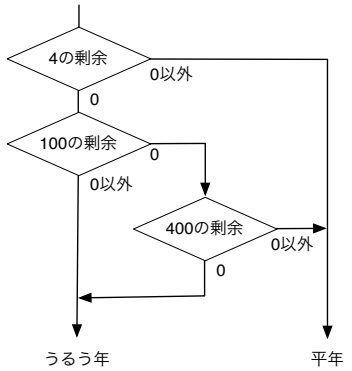
基礎プログラミング演習 II 教材 (#2)

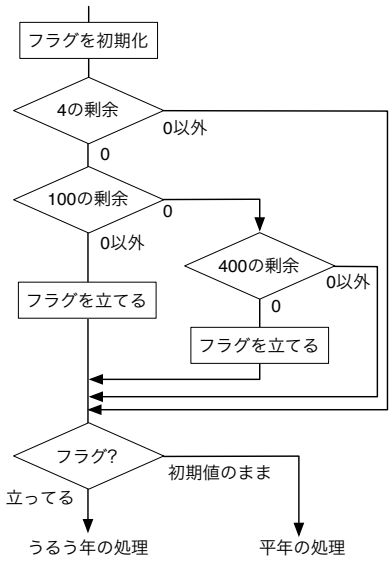
■ リハビリ演習 4（フラグ・論理演算子）

□ うるう年判定



ある年がうるう年か否かを調べるプログラムを作ります。うるう年の定義は以下の通りです。
1. 4 で割り切れる年はうるう年
2. ただし 100 で割り切れる年は平年
3. ただし 400 で割り切れる年はうるう年

この定義そのままに判定処理の手順を書き下すと恐らく右（上）図のようになるでしょう。
これをプログラムに使える手順として書いたものを右（下）の流れ図に示します。
「フラグ」と呼ばれる手法を用いています。



プログラミングにおけるフラグ(flag) とは、何かの条件が揃ったり、ある処理を通過したことを判定するためのテクニックのひとつです。
それを表現する変数（フラグ変数などと言います）を用意しておき、事象が発生した時にフラグ変数をセットし、後でフラグ変数の中身をチェックして反応する処理を実行します。
つまり「前段で判定」し、「あとで反応する」処理に使えるパターンです。

下に実際のコードと実行結果を示します。
実行して西暦年を入力すると、うるう年に leap とつけて表示します。

```
int year, leapflag=0;

scanf("%d", &year);
printf("%d", year);

if( year % 4 == 0 ) {
    if( year % 100 != 0 ) {
        leapflag = 1;
    } else {
        if( year % 400 == 0 ) {
            leapflag = 1;
        }
    }
}

if( leapflag == 1 ) printf(" leap");
printf("¥n");
```

注目して欲しいところ：
・ leapflag 変数をフラグとして使っている
・ はじめにゼロで初期化している
・ うるう年が確定したところでフラグを立てている（1 を代入する）
・ 最後にフラグが立っているか否か確認して処理している

□ 論理演算子

しかし定義とコードをよくよく見てみると「4 で割り切れること」は必須条件であり、これと同時に「100 で割り切れ*無い*か、あるいは 400 で割り切れ*る*こと」の「どちらか一方」が成立すればうるう年の条件が成立することがわかります。
C 言語にはこのような複数の条件節の結果（真偽）を「かつ (AND)」「または（OR)」で結ぶ「論理演算子」が用意されています。
教科書：5.6 論理演算子を参照

論理演算子を用いれば、上のコードは右の一つの if 文できれいに表現できることがわかります。
（そしてフラグが不要になった！）

```
if( ( year % 4 == 0 ) &&
    ( ( year % 100 != 0 ) || ( year % 400 == 0 ) ) ){
        printf(" leap");
}
```

実際にそのようなプログラムを作って、動作を確認してください。

■ 二つのプログラムの出力を較べて動作確認する

うるう年判定のために作った（論理演算子を用いた）プログラムがサンプルのプログラムと同一の結果を出すことを機械的に確認する方法を示します。

1. 標準入力を echo コマンドで与えて実行（パイプライン）

うるう年判定プログラムは標準入力から年を入れて、うるう年には leap と付けて表示します。実行するたびにキーボードから年を打ち込むのではなく、まず echo コマンドで与えて動作することを確認します。(右図参照)

```
$ echo 100 | ./a.out
100
$ echo 104 | ./a.out
104 leap
$
```

2. シェルスクリプトによる連続実行

まず leap.sh を取得してください。これは 1 から 2100 までの数字について、上例のように echo コマンドとパイプラインを使って繰り返して a.out を実行するシェルスクリプトファイルです。
これを右例のように bash に与えて実行してください。すると画面に結果がドバーと流れて行くでしょう。

```
$ bash leap.sh
1
2
3
4 leap
5
（以下略）
```

3. 実行結果の保存（リダイレクション）

次に右例のようにリダイレクションを付けて再度実行し、その結果を適当なファイルに保存して下さい。
（右例の「>」記号に注目。例では wc コマンドに l (エル)オプションをつけて、ファイルの行数を確認した。）

```
$ bash leap.sh > leap2.dat
$ wc -l leap2.dat
   2100 leap2.dat
$
```

4. 正しい結果の取得と照合

次に leap1.dat を取得し、less コマンドやエディタなりで中身を確認してください。これは正しく動作することを確かめたプログラムによる leap.sh の実行結果です。
つまり実行結果をそれぞれファイルに残し、右のように両者を diff コマンドで比較することで、自分のプログラムが正しく動作していると確認できます。

```
$ diff leap1.dat leap2.dat
$
```

□ diff コマンドによるファイルの比較

Unix 環境には二つのテキストファイルを比較するための diff コマンドがあります。

$ diff ファイル１　ファイル２

と、コマンド引き数として二つのファイルの名前を指定して実行すると、そのファイルの違いを表示してくれます。動作確認のために、まず作成したプログラム二つに対して実行してみると良いでしょう。違いがある箇所を > や < の記号つきで表示してくれます。両者に全く違いがない場合は、結果は何も表示されません。

diff コマンドを使って作成した二つの実行結果ファイルを比較すれば、それが同じであることを確認できます。

もし diff コマンドが何か結果を出力した場合、あなたの作ったプログラムにはバグがあります。原因を調べて直し、再び実行してテストしてください。

（一部だけ違うファイルを用意して、違っていた場合の結果を見ておくように。何でも確認、です。）

□ 参考：自動的なスクリプトの作成

先の動作確認では leap.sh シェルスクリプトを用意していましたが、実際にはこのようなものを自分で作る必要があります。以下に ruby のスクリプトを用いて連続的な数字を出力し、これを整形して目的のスクリプトを合成する例を示します。

まず ruby コマンド（これは ruby プログラミング言語の処理系です）を使って、echo 1 から echo 10 までのコマンド列を試しに作ってみます。

```
$ ruby -e '(1..10).each {|num| printf("echo %d\n",num)}'
echo 1
echo 2
echo 3
echo 4
echo 5
echo 6
echo 7
echo 8
echo 9
echo 10
```

ruby のスクリプトで 1 から 10 まで出力する echo 列を生成

結果が正しく出ていることを確認

echo だけでは意味がありませんが、パイプ（ | ）で a.out プログラムにつなぐコマンド列として作ります。このコマンド記述（echo 10 | ./a.out 等）を実行すれば、指定の年がうるう年かどうかテストすることができます。

```
$ ruby -e '(1..10).each {|num| printf("echo %d | ./a.out\n",num)}'
echo 1 | ./a.out
echo 2 | ./a.out
echo 3 | ./a.out
echo 4 | ./a.out
echo 5 | ./a.out
echo 6 | ./a.out
echo 7 | ./a.out
echo 8 | ./a.out
echo 9 | ./a.out
echo 10 | ./a.out
```

パイプで ./a.out に出力するように修正

結果が正しく出ていることを確認

これをリダイレクション（ > ）によって leap.sh などといったファイルに保存し、bash コマンドで実行させます。

例は 1 から 10 までの繰り返しですが、(1..10) を (1..2100) などとすることで今までの全西暦年についてテストするコマンド列を出力させることができます。

```
$ ruby -e '(1..10).each {|num| printf("echo %d | ./a.out\n",num)}' > leap.sh
$ bash leap.sh
1
2
3
4 leap
5
6
7
8 leap
9
10
$
```

リダイレクション（ > ）でファイルに保存

保存した結果をbash で実行

結果が正しく出ていることを確認

ところでスクリプトはリダイレクションを用いてファイルに保存しなくても、パイプラインを用いて直接 bash に吸い込ませることができます。以下のようにするのです。

```
$ ruby -e '(1..2100).each {|num| printf("echo %d | ./a.out¥n",num)}' | sh > leap1.dat
```

なお ruby では –e 以降の ‘‘ に囲まれた処理が実行されます。(1..2100).each は指定範囲の数値を変化させながら繰り返し、|num| のように | 記号で囲まれた変数にその値が入ります。printf()は C 言語のそれとほぼ同じように動作します。
C でもこうした処理は書けますが、簡単なスクリプト言語の簡単な処理パターンを丸覚えして使えるようにすることを勧めます。作業効率が断然上がります。

□ Ruby プログラミング言語

Ruby はプログラミング言語です。まつもとゆきひろ氏が作り、オープンソース・ソフトウェアとして開発が続けられています。業務で使う事例も増えています。

上の例ではプログラムによってスクリプトを作成することを通じて、手作業ではなくコンピュータを使ってコンピュータの操作を効率化・自動化することのパワフルさを示しました。この種の事は研究作業でも現場の仕事でも、とても普通に行われています。

ここでは Ruby を使っていますが、この種のテキスト処理が得意な言語処理系を一つは修得しましょう。Ruby でなく、例えば perl でも何でも良いでしょう。

■ リハビリ演習 5（ループ・break・フラグ）

□ 素数判定を行う

入力された数が素数か否か判定する処理について考えます。素数ですから「入力値が 2 から(入力値-1)までのどの数でも割りきれることがない」が判定条件となります。つまり「ループで片端から試しに割ってみて、一度も割り切れたことが無い」ことを確認すれば良いわけです。

入力値を n とすると、
・ i を 2 から n-1 までループさせ、
・ 一度も n が i で割り切れることなく
・ ループを最後まで回り終えたら、n は素数である
と判定するロジックを右図に描いてみました。

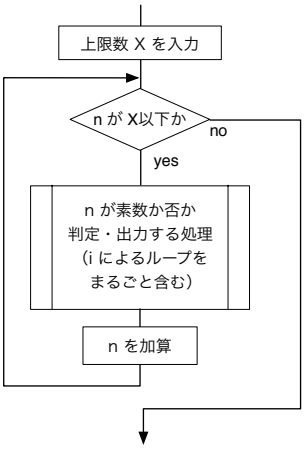
コードは右のようなものになるでしょう。
（教材 Web に置いてあります）

実行して動作を確認してください。

```
for(i=2; i<number; i++) {
  if( number % i == 0 ) { // 割り切れた
    printf("%d is not prime¥n", number);
    exit(0); // プログラムを終了する
  }
}
// ループを回りきった
printf("%d is prime¥n", number);
```

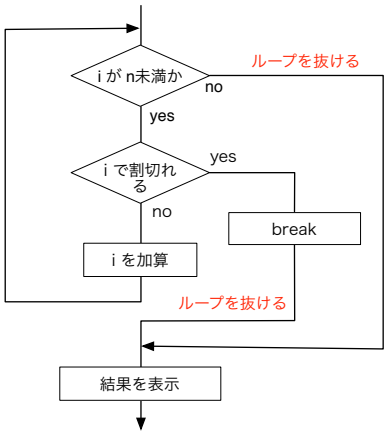
□ 素数列を表示する

次にこれに手を加えて、「入力値 n」だけでなく、「2 から入力値 X までの全ての数」について素数判定を行い、その結果を表示するプログラムを作って下さい。



右にロジックを示しますが、基本的には上に示した素数判定の処理全体をループに取り込んで繰り返して実行すれば良いはずです。ところが難点がひとつあって、上のサンプルコードは途中で実行終了することで、素数では無いことを見分けています。しかし素数を列挙させるためには判定処理を繰り返して行う必要があり、途中で終了させるわけにはいきません。

つまり、まず上のロジックについて、
・ 実行終了させることなく素数と判定させるように修正し、
・ それから全体をループさせる
ように修正してください。



「実行終了させることなく判定する」方法としては主に二つあると思います。

1. for 文によるループでは、ループを回り終えたときループカウンタが上限値になっている（最初の例の場合なら、for ループの後では i は n に等しくなっている筈）ことを見て判定する。
2. フラグを用いる。はじめフラグは倒しておき、ループの途中で割り切れた時にフラグを立て、ループを回り終えたときにフラグの状態を見て判定する。

いずれの場合でも、一度でも割り切れた時は break によるループからの途中脱出が可能です。そうすることで無駄な処理をせずに済みます。
右図にループを break で途中脱出する場合の流れを図示します。
break については基礎プロ I「条件分岐と繰り返し」や教科書 p.168 を参照。

□ 実行結果の確認

2 から入力した数まで素数判定を行い、右のような結果を出力するプログラムを作成し、教材 Web に用意されている、100 までの素数列判定結果と照合して、正しく動作していることを確認してください。自分のプログラムの出力をリダイレクションによってファイルに残し、正解データと diff コマンドによる比較を行えば良いでしょう。

```
2 is prime
3 is prime
4 is not prime
5 is prime
6 is not prime
…（中略）…
100 is not prime
```

■ リハビリ演習 6（ループ・配列）

□ 金種計算

入力された金額を各種硬貨で支払う場合、どの硬貨を何枚払えば良いか表示するロジックについて考えて下さい。

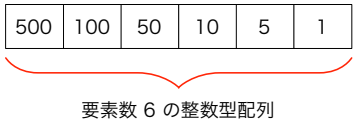
1. 支払い金額を設定し、
2. まず 500 円硬貨が何枚必要か計算し、
3. それで払った後の残額を計算し、これを次の金種での支払金額とする
4. 次の金種について 2. 3. 相当の処理を行い、これを繰り返す

処理 2. は支払額と対象硬貨の額の商、処理 3. はその剰余で得られることが分かるでしょうか。以下にこのロジックをそのまま書いたコードと、そうして作ったプログラムの実行結果を示します。

```
num500 = total / 500; total = total % 500;
num100 = total / 100; total = total % 100;
（中略）
num5   = total / 5;   total = total % 5;
num1   = total / 1;   total = total % 1;
```

```
$ echo 1234 | ./a.out
500:2 100:2 50:0 10:3 5:0 1:4
$ echo 567 | ./a.out
500:1 100:0 50:1 10:1 5:1 1:2
$
```

処理を見ると 6 回、同じ操作を繰り返していることが分かります。つまりこれはループで実現すべきことです。ループを回るごとに、500,100,...と、異なる額面で操作を繰り返せば良いのです。
つまり 6 つの要素を持つ配列を用意し、その各要素に各金種の額面を設定し、それを利用して処理を繰り返せば良いのです。

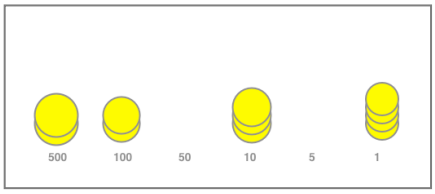


出来上がったら Moodle で提出してください。なお出力結果は上記の例と同じフォーマットになるよう注意して下さい。正しく動作していることを確認するために重要です。

注意：動作確認をしっかり。結果比較のために、上の例と全く同じ出力形式となるようにしてください。

■ 課題：金種計算の結果を可視化する

次に金種計算の結果を何らかの形でグラフィカルに表示してください。



キックスタート最後の課題として、自由度を高めに設定します。何かしら自分に出来る最高のものを提出してください。例えば右（の上側）に示したものは硬貨の枚数を円の重なりで表現したものです。少し凝ってみたポイントは、各硬貨のサイズを反映させたことです。

円の下にある数字は HgText()関数で出したものですが、HgText()関数は文字列しか受け付けないため、右に示すような記述で数値(100)を文字列(文字型配列変数 str)に変換して描画できます。文字型についてはこれからやりますので、これだけでは意味が分からないでしょうが、参考として。

```
char str[5];
sprintf(str, "%d", 100);
HgText(50.0, 50.0, str);
```

右（の下側）の例は単純な円形の代わりに HgImageLoad()関数と HgImagePut()関数で硬貨の絵柄を表示させています。
絵柄は教材サイトに置いてありますが、ネット上で検索すれば無償利用が可能なものは多く見つかるでしょう。見栄えの良いものを使ってくれれば良いと思います。

制限はありませんので、アニメーションでも何でもトライしてください。
出来上がったら Moodle で提出してください。

（上の例のようにプログラムコードだけでなくデータファイルなどが必要な場合はそれらをまとめて圧縮した ZIP 型式ファイルで提出してください。）