10 関数 (続き)

課題 6 (5/17 出題). 2 数の最大公約数を返す関数 gcd() を作成し、それを用いて、正整数 n に対し、それと互いに素な n 以下の正整数の個数 $\varphi(n)$ を求めるプログラム euler.c を作成せよ。

解答例 6.1. ここでは入力値は正整数と仮定している。

```
/*   euler.c   2010-05-17   */

#include <stdio.h>

int gcd(int, int);

int main(int argc, char** argv)
{
    int i, n, euler = 0;

    printf("Input a natural number: ");
    scanf("%d", &n);

    for( i=1; i<=n; i++ )
    {
        if( gcd(n,i) == 1 )
        {
            euler++;
        }
    }
    printf("phi(%d) = %d\n", n, euler);
}

int gcd( int m, int n )
{
    int r;

    while( n )
    {
        r = m % n;
        m = n;
        n = r;
    }
    return m;
}
```

if(gcd(n,i) == 1)
に注目。関数呼出 gcd(n,i) は返値の型を持つ式なので、このように他の式の中で使うことが出来る。gcd(n,i) の値を他にも使うなら何かの変数に代入して覚えておく必要があるが、ここでは 1 かどうかの判定をしたら用済みなので、直接比較してみた。

情報処理 I では減算の繰返しで書いたが、C 言語には剰余演算子 % があるので、これを使おう。

while(n)
は
while(n != 0)
と同じこと (非零が真なので)。しばしば用いられる書き方だが、解り難ければ
while(n != 0)
で一向に構わない。

考察 . 関数 gcd() の引数が $m < n$ の場合でも、これで大丈夫か。数学的な必要性は？ 計算上の得失は？

この解答例では、2 数の最大公約数を計算する部分を、関数 `gcd()` として部品化して利用している。このように関数として部品化する利点は沢山あるが、幾つかを挙げると、

- プログラムの各所で何度も現れる一連の手続きを一まとめにすることで、簡潔に書くことが出来る
- (一度しか現れないとしても) プログラム全体の構造が見易くなる。
- 部品化しようとすることで問題の分析が進む。
- 他のプログラムでの利用が容易になる。(「使い回しの心」)
- より効率的な方法が見付かったら、その関数だけ置き換えれば、プログラムの改良が実現できる。(多くのプログラムで利用していたら、それらが一挙に改良されることになる。)
- 作業を分担できる。(引数と返値とだけ決めておけば良い。)
- 再帰呼出を表現するのに必要。

などなど。

実習 10.1. 必要なら、自分の作成したプログラム内の関数 `gcd()` を、他の実装に置き換えてみよ。

考察 10.1.1. `main()` と `gcd()` とで変数名 `n` が重複しているが、大丈夫なのか。関数 `gcd()` 内で、引数として受け取った `m`, `n` の値を書き換えているが大丈夫なのか。

演習 2. 先の課題から、正整数 n に対し Euler の φ 関数の値 $\varphi(n)$ を返す関数 `euler()` を作成し、それを用いて 1 から適当な数までの $\varphi(n)$ の表を作成せよ。

11　関数 (2) 〜ライブラリ関数の使い方〜

11–1　コンパイルの詳細

cc コマンドによってソースファイルがコンパイルされて実行形式が生成されるが、コンパイルは以下の 4 つの工程に大きく分けられる。

(0) プリプロセス (preprocess): プリプロセッサ (後述) により、指定された各ソースファイル (〜.c) 内の、#include など # で始まる前処理命令などが処理される。
(1) コンパイル (compile): 前処理された各ソースファイルから、CPU 命令と一対一に対応したアセンブラコード (〜.s) が生成される。
(2) アセンブル (assemble): アセンブラコードから、CPU が解釈するオブジェクトファイル (〜.o) が生成される。
(3) リンク (link): 各オブジェクトファイルと標準関数ライブラリ libc.a・その他指定されたライブラリやオブジェクトファイルを繋げて、実行形式が生成される。

(1) の工程が狭い意味でのコンパイルである。(1), (2) の工程を併せてコンパイルを呼ぶこともある。

システムによっても異なるが、cc では -V オプションを、gcc では -v オプションを、それぞれ用いると、その過程をより詳細に表示させて見ることが出来る。

-E, -S, -c オプションで、それぞれ (0), (1), (2) の各段階で処理を止めることが出来る。-c オプションでのオブジェクトファイル (〜.o) の生成は、今後、実際に使うこともあるだろう。

11–2　関数ライブラリ

関数ライブラリとは、幾つもの関数のオブジェクト (関数のソースファイルからコンパイル・アセンブルまで行なったもの) を一つにまとめたものである。ライブラリに用意された関数を用いるには、適切なヘッダファイル (〜.h) を #include で読込み (ここまでで狭義のコンパイルは可能)、かつ、適切なライブラリから既にコンパイルされているオブジェクトを探してリンクする必要がある。

printf() などの標準関数は標準ライブラリ libc.a に含まれていて、これはコンパイル時に自動的にリンクされるので何も指定する必要はないが、他のライブラリで定義されている関数を使いたい場合には適切なライブラリをリンクする必要がある。

ヘッダファイル (header file):
関数プロトタイプなどを予めまとめて書いてあるファイル

標準関数でも適切なヘッダファイルの読込は必要。例えば printf() なら stdio.h を include する。

実習 11.1. 以下のように数学関数 sin() を用いて 10 度刻みで $\sin x$ の値を表示するプログラム sinx.c を作成せよ。

正弦関数 sin() などの多くの数学関数が標準的に用意されていて、プロトタイプはヘッダファイル math.h に、オブジェクトは数学関数ライブラリ libm.a にある。

#define については後述。

表示の右辺では printf 変換指定 %f のオプション ’ ’ (空白) を用いてみた。

```
/*   sinx.c   2010-05-24   */
#include <stdio.h>
#include <math.h>

#define PI 3.1416

int main(int argc, char** argv)
{
    int i;
    double x;

    for( i=-90; i<=90; i+=10 )
    {
        x = i * PI / 180;
        printf("sin(%6.3f) = % f\n", x, sin(x));
    }
}
```

実行例 . #include <math.h> としてヘッダファイルを読み込んでおけば、オブジェクトの生成までは出来るが……。

```
=> cc -o sinx sinx.c                    ← これでコンパイル・アセンブルまでは出来るが、
Undefined                       first referenced      ← ファイル sinx.o で参照された
 symbol                             in file              sin という名前の symbol が
sin                                 sinx.o                          解からない
ld: fatal: Symbol referencing errors. No output written to sinx
                                    ← と言って、リンカ ld がエラーメッセージを出した。
=>
```

実行例．実行形式を生成する為には、数学関数ライブラリ libm.a をリンクする必要がある。それにはコンパイル時にコマンドの最後にオプション -lm を付ける。

```
=> cc -o sinx sinx.c -lm     ← リンカオプション -lm を付ける。
=> ./sinx
sin(-1.571) = -1.000000
sin(-1.396) = -0.984808
    ...
sin( 1.571) =  1.000000
=>
```

11–3 オンラインマニュアル man の利用

ライブラリ関数を利用したいが関数名が判らない場合は、man のキーワード検索オプション -k を使うと良い (apropos コマンドでも同じ)。関数名が判ったら、その関数の引数や返値についての情報を man で調べる。

例．man sin の冒頭部分

```
Mathematical Library Functions                                  sin(3M)

NAME
     sin, sinf, sinl - sine function

SYNOPSIS
     c99 [ flag... ] file... -lm [ library... ]      ← ここで必要なライブラリを確認
     #include <math.h>                               ← ここで必要なヘッダファイルを確認

     double sin(double x);                           ← ここで関数の返値・引数の数・型を確認

     float sinf(float x);

     long double sinl(long double x);

DESCRIPTION
     These functions compute the sine of its argument x, measured
     in radians.
...
```

一般にライブラリは libxxx.a という形のファイル名を持つ。libxxx.a をリンクするには、リンカオプションを -lxxx とする。

このような表形式の出力をしたい場合には、printf 変換の様々なオプションが便利。

例: コマンド ls について調べる

```
man ls
```

man で表示されるオンラインマニュアルは、システムによって異なる。日本語のオンラインマニュアルが提供されている場合もあるが、一般に解り易いとは限らないようだ。

11–4　プリプロセッサ・#define 行について

で始まる行はプリプロセッサ (前処理系) への指示であり、狭義のコンパイルに先立ち、プリプロセッサで処理される。#include 行は、(ヘッダファイルと限らず) 指定されたファイルをその位置に読込む。

#define 行は次の書式を持ち、プリプロセッサはそのファイル内でそれ以降に現れる識別子を置換文字列で置換える。

引数を取るもっと複雑な書式も取れるが、ここでは割愛。

```
#define 識別子 置換文字列
```

C 言語の文ではないので、行末に ; は不要。書けば文字列の一部と見做される。

これを用いると、プログラム中で「意味の判る定数」を記号で書くことが出来る。例えば、

```
#define PI 3.1416
```

と宣言しておくと、プログラム中で 3.1416 という数値を直接書く (埋込む) 代わりに PI と書ける。

C のプログラムは殆どが小文字で書かれるので、関数名や変数名と区別し易くするため、#define で定義する識別子は大文字を用いるのが一般的。

わざわざ #define を用いて定数を書く利点は沢山あるが、幾つかを挙げると、

- 定数の意味が明確になってプログラムが読み易くなる。
- プログラム中で同じ定数を何度も用いるような場合、修正が冒頭の #define 行の一ヶ所だけで済む。
- 違う意味の定数が偶々同じ値を取った場合、生の値をソース中に埋込んでしまうと、その片方だけを変更する場合に非常な困難に遭うが、それぞれを別の名前で #define で記述しておけば、その区別が可能で、修正も容易である。

「出てこい出てこい、いけのこい」

などなど。

課題 7 (〆切 5/30(日)). 正の実数 a に対し Newton-Raphson 法で $\sqrt{a}$ を求める関数 root() を作成し、それを用いて $\sqrt{a}$ の値を求めると共に、標準で用意されている数学関数を用いて求められた値と比較するプログラム root2.c を作成せよ。(ここでは、Newton-Raphson 法における初期値は a、誤差範囲は $\varepsilon = 10^{-6} = 0.000001$ としてよい。)

単に「Newton 法」とも言う。解からない人はちゃんと調べるように!!

平方根を求める数学関数の名前が判らない場合の調べ方は前頁を参照。
(TEX に馴染みがあれば見当が付く?)