ゼロからはじめるプログラミング講座(Perl) #2 @越谷 講義ノート

配列を思い出す

- 配列の定義のしかた

配列の基本は、全体を「丸いカッコ"("と")"」で囲う。中の要素は、文字列ならシングルクォート（'）で囲う。ダブルクォートで囲ってもいいけど、それを使うのは、変数展開をするときに適している。数字なら、クォートで囲う必要はない。

- 配列の基本形 - 配列のシジルは@

```
@eras = ('明治', '大正', '昭和', '平成');
```

- 別にダブルクォートで文字列を囲んでもいい

```
@eras = ("明治", "大正", "昭和", "平成");
```

この他に、カッコやクォートを書くのがめんどくさい場合に、qwというのがある。

カッコやクォートを書くのがめんどくさかったら

```
@eras = qw/明治 大正 昭和 平成/;
@eras = qw!明治 大正 昭和 平成!;
@eras = qw{明治 大正 昭和 平成};
```

qw は quote words の略で、words（単語）をクォートして（囲って）ね、という意味。普通は、qw/単語1 単語2 単語3../と、始まりと終わりをスラッシュ（/）でくくるけど、ここの文字は他の文字でもよい。例えば、 | とか ! とか # とか $ とかでもOK。かなりいろいろな記号が許容される。また、４種類のカッコのペア () と{} と [] と <> でもよい。

この qw// の記法はクォートとカンマを省略できるメリットの他に、もう一つついいことがあって、それは、クォートの記号のエスケープをしなくてすむというところ。

例えば、「明治」という単語の中が、（なぜか）「明'治」となって、「明」と「治」の間に、シングルクォートが入っていたとしよう。それを配列に入れようとすると、「明」と「治」の間のシングルクォートは、単語をくくっているクォートとは別なんだよ、と示すために、前にバックスラッシュ（円マーク）をつけないといけない。（下記のように）

```
@eras = ('明¥'治', '大正', '昭和', '平成');
```

これを、「エスケープする」というのだけど、こういうことが増えてくると、くくるためのクォートなのか、単語の中に（たまたま）含まれているクォートなのかを判別して、エスケープしていくのが、煩雑になってくる。

しかし、qw// の場合は、くくる文字がシングルクォートではないので、エスケープする必要が生じない。

```
@eras = qw/明'治 大正 昭和 平成/;
```

じゃあ、文字の中にスラッシュ（/）が入ってしまったら？そうしたら、/以外の文字でくくってやればいい。

```
@eras = qw!明'治 大/正 昭和 平成!;
```

このように、エスケープする必要が生じたら、それをしなくていいように囲う文字を造作もなく変えることができるので、qw// は便利なのである。

配列の要素を呼び出すには

配列は、何番目の要素かを指定してやることで、その要素を呼び出すことができる。

たとえば、

```
print "$eras[0] ¥n";  # 「明治」とでてくる
```

注意しなくてはいけないのは、最初の要素は1番目ではなく、0番目であるということ。それから、添字は [ ] という角カッコで囲むこと。（違うカッコの形、たとえば { } だと違う意味になってしまいNG）

表示するのは、

```
print "$eras[0] ¥n"; # 明治
print "$eras[1] ¥n"; # 大正
print "$eras[2] ¥n"; # 昭和
print "$eras[3] ¥n"; # 平成
```

とやればいいけど、もっとたくさんの要素数になったら大変。どうする？→ループを使おう

for文

- C言語っぽい書き方

```
#　はじめ おわり 1ステップ進んだら
for ( $i = 0; $i < 4; $i++ ){
    print " $eras[ $i ] ¥n ";
}

for ( 条件式 ){
    ループ１回の処理内容
}
```

最初の $i = 0 は、$i という変数を 0 という値にしていて、初期化の意味合い。２番目は、このループが継続される条件を示していて、この場合だと、4より小さければ続行。３番めの $i++ は $i を１増やすという意味。ちなみに、$-- は１減らす。

このループを日本語にしてみると、

最初は $i は 0
$i は 4より小さいので { } の処理をする
終わったら $i を 1増やす→ $i は 1 になった！
ループの先頭に戻ってくる
いま $i は 1 で、 4より小さい→ループ続行
{ } の処理をする
終わったので $i を 1 増やす→ $i は 2 になった！
ループの先頭に戻ってくる
（中略）
$i を 1増やす→ $i は 4 になった！
ループの先頭に戻ってくる
いま $i は 4 で、 4より小さくない！→ループ終了

という感じである

{ } の中では、条件式で使った $i を使用している。

```
print " $eras[ $i ] ¥n ";
```

これは、 $i 番目の添字の要素を表示するという意味になっており、$i は１ずつ増えていくので、ループがまわるごとに１個ずつ要素が表示されていくことになる。

foreachを使うやり方

C言語ライクなfor文は、配列の添字を１個ずつずらしていくことで、一個ずつ配列の要素を取り出していく方法だった。この方法だと、添字（$i）をわざわざ用意して、それを一個ずつ増やしていくという ことをしなくてはならないし、添字の始まりはいくつで、終わりはいくつという”境界条件”について、考えなければ書くことができない。

添字を使う書き方は、慣れてしまえばどうということはないかもしれないが、やはり煩わしいのではないか。

その煩わしさから開放される書き方が foreach文による書き方だ。

実は、for文とforeach文はPerl的には等価なので、以下のコードはforeachと書いてあるところを、forと書いてもらっても全く問題ない。

```
foreach $jidai ( @eras ){
    print $jidai, "\n";
}

#解説
foreach 要素 ( 配列 ){
    ループ一回分の処理
}
```

配列という入れ物の中から、要素を一個ずつ取り出していって、中身がなくなったらそれで終わり、というイメージである。わざわざ始まりと終わりを指定するコードを書かないで済むので、個人的に書いていて楽だと感じる。

今回は、@erasから要素を一個取り出したら、$jidaiという変数でそれを受け取ったが、変数名は自由につけて構わない。

ちなみに、要素を受け取る変数名は、省略してしまってもいい。その場合は、「$_」という特殊変数に要素がセットされる。

```
foreach ( @eras ){
    print $_, "\n";
}
```

if文

今度は条件分岐のための構文の、if文である。

例えば、平成という年号は1989年にはじまるので、もし1995年生まれだったら平成生まれだし、1980年生まれだったら昭和生まれだということになる。

その人が平成生まれか昭和生まれか、プログラムで判定するプログラムを書いてみることにする。

```
print 'Born year?:';

$b_year = <STDIN>;

chomp( $b_year);

if( $b_year < 1989 ){
        print "Showa!\n";
}else{
    print "Heisei!\n";
}
```

最初は、「生まれた年は？」と聞いてるつもりで、「Born year?」と表示し、その直後で、入力を受け付けるようになる。（）受け取った入力の改行コードを削るのが chomp($b_year) で、このあたりは第一回でやったのでそちらを参照してほしい。

$b_year という変数に何年生まれかの数字（おそらく19ではじまる４桁の数字）が格納されていて、それが if文 の条件式にはいる。

```
if( $b_year < 1989 ){
        print "Showa!\n";
}
```

「 $byear < 1989 」は$byearの値が1989よりも小さい、という意味で、この式が正しければ、つまり、$b_yearに入っている数字が1989より小さければ、そのあとの { } で囲まれたブロックが実行される。今回は、単純に「 Showa!」と表示するだけである。

if文 は、単にこれだけでも成立する。

しかし、これだけだと、1989以降の値を入力した人には何も表示されなくて切ないことになってしまう。そこで、 else という分岐を加えて、先の「$b_yearの値が1989よりも小さい」という条件に合致しない場合は、こちらのブロックを実行せよ、と指示している。

今回は、1989以上の数字が入力された場合は、「 Heisei! 」と表示するようになっている。

```
if( $b_year < 1989 ){       # 1989より小さかったら
    print "Showa!\n";  # 昭和！
}else{                          # そうじゃなかったら
   print "Heisei!\n";          # 平成！
}
```

このように、if文はelseと同時に使うことが多い。

そして、もう一つ、if と else の他に、 elsif というものがある。

elsif

注意点は、「 elseif 」でも「 else if 」でもなくて、「 elsif 」だということである。

先の、 if - else の場合は、「条件Aが真ならばこれこれ、そうでなければそれそれ」、という二者択一的な分岐だった。elsifを用いると、「条件Aが真ならばこれこれ、（条件A、は真ではないけど）条件Bが真ならばあれそれ、そうでなければそれそれ」と分岐の選択肢を増やすことができる。

1989年に生まれた人は、生まれた日によっては、平成生まれかもしれないし、昭和生まれかもしれないので、もし、1989と入力されたときは、即「平成！」とレッテルを貼るのは不正確である。ということで、以下のように修正をした。

```
if( $b_year < 1989 ){
    print "Showa!\n";
}elsif($b_year == 1989){
    print "Showa or Heisei!\n";
}else{
    print "Heisei!\n";
}
```

今度は、 elsif($b_year == 1989) という条件が加わった。今度は、$b_yearが1989に一致したら、という意味になる。イコールが２つつながる（ == ）と数字として一致するという意味になります。

「 $b_year == 1989 （$b_yearが1989と一致するという条件式）」と「 $b_year = 1989 （$b_yearに1989という値を代入）」は「 = 」が一個あるかないかの違いしかありませんが、意味がまったく違うので、注意しましょう。

練習：for文とif文を組み合わせる

配列の宣言の仕方として、こんなのもあります。

```
@years = ( 1970 .. 2013 );

#上は下と等価
@years = ( 1970, 1971, 1972, 1973, （中略）, 2010, 2011, 2012, 2013 )
```

1970 .. 2013 と書くことで、1970から１ずつ増やして2013までの連続した数字の配列をあらわすことができます。何十もの要素をもつ配列ですが、こんなに簡単に表現できてしまいます。

これを使って、 1970年から2013年までを１年ずつ表示し、それが平成か昭和か判定するプログラムを書いてみます。

```
@years = (1970..2013);

for $y ( @years ){
    if( $y < 1989){
        print "$y : Syowa !\n";
    }elsif( $y == 1989){
        print "$y : Syowa or Heisei !\n";
    }else{
        print "$y : Heisei !\n";
    }
}
```

forのブロックの中に、if文で条件分岐し、条件ごとに違った文字列を出力しています。 自力で書くことができたでしょうか？

ハッシュの練習

ハッシュは前回触れましたが、もう一度ハッシュを扱う練習です。

ハッシュとは、キーとバリュー（値）がペアになっている変数の集まりのことです。

例えば、平成は1989年からで、昭和は1926年から、大正は1912年からで、明治は1868年から、という風に、年号と始まりの年のペアをそれぞれキーとバリューと考えた時に、それを表現するハッシュを書いてみます。

```
%era_year = (   'Heisei' => 1989,
                'Syowa' => 1926,
                'Taisyo' => 1912,
                'Meiji' => 1868
            );
```

これは見やすくするために、複数行にわけて書いていますが、一行にしてしまっても構いません。

また、キーとバリューを結ぶ記号として「 => 」を使っていますが、 実は、単にカンマでもOKです。奇数目の変数がキーで偶数目がバリューだとPerlは解釈できるからです。しかし、人間の目にはわかりずらいので、たいていは「 => 」を使って書きます。

```
#一行にしちゃってもOK（でも見にくい）
my %era_year = ( 'Heisei' => 1989, 'Syowa' =>
1926, 'Taisyo' => 1912, 'Meiji' => 1868 );
# => 使わなくてもOK（でも見にくい）
my %era_year =
( 'Heisei', 1989, 'Syowa', 1926, 'Taisyo', 1912, 'Meiji',
```

キーは文字列なので今回はシングルクォートで囲んでいますが、ダブルクォートで囲んでもいいですし、実は文字列が一定の条件（ハイフンをふくまないなど）をみたせば、クォートを省略することができます。 Perl は => という記号を目印に、その左側は文字列であると判断できるからです。

バリューの方は、その変数の性質（数字なのか文字列なのかオブジェクトなのか変数なのか）によって、適切にクォートをつけたりしてください。

ハッシュとfor文を組み合わせる練習

今度は、ハッシュと先ほど練習したfor文の練習です。

「（年号）は（西暦）年から始まります」

という文を、次々に表示するコードを書いて下さい。

ハッシュのキーは keys という関数を使うと、キーだけを一括して取り出し、配列にすることが出来ます。 そのキーの配列をfor文で回して、ハッシュ

```
@era = keys %era_year;

for $jidai (@era){
    print "$jidai は $era_year{$jidai} にはじまります。\n";
}
```

use strictとmy

ここまでやってきて、プログラムが動かない！という場面に何度も遭遇したと思いますが、変数名を打ち間違えていたというケースがけっこう多かったように見受けられます。

例えば @era と書くところを @ear と書いてしまったり、$eras[$i]と書くところを@eras[$i]と書いていたり。

そのようなたかが１文字違うだけでも、Perlはまったく違うものとして扱うため、プログラムは止まってしまいます。そして、人間の目には１文字の違いは見落としがちで、なぜプログラムが動かないのかわからずに煩悶しがちです。

そういうミスを回避するために、「use strict;」を使います。

use strictを使うと、変数は、使う前にあらかじめ宣言することが必要になります。use strictの使い方は、単に、ファイルの中に「use strict;」と一行書くだけです。各場所はどこでもいいのですが、たいていはソースコードの一番上、シバン行のすぐ後に書きます。

```
#!/usr/bin/perl

use strict;

my $words = 'Hello, World';
print $words, "\n";
```

上のコードでは、 $words という変数の前に my というものがくっついていることに気づいたと思います。

my を変数の前に書くと、「この変数名をこれから使います！」という宣言になります。「私の」という意味のmy です。（似た言葉に our があります)

先ほどのハッシュのコードは、use strictすると、下記のように書かないと動きません。

```
use strict;

my %era_year = (    'Heisei' => 1989,
                    'Syowa' => 1926,
                    'Taisyo' => 1912,
                    'Meiji' => 1868
                );

my @era = keys %era_year;

for my $jidai (@era){
    print "$jidai は $era_year{$jidai} にはじまります。\n";
}
```

こうすると、何が嬉しいのでしょう？ my という文字を打つ手間が増えるだけじゃないかと思うかもしれませんが、そうではありません。

use strictすると、宣言をした変数しか使えなくなるため、 @eraと書くべきところで @earと書いてしまったら、Perlがエラーメッセージを吐いてくれるようになります。

@eraは宣言されていますが、@earは宣言されていません。なので、宣言されていない @earが突然でてきたら、Perlは、「そんな変数を使うなんて聞いてない」とプログラムを停止し、エラーを出力します。

そのようにエラーがでてくれば、プログラマは、すぐにどこかでミスをしたことに気付くことができます。もし use strict をしていないと、@earと書いてあっても、ひとまずはプログラムが止まることなく動いてしまうので、プログラマはミスに気付くのが遅くなります。

use strictすると、他にもいろいろと"厳しく"なるのですが、その厳しさは、どれもプログラマを助けてくれるためなので、今後はuse strict; を必ず書くようにして、変数には my をつけて宣言してから使うようにしましょう。