SUP` が出るたびに、それは 40 に置き換

[1] フィボナッチ (1174?–1250?) : イタリア、ピサのレオナルドの通称。

わる。今は 40 項までの計算をさせたいため、あらかじめ SUP を 40 にしているわけだ。こうする利点は、後で 100 項まで計算させたくなったとき、この行の変更だけで済ますことができる。このような短いプログラムでは恩恵が分かりにくいが、あちこちに一定の値が使われるプログラムなら、#define で定義しておくとよいだろう。また注意したいのは、SUP に 40 が代入されるわけだが、= がないことと行末に ; がないことである。

では、プログラムの本体へ入ろう。まず 5:行目で、カウンタに使う変数 i の他に、フィボナッチ数列の計算に必要な 2 つの変数 f1, f0 を用意した。数列は 1, 1 の項から始まっているので、変数を用意すると同時に 1 を代入している。変数はどこかで初期値に当たる数を代入—すなわち**初期化**—する必要があるが、この例のように変数を用意したときに代入することもできる。もし、ここで初期化を怠ると、8:行目でなされる f1 + f0 の結果がいくつになるか保障できない。

TRY! 実際、f0, f1 を初期化しないで実行してみよ。とんでもない数列を目にするはずだ。

6:行目の for 文は i = 3 から始めて、SUP—今は 40 で定義されている—まで 1 ずつ増加させている。i = 3 から始めた訳は、f1 + f0 で与えられる項がすでに第 3 項になるからである。すると、for 文で回されるのはフィボナッチ数列の 3 項目から 40 項目の意味になる。もし i = 1 から始めれば、フィボナッチ数列を単に 1 個目から 40 個目まで計算する意味になるが、それはフィボナッチ数列の 3 項目から 42 項目である。

7:行目の std::cout は、整数値の後ろにひとつの空白を付けて画面に表示する。std::cout は for 文で繰り返されるので、フィボナッチ数は空白ひとつ分の間隔で表示されるのだ。その際、復帰改行をしないことに注意しよう。

8:–9:行目はかなりトリッキーなことをしている。フィボナッチ数は常に f1 + f0 で計算するので、ここの f1 + f0 の値が次は f1 となり、今まで f1 だった値が f0 にならなくてはならない。だったら、f1 = f1 + f0;、f0 = f1; でいいんじゃない？と思うだろう。残念だがそれは違う。

EX. なぜだめなのか、よーく考えてみよ。

ならば代入の順番を変えて、f0 = f1;、f1 = f1 + f0; としたらどうだろう。だが、これもさっきの理由と同じでだめなのだ。

そこでトリッキーな解決をしている。8:行目が終わったとき f1 には f1 + f0 が代入される。これは予定通りだ。次に f0 には “古い”f1 の値がほしいのだが、この時点での f1 は “新しい” ものになってしまった。しかし安心してほしい。“新しい”f1 は “古い”f1 を含んでいる。なぜなら f1 = f1 + f0 だからだ。f0 がなければ “古い”f1 を取り出せる。そこで 9:行目の式となる。見事に f0 に “古い”f1 の値が入っただろう。

また、11:行目で初めて `std::endl` を送って改行をした理由は、[`printout.cpp`] で経験したので分かると思う。

このプログラムでは、あまり先の項まで調べることは無理だが、ある程度のところまでのフィボナッチ数列を感じることができるはずだ。

TRY! `#define SUP 40` の値を変更して、もう少し先のフィボナッチ数を求めてみよ。

TRY! このままのプログラムでは、例えば 34 が何項目のフィボナッチ数か分かりづらい。そこで、"fibonacci[9] = 34" のような表示をするプログラムにせよ。

TRY! ここではトリッキーな代入をしているが、新しいフィボナッチ数を一時的に代入する変数 `temp` を用意すれば、もっと直観的にプログラムが組める。変数 `temp` を追加してプログラムを書き換えよ。