PERL 程式設計

Perl 是一個功能非常強大的 script 語言。它的全名是 Practical Extraction and Report Language。 Larry Wall 是它的創造者。從 1988 年 1 月發行第一版到現在的第五版。Perl 是我們網路上經常用的 CGI 語言，因此要架站，必需了解它。Openwebmail 網路郵局、MRTG 網路流量偵測和 Webmin 網路管理軟體都是使用 Perl 寫成，因此我們可以了解 Perl 的重要性。一般網管人員、系統開發者和網頁設計師都必需了解 Perl。由於 Perl 是 CGI 語言，因此很容易學習。Perl 包括了許多的套件與模組， 因此 Perl 適合於做檔案輸出、輸入和存取資料庫系統。因為在 Linux 中已經內建 Perl 軟體，所以我們可以直接執行它。Perl 的網站是 www.perl.com。我們在這裏主要實作 Perl 的作業系統是 Unix 和 Linux。

範例 hello.pl

這是我們第一個程式 hello.pl。我們使用文字編譯器 vi 來編輯。我們使用 print 來輸出字串"你好"，而"\n"是換行符號。我們使用 perl hello.pl 來執行 hello.pl 程式。

#vi hello.pl

print "你好\n"

```
[root@flash perl]# perl hello.pl
你好
```

我們也可以輸入/usr/bin/perl 絕對路徑到 perl 程式來執行 hello.pl。

```
[root@flash perl]# /usr/bin/perl hello.pl
你好
```

範例：welcome.pl

#是註解的意義，表是那一行沒有作用。而第一行的#!符號則是指令 Perl 直譯器的路徑，一般我們在 Linux 系統上是/usr/bin/perl。如果在 Windows 系統上，其直譯器的路徑是#!c:\perl\bin\perl.exe。我們在第三行使用 print 來列印出字串。"\n" 是換行符號。

```
#!/usr/bin/perl
# welcom.pl
print "歡迎來到Perl!\n";
```

我們使用 perl 來執行 welcome.pl

```
[root@flash PERL1]# perl welcome.pl
歡迎來到Perl!
```

1-1Perl 程式設計基礎

Perl 是 script 語言，很簡單。它經常被用在 CGI 的設計上，目前很多網站的設計都是使用 Perl 來撰寫的。

範例：semicolon.pl

我們使用；分號來代表在 perl 中該行已經結束。

第二行第二個 print 代表著已經是另外一行的程式，因為前面有；分號。

第二行的兩個 print 所輸出的結果和第三行與第四行輸出的結果是相同的，雖然第二行的第二個 print 是寫在第二行的；分號後面，其和換行寫是相同的意義。

```
1 #!/usr/bin/perl
2 print("分號代保該行結束");print("分號代保該行結束\n");
3 print("分號代保該行結束");
4 print("分號代保該行結束\n");
```

範例：print.pl

我們可以使用 print 來輸出字串或變數到螢幕。我們也可以使用 print 來輸出字串與變數。

第二行將 3 的數值給變數$variable。

第三行 print 列印出括號內的字串” 使用 print 來輸出變數$variable\n”， ”\n”在網頁中是表示換行的意義。

第四行使用 print 顯示”使用 print 也可以輸出變數$variable"。

```
1 #!/usr/bin/perl
2 $variable=3;
3 print("使用print來輸出$variable\n");
4 print "使用print 也可以輸出$variable\n";
```

因為使用\n 網頁換行符號，所以顯示兩行。

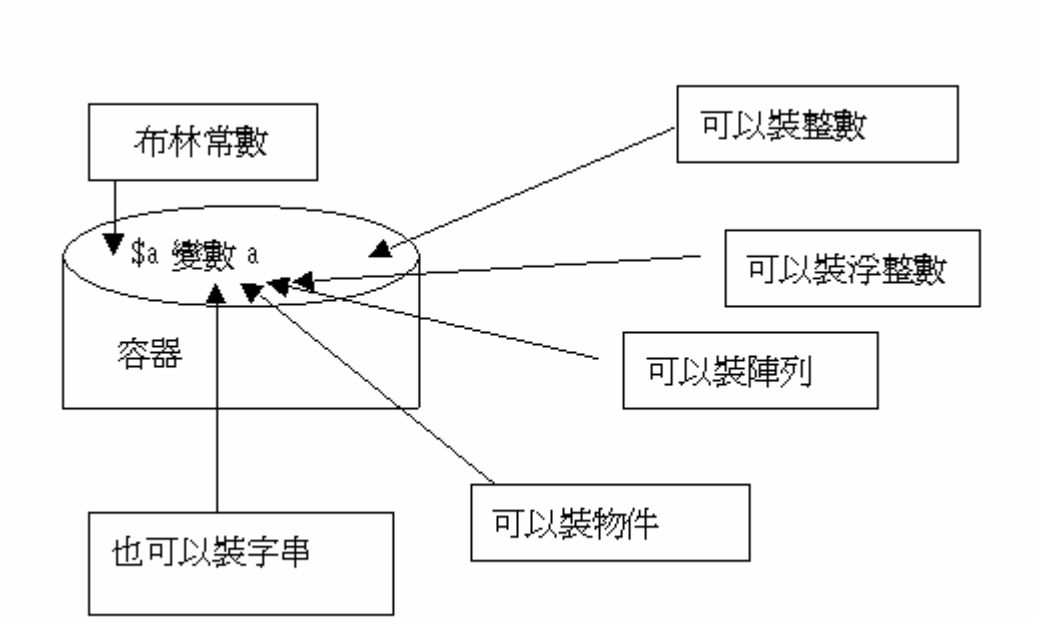
```
[root@flash perl]# perl print.pl
使用print來輸出3
使用print 也可以輸出3
```

1-1-1 **資料型態**

Perl **支援的資料型態有，陣列**(array)**、浮整數**(floating point)**、整數**(integer)**、物件**(object)**、布林常數**(Boolean)**、空值**(null)**和字串**(string)。

$這個錢符號為變數的符號，例如**$a **就是變數 a **的意義。**

這就像是容器可以裝很多東西一樣。



範例：add.pl

變數就像是容器一樣，我們將數值或字串給變數來儲藏，再將變數作運算處理。

第一行是指定 perl **程式的執行路徑。**

第二行是註解。

第五行是將輸入的資料給$number1 **變數。**

第六行的 chomp **函數會移走指定變數尾後的字串如換行符號**"\n"**，這經常用在輸入資料。**

第九行是將輸入的資料給$number2 **變數。**

第十二行是將變數$number1 **和變數**$number2 **相加然後再給**$sum **變數。**

```
1 #!/usr/bin/perl
2 #兩數相加
3
4 print "請輸入第一個數:\n";
5 $number1=<STDIN>;
6 chomp $number1;
7
8 print "請輸入第二個數:\n";
9 $number2=<STDIN>;
10 chomp $number2;
11
12 $sum=$number1+$number2;
13 print "總合是$sum\n";
```

這是我們執行的情況。我們輸入 5 **和** 6**，則得到** 11**。**

```
[root@flash perl]# perl add.pl
請輸入第一個數:
5
請輸入第二個數:
6
總合是11
```

(1)整數

一般我們都是以十進位為基底，這個變數 a 可以是正整數也可以是負整數。

整數就是 Z={..., -2, -1, 0, 1, 2, ...}**的集合。整數可以用十六進位、十進位或八進位來表示，前面可以加正負符號。**

範例：integer.pl

```
#!/usr/bin/perl
$a = 1234; # 十進位
print $a;
print "\n";
$a = -123; # 負數
print $a;
print"\n";
$a = 0123; # 八進位 (等於十進位的83)
print $a;
print "\n";
$a = 0x12; # 十六進位 (等於十進位的18)
print $a;
print "\n";
```

這是我們執行的情況。

```
[root@flash perl]# perl integer.pl
1234
-123
83
18
```

(2)浮點數

浮點數就是有小數的數值，例如 3.1425 **這就是浮點數。**

範例：float.pl

第二行的"\"**符號為跳脫字元，** "\n"**為換行符號。**

第三行是顯示浮點數$a=1.1235

```
#!/usr/bin/perl
print "\$a=3.1425\n";
print $a=1.1235;
print "\n";
```

```
[root@flash perl]# perl float.pl
$a=3.1425
1.1235
```

(3)字串

字串就是用"... "(雙引號)這個符號包起來的內容。字串就是由一連串的字元所組成。一個字元就像是一個位元組，而總共有 256 個不同的字元可以讓我們選擇。字串是沒有長度限制的。

我們可以使用單引號'、雙引號"。

範例：string.pl

我是大好人就是字串，它的前後都有雙引號"。

先將字串給$a 這個變數，再用 print 來顯示$a 這個變數。

```
1 #!/usr/bin/perl
2 $a="我是大好人";
3 print $a;
4 print "\n";
```

執行時就會顯示變數$a 的值"我是大好人。

```
[root@flash perl]# perl strings.pl
我是大好人
```

範例：sto.pl

'$'錢的符號前面需加跳脫字元\才能顯示出來。我們在第九行使用字串符號". " 就能將變數$c 的值和字串相連了。

```
 1 #!/usr/bin/perl
 2 $a=5;
 3 $b=3;
 4 $c=6;
 5 print   "$b我是好人";
 6 print   "\n";
 7 print   "\$c我是大好人";
 8 print   "\n";
 9 print   $c."我是大好人";
10 print   "\n";
```

```
[root@flash perl]# perl sto.pl
3我是好人
$c我是大好人
6我是大好人
```

跳脫字元，單引號內所包含的字串都會顯示。'$'錢的符號前面需加跳脫字元\才能顯示出來。

跳脫字元	說明
\"	將雙引號"當作字元
\n	換行
\$	將$當作字元
\\	將\當作字元
\'	單引號'當作字元

(4)陣列

陣列就是一個連續的空間放著個種資料。

例如下面是一段記億體的位址。

0	你好
1	我好
2	大家好
3	123
4	就有
5	0.567
6	thanks

範例 array.pl

陣列表現的方法就像下面的敘述。

第二行將"你好"放到陣列 a **在** 0 **的位址。**

第三行將"我好"放到陣列 a **在** 1 **的位址。**

第四行將"大家好"放到陣列 a **在** 2 **的位址。**

第五行將整數 123 **放到陣列** a **在** 3 **的位址。**

第六行將字串"就是"放到陣列 a **在** 4 **的位置。**

第七行將浮整數放到陣列 a **在** 5 **的位址。**

第九行中陣列$a **第** 3 **個數值** 123 **加陣列**$a **第** 5 **個數值** 0.567 **共為** 123.567**。**

第十一行字串相加為 0**。**

第十三字串加數值 123 **共為** 123**。它會自動將字串轉為數值** 0**, 再作相加的動作。**

```
#!/usr/bin/perl
$a[0]="你好";
$a[1]="我好";
$a[2]="大家好";
$a[3]=123;
$a[4]="就是";
$a[5]=0.567;
$a[6]="thanks";
print   $a[3]+$a[5];
print   "\n";
print   $a[0]+$a[2];
print   "\n";
print   $a[0]+$a[3];
print   "\n";
```

這是執行的情況。

```
[root@flash perl]# perl array.pl
123.567
0
123
```

範例：array1.pl

範例 array.php **的** a[]**陣列可以表示成第二行的表示法。**

```
#!/usr/bin/perl
@a=("你好","我好","大家好",123,"就是",0.567,"thanks");
print   $a[3]+$a[5];
print   "\n";
print   $a[0]+$a[2];
print   "\n";
print   $a[0]+$a[3];
print   "\n";
```

執行的情況一樣。

```
[root@flash perl]# perl array1.pl
123.567
0
123
```

範例：arrays.pl

陣列也可以使用二維的方式來呈現。下面就是一個 2 **維陣列**$a**，第一列我們存放著** 4 **個變數，第二列我們也存放著** 4 **個變數。**

第 0 **列第** 0 **行**	**第** 1 **欄**	**第** 2 **欄**	**第** 3 **欄**	**第** 4 **欄**
第 1 **列**	1	2	3	5
第 2 **列**	6	7	8	9

$a[1][1]=1;**將** 1 **這個數放到陣列**$a **第一列第一行**

$a[1][2]=2;

$a[1][3]=3;**將** 3 **這個數放到陣列**$a **第一列第三行**

$a[1][4]=5;**將** 5 **這個數放到陣列**$a **第一列第四行**

$a[2][1]=6;

$a[2][2]=7;

$a[2][3]=8;**將** 8 **這個數放到陣列**$a **第二列第三行**

$a[2][4]=9;**將** 9 **這個數放到陣列**$a **第二列第四行**

```
#!/usr/bin/perl
$a[1][1]=1;
$a[1][2]=2;
$a[1][3]=3;
$a[1][4]=5;
$a[2][1]=6;
$a[2][2]=7;
$a[2][3]=8;
$a[2][4]=9;
print $a[1][3]+$a[2][3];
print "\n";
```

將陣列$a **第一列第三行的值** 3 **加上第二列第三行的值** 8 **就會等於** 11。

```
[root@flash perl]# perl arrays.pl
11
```

範例：arraym.pl

我們可以用一維陣列，二維陣列，當然也可以用多維陣列。

第七行字串相加為 0。

第八行為換行符號”\n”。

第九行為合併符號逗點’.’。

```
#!/usr/bin/perl
$a[0][0][0]="a";
$a[0][0][1][0]="b";
$a[0][0][1][1]="c";
$a[0][0][1][1][1]="d";
$a[0][1][2][3]="d";
print  $a[0][0][0]+$a[0][0][1][0];
print  "\n";
print  $a[0][0][0].$a[0][0][1][0];
print  "\n";
```

字串相加為 0。

使用字串合併符號就可顯示 ab **連在一起了。**

```
[root@flash perl]# perl arraym.pl
0
ab
```

(5)物件

物件也是資料形態的一種。類別是抽像化(abstrction)**的，而物件就是實體化。對電腦而言，類別只是一些程式碼，而真正在記憶體上配置空間的是物件，也就是說物件才是我們互動的資料。物件導向使我們寫程式更為方便，雖然** Perl **不像** C++**或** JAVA **一樣這麼完整的設計，但對於** script **來說已經足夠了。當我們要使用之前開發過的程式，然後要改成目前計畫的程式，只要將它的類別繼承再作修改或增加，成為衍生類別**(derived class)**了。如果要將類別實體化，則要用** new **這個動作，來配置記憶體給這個物件，也就實體化了。物件導向還有一項優點，在類別裏面，有著許多的資料和方法，而只有在類別裏才看得到，這個就稱為封裝**(encapsule)。

1-1-2 **變數**

在 perl **中變數的表示就是前面加一個**$**錢的符號，後面加上名子。名子的寫法可以第一個字是大小寫英文字也可以是底線，但第一個字不可以為數字。**

第一個字元從 a-z、 A-Z **和** ASCII **字元從** 127 **到** 255 (0x7f-0xff).

範例：de.pl

變數的名稱為 var，**可以。**

變數名稱 4site **第一個字為** 4，**不可以。**

第一個字為底線，可以。

第五行及第六行宣告及使用不合法的變數$4site，**因為使用數字為開頭的變數。**

```
1 #!/usr/bin/perl
2 $var = "吳佳諺";
3 $Var = "吳東憬";
4 print  "$var, $Var"."\n";
5 $4site = "可不可以當變數";
6 print  $4site."\n";
7 $_4site = "可以當變數";
8 print  $_4site;
9 print  "\n"
```

這時只要執行就會發生語法錯誤。發生錯誤在第五行和第六行的$4site。

```
[root@flash perl]# perl de.pl
Bareword found where operator expected at de.pl line 5, near "$4site"
        (Missing operator before site?)
Bareword found where operator expected at de.pl line 6, near "$4site"
        (Missing operator before site?)
syntax error at de.pl line 5, near "$4site "
Execution of de.pl aborted due to compilation errors.
```

範例：dea.pl
將第五行和第六行去除後，就可以顯示資料了。
第七行使用_底線為開頭的變數。

```
#!/usr/bin/perl
$var = "吳佳諺";
$Var = "吳東憬";
print  "$var, $Var"."\n";
#$4site = "可不可以當變數";
#print  $4site."\n";
$_4site = "可以當變數";
print  $_4site;
print  "\n"
```

這是顯示的情況。

```
[root@flash perl]# perl dea.pl
吳佳諺, 吳東憬
可以當變數
```

perl **並不支援清楚的型態定義。在** c **語言理，整數是用** int **的形態、字元是用** char **的形態、浮點數是用** float **的形態，在** perl **裏頭變數都是** var **來表示，當要用到時的狀況，**perl **會自動決定變數的形態。例如我們將兩個字串相加**(+)**時，所得的就是** 0 **的數值，因為字串作數值運算時，**perl **會自動把字串轉為數值** 0**，相加的值就變成** 0 **了。這個就稱為資料的型別轉換。**
範例：casting.pl
在第四行字串相加時 perl **會把字串形態轉為數值** 0 **再相加。**

```
#!/usr/bin/perl
$a="我";
$b="要";
$c=$a+$b;
print $c;
print "\n";
```

$a+$b **的數值顯示為** 0**。**

```
[root@flash perl]# perl casting.pl
0
```

1- 2 運算子

變數$a=5，$c=$a+5，變數$c 的值為 10 在這個中，$a=5 是一個運算式，$c=$a+5 是一個運算式，+和=是運算子，變數$a 和數值 5 是運算元。我們可以了解運算式就是由運算子和運算元所組成。

我們的程式主要是由運算子與運算元所組成。我們的想法，轉換成程式的架構，再由運算子來架構骨架與運算元的組成，這樣就可以組成大部份的程式邏輯。

1-2-1 算術運算子

算數的加+ 減- 乘* 除-和餘數%稱作為算術運算子。我們常用算術運算子來作運算式，而程式的百分之五十以上都是由算數運算子所組成。

範例 arith.pl

第二行的一加一等於二，再將 2 指派給變數$a，使用加法運算子+。

第五行使用乘法運算子*，將 10 指派給變數$b

第十一行使用餘數運算子，13%6 值為 1；2.5%8 值為 2(2.5 取 8 的餘數為 2)。

```
1 #!/usr/bin/perl
2 $a=1+1;
3 print  $a;
4 print  "\n";
5 $b=5*$a;
6 print  $b;
7 print  "\n";
8 $c=25/$b;
9 print  $c;
10 print  "\n";
11 print  $c%8;
12 print  "\n";
```

這是執行的情況。

```
[root@flash 1-2]# perl arith.pl
2
10
2.5
2
```

範例：minus.pl
減號(-)可以配合數字而產生負數值。
第二行將 5 **分配給變數**$a。
第五行將變數$a **取負號，在分配給變數**$b，**因此可以得到-**5 **的值。**

```
1 #!/usr/bin/perl
2 $a=5;
3 print $a;
4 print "\n";
5 $b=-$a;
6 print $b;
7 print "\n";
```

```
[root@flash 1-2]# perl minus.pl
5
-5
```

1-2-2 **分配運算子**
在數學上我們稱 1 **加** 1 **等於** 2。**但在程式的執行有先後順序的過程，例如先一加一再將二的值指派給變數**$a，**這時變數**$a **就像是容器，把** 2 **給裝進去了。這個符號在數學上叫等於，在程式中叫分配運算子** assignment operators。

範例 assign.pl
第二行將數值 2 **指定給變數**$a，**然後第三行顯示變數**$a **的值。**
第五行將字串指派，指定給變數$a。

```
1 #!/usr/bin/perl
2 $a=1+1;
3 print  $a;
4 print  "\n";
5 $a="指派";
6 print  $a;
7 print  "\n";
```

這是執行的情況。

```
[root@flash 1-2]# perl assign.pl
2
指派
```

範例 assign2.pl

第三行將$a+5 **的值** 8 **分配給變數**$a

第七行設定$b **為**"Hello There!",**就像**$b = $b."There!";。

```
1 #!/usr/bin/perl
2 $a = 3;
3 $a =$a+5;
4 print  $a;
5 print  "\n";
6 $b = "Hello ";
7 $b = $b."There!";
8 print  $b;
9 print  "\n";
```

```
[root@flash 1-2]# perl assign2.pl
8
Hello There!
```

1-2-3 **位元操作運算子**(bitwise)

所謂位元操作運算子，就是將其運算元當作一個有次序的位元集合，並以其中的個別位元或群組位元為活動範圍，每一個位元的值不是 0 **就是** 1。

位元操作運算子	功能	用法
~	Bitwise not	~expr
<<	左移	Expr1 << expr2
>>	右移	Expr1 >> expr 2
&	Bitwise and	Expr1 & expr2
|	Bitwise or	Expr1 |expr2

範例 xor.pl

^**為**"**不相同單元二進位邏輯運算子**"。**當左邊為** 0 **而右邊為** 1，**或右邊為** 1 **左邊為** 0 **時，才會回傳** 1，**其它傳回** 0。15 **的二進位表示法為** 1111，9 **的二進位表示法為** 1001，**因此在第二行** 15^9 **後，會得到** 0110，**而** 0110 **就是十進位的** 6。

```
1 #!/usr/bin/perl
2 $x=15^9;
3 print $x;
4 print "\n"
```

```
[root@flash 1-2]# perl xor.pl
6
```

範例練習：bit_or.pl

|為單元或二進位邏輯運算子。當左邊為 0 或右邊為 1，或右邊為 1 左邊為 0 時，或當左邊為 1 右邊為 1，都會傳回 1，只有當左右兩邊都為 0 時才會傳回 0。15 的二進位表示法為 1111，9 的二進位表示法為 1001，因此在第三行 15 | 9 後，會得到 1111，而 1111 就是十進位的 15。

```
1 #!/usr/bin/perl
2 $x=15|9;
3 print $x;
4 print "\n";
```

這是執行的情況。

```
[root@flash 1-2]# perl bit_or.pl
15
```

範例練習：not.pl

~為單元反二進位邏輯運算子。它會將變數的正負符號相反(二進位取補數)。在第三行，~0 為 4294967295。在第六行，~1 為 42949677294。在第十二行，-2 的補數為 2，而 2 減 1 則為 1。

```
 1 #!/usr/bin/perl
 2 $x=0;
 3 print  ~$x;
 4 print  "\n";
 5 $x=1;
 6 print  ~$x;
 7 print  "\n";
 8 $x=-1;
 9 print  ~$x;
10 print  "\n";
11 $x=-2;
12 print  ~$x;
13 print  "\n";
```

```
[root@flash 1-2]# perl not.pl
4294967295
4294967294
0
1
```

```
Perl的無正負號整數使用32位元來儲存.所以1的二進位表示法爲
00000000 00000000 00000000 00000001
~1等於
11111111 11111111 11111111 11111110
這就是10進爲的4294967294

Perl的無正負號整數使用32位元來儲存.所以0的二進位表示法爲
00000000 00000000 00000000 00000000
~1等於
11111111 11111111 11111111 11111111
這就是10進位的4294967295
```

範例練習：left_shift.pl

<<為單位位元左移運算子。我們在第一行將 1(**二進位** 0001)**向左移** 10 **個數，因此可以得到** 10000000000，**而得到** 2 **的** 10 **次方為** 1024。**我們在第三行將** 7(**二進位** 0111)**向左移** 8 **個數得到**(011100000000)，**得到** 1024(2 **的** 10 **次方**)+512(2 **的** 9 **次方**)+256(2 **的** 8 **次方**)=1792，**因此會將** 1792 **分配給** x。

```
1 #!/usr/bin/perl
2 $x=1<<10;
3 print  $x;
4 print  "\n";
5 $x=7<<8;
6 print  $x;
7 print "\n";
```

```
[root@flash 1-2]# perl left_shift.pl
1024
1792
```

1-2-4 比較運算子

比較運算子和邏輯運算子，所得到的結果就是一個布林常數(0 或 1)。數值 0 代表 false 數值 1 代表 true。

當 3<5 時會回傳 true，當 3<2 時會回傳 false。比較運算子就是用來比較兩個運算式。

比較運算子	功能	用法
<	小於(less than)	Expr < expr
>	大於(greater than)	Expr > expr
>=	大於或等於	Expr>=expr
==	相等(equal)	Expr==expr
!=	不相等	Expr !=expr
<=	小於等於	Expr <=expr
<=>	左邊運算元小於右邊運算元則傳回-1。如果相等則傳回 0。如果大於則傳回 1。	Expr<=>Expr

範例：less.pl

當 3<5 時會回傳 true(1)。

```
1 #!/usr/bin/perl
2 print (3<5);
3 print "\n"
```

```
[root@flash 1-2]# perl less.pl
1
```

範例:less2.pl

3>5 時會回傳 false，因為 false 是空值。如果和其它的運算式組合起來則會發生錯誤。

```
1 #!/usr/bin/perl
2 print (3>5);
3 print "\n"
```

執行時傳回空值。

```
[root@flash 1-2]# perl less2.pl
```

範例:less3.pl

因為第二行會傳回空值，也就是錯誤的情況，所以發生語法錯誤的情況。

```
#!/usr/bin/perl
print (3>5);
print "\n"
print (3<5);
print "\n";
```

這是發生語法錯誤的情況。

```
[root@flash 1-2]# perl less3.pl
syntax error at less3.pl line 4, near "print"
Execution of less3.pl aborted due to compilation errors.
```

範例：equal.pl

第二行(5==5)**回傳** true(1)。

```
#!/usr/bin/perl
print (5==5);
print "\n";
```

```
[root@flash 1-2]# perl equal.pl
1
```

範例：less_than.pl

第二行因為 3 **小於等於** 5，**所以**(3<=5)**回傳** true(1)。

第四行因為 5 **小於等於** 5，**所以**(5<=5)**回傳** true(1)。

```
#!/usr/bin/perl
print (3<=5);
print "\n";
print (5<=5);
print "\n";
```

```
[root@flash 1-2]# perl less_than.pl
1
1
```

範例：not_equal.pl
第二行因為 3 **不等於** 5，**所以**(3!=5)**回傳** true(1)。
第五行因為 3 **小於** 5，**所以回傳**-1。

```
1 #!/usr/bin/perl
2 print (3!=5);
3 print "\n";
4 print (3<=>5);
5 print "\n";
```

```
[root@flash 1-2]# perl not_equal.pl
1
-1
```

1-2-5 **邏輯運算子**
邏輯運算子可以結合條件，以一個表達式判斷許多條件，而這些條件的結果不是真 true **就是假** false。
&&**或** and **稱為與邏輯運算子，只有當所有條件都成立時才會回傳真** true，**否則回傳假**。
||**或** or **稱為或邏輯運算子，只要運算式中一個條件成立就會回傳真** true，**只有當所有的條件都為假** fasle **時，才會回傳假** false。
!**為相反邏輯運算子，真** true **的條件加上**!**相反邏輯運算子時，就會變成假** false；**當假** false **的條件加上**!**相反邏輯運算子時，就會變成真** true。
Xor **為互斥運算子，當只有條件都不相同(互斥)時才會回傳真** true，**其它都回傳** false。**當(條件** A(true))Xor(**互斥**)(**條件** B(false))**傳回真，或當(條件** A(false))Xor(**互斥**)(**條件** B(true))**傳回真，其它則傳回** false。

邏輯運算子	**功能**	**用法**
&&	**邏輯運算** and(**與**)	Expr && expr
||	**邏輯運算** or(**或**)	Expr ||expr
!	**邏輯運算子** not(**否**)	!expr
And	And(**與**)	Expr and expr
Or	Or(**或**)	Expr or expr
Xor	**互斥** exclusive or	Expr xor expr

這些運算子的結果不是真(成立)就是假(不成立)，在寫程式時的邏輯判斷經常用到，可以多練習。在這裏 expr 指的是運算式 expressions。
布林 Boolean 代數定義在一個二元素的集合上，即 B={0,1}，0 為真，1 為假，再加上對兩個二元運算子 AND 及 OR 的規則表。在 AND 運算子中只有當 A 和 B 為真時才為真。在 OR 運算子中，只要 A 或 B 有任何一個為真就會為真。NOT 就是相反的意義，當 A 為真時 NOT A 就會為假；當 B 為真時 NOT B 就會為假，剛好和原來的 Boolean 值相反。A XOR B 就是 A EXCLUSIVE OR B，當只有 A 和 B 的值不相等時，才會回傳真 true。
在下面的真值表格 A，B 為兩運算式的結果布林值。

A	B	A AND B	A OR B	NOT B	A XOR B
真	真	真	真	假	假
假	假	假	假	真	假
真	假	假	真	真	真
假	真	假	真	假	真

範例：logical.pl
第二行不為假所以為 true(1)，因此執行第三行。
第六行 5 大於 3 所以 true(1)，因此執行第七行。
第十行兩個都相等所以 true(1)，因此執行第十五行。
第十四行因為 5>3 成立，在(3>5)||(5>3)只要有一個條件成立就為真，所以為 true(1)，因此執行第十五行。
第十八行 3 不等於 5 所以成立，因此執行第十九行。
第二十二行 3>5 不成立所以不成立，因此不執行第二十三行而執行第二十五行。

```
#!/usr/bin/perl
if(!(3>5)){
print  "不爲假(not false)時爲成立(true)";
}
print  "\n";
if(5>3){
print  "5>3成立";
}
print  "\n";
if(5==5){
print  "5等於5成立";
}
print  "\n";
if((3>5)||(5>3)){
print  "只要(3>5)或(5>3)時成立就成立(true)";
}
print "\n";
if(3!=5){
print  "3!=5稱3不等於5爲true";
}
print  "\n";
if((3>5) && (5>3)){
print  "哈";
}else{
print  "(3>5)和(5>3)要兩者都成立才會成立";
}
print"\n";
```

這是執行的情況。

```
[root@flash 1-2]# perl logical.pl
不爲假(not false)時爲成立(true)
5>3成立
5等於5成立
只要(3>5)或(5>3)時成立就成立(true)
3!=5稱3不等於5爲true
(3>5)和(5>3)要兩者都成立才會成立
```

範例：and.pl

&&或 and **稱為與邏輯運算子，只有當所有條件都成立時才會回傳真** true**，否則回傳假**。

第二行兩個都為真所以回傳真 true(1)。

第四行因為第二個條件為假 false**，所以回傳假** false(0)。

第八行也是兩個都為真所以回傳 true(1)。

第十四行有三個條件，因為第三個為假 false **所以其運算式結果為假** false**，因此回傳假** false(0)。

```
1 #!/usr/bin/perl
2 print (1 && 1);
3 print "\n";
4 print (1 && 0);
5 print "\n";
6 print (0 && 0);
7 print "\n";
8 print (1 and 1);
9 print "\n";
10 print (1 and 0);
11 print "\n";
12 print (0 and 0);
13 print "\n";
14 print (1 and 1 and 0);
15 print "\n";
```

這是執行的結果。

```
[root@flash 1-2]# perl and.pl
1
0
0
1
0
0
0
```

範例：or.pl

||或 or 稱為或邏輯運算子，只要運算式中一個條件成立就會回傳真 true(1)，**只有當所有的條件都為假** fasle(0)**時，才會回傳假** false(0)。

第六行和第十二行因為兩個條件都為假 false(0)，**所以回傳假** false(0)。

第十六行第一個條件和第二個條件都為假 false(0)，**但是第三個條件為真**(1)，**所以回傳真** true(1)。

```
1 #!/usr/bin/perl
2 print (1 || 1);
3 print "\n";
4 print (1 || 0);
5 print "\n";
6 print (0 || 0);
7 print "\n";
8 print (1 or 1);
9 print "\n";
10 print (1 or 0);
11 print "\n";
12 print (0 or 0);
13 print "\n";
14 print (1 or 1 or 0);
15 print "\n";
```

這是執行的情況。

```
[root@flash 1-2]# perl or.pl
1
1
0
1
1
0
1
```

範例：not.pl
第二行 not true **所以回傳假** false(**空值**)。
第四行 not false **所以回傳真** true(1)。
第六行(!false)**回傳真** true(1)，true **再和** false **作** or **運算，所以回傳真** true(1)。
第八行(!true)**回傳假** false(0)，false **再和** false **作** or **運算，所以回傳假** false(**空值**)。

```
#!/usr/bin/perl
print (!1);
print "\n";
print (!0);
print "\n";
print ((!0)||0);
print "\n";
print (0 || (!1));
print "\n";
```

```
[root@flash 1-2]# perl not1.pl

1
1
```

範例：xor1.pl
第二行因為 true xor true，**不為互斥，所以傳回假** false(**空值**)。
第四行因為 true xor false，**真和假為互斥，所以回傳真** true(1)。
第六行因為 false xor false，**假和假不為互斥，所以傳回假** false(**空值**)。
第十行因為 false xor true **傳回真**(1)，false xor false **傳回假**(**空值**)，**而真**(1)xor **假**(**空值**)**則為互斥，所以回傳真** true(1)。

```
#!/usr/bin/perl
print (1 xor 1);
print "\n";
print (0 xor 1);
print "\n";
print (0 xor 0);
print "\n";
print (1 xor 0);
print "\n";
print ((0 xor 1) xor ( 0 xor 0));
print "\n";
```

```
[root@flash 1-2]# perl xor1.pl

1

1
1
```

1-2-6 執行運算子

反引號｀這個符號為 shell **執行運算子。這個值行運算元不是分號，而其位置在鍵盤的左上方，' '反引號裏面可以執行** shell **的指令，如**`ls-l` **則可以顯示這個目錄底下所有檔案。**

範例 exe.pl

執行運算元可以讓裏面的指令執行 shell **指令。**

```
1 #!/usr/bin/perl
2 $output=`ls -al`;
3 print $output;
4 print "\n";
```

使用`ls-l`**可以顯示此目錄下的檔案。**

```
[root@flash 1-2]# perl exe.pl
總計 148
drwxr-xr-x    2 nobody   nobody       4096 11月  5 10:20 .
drwxrwxrwx   12 root     root         4096 11月  4 20:00 ..
-rwxr--r--    1 nobody   nobody        237 11月  5 10:01 and.pl
-rwxr--r--    1 nobody   nobody        202 11月  5 09:57 and.pl.bak
-rwxr--r--    1 nobody   nobody        155 11月  4 20:11 arith.pl
-rwxr--r--    1 nobody   nobody        143 11月  4 20:11 arith.pl.bak
-rwxr--r--    1 nobody   nobody        124 11月  4 20:30 assign2.pl
```

1-2-7 遞增遞減運算子

遞增(++)和遞減(--)運算子提供一個方便的記號，用來將變數加一或減一，它們經常用來遞增或遞減索引值。根據運算子放置在變數的前後又可分，運算子放置在變數前面稱為前置運算子；運算子放置在變數後面稱為後置運算子。

(1)**遞增運算子**

遞增運算子	**名稱**	**意義**
++$a	**前置遞增**	**變數**$a **先加** 1 **再回傳**
$a++	**後置遞增**	**變數**$a **先回傳再加** 1

範例 increase.pl

第三行為前置遞增；將變數$a **加** 1 **後再輸出，所以值為** 2。

第八行為後置遞增；將變數$a **先輸出再將變數**$a **加** 1。

```
1 #!/usr/bin/perl
2 $a=1;
3 print   ++$a;
4 print   "\n";
5 print   $a;
6 print   "\n";
7 $a=1;
8 print   $a++;
9 print   "\n";
10 print   $a;
11 print   "\n";
```

第五行的變數$a **先加** 1 **為** 2，**再輸出顯示**。

第八行的變數$a++**先輸出顯示為** 1

```
[root@flash 1-2]# perl increase.pl
2
2
1
2
```

(2)**遞減運算子**

遞減運算子就是將變數減 1，**根據遞減運算子的所在位址，可分為前置遞減，與後置遞減**。

遞減運算子	**名稱**	**意義**
--$a	**前置遞減**	**先將變數**$a **減** 1 **再傳回**
$a--	**後置遞減**	**先將變數**$a **傳回再減** 1

範例 decrease.pl

第三行前置遞減運算子，先將變數$a **減** 1 **後再輸出顯示，所以為** 0。

第八行後置遞減運算子，先將變數$a **輸出顯示為** 1，**所以顯示為** 1。

```
1 #!/usr/bin/perl
2 $a=1;
3 print   --$a;
4 print   "\n";
5 print   $a;
6 print   "\n";
7 $a=1;
8 print   $a--;
9 print   "\n";
10 print   $a;
11 print   "\n";
```

第八行後置遞減運算子，先將變數$a **輸出顯示為** 1**，再作遞減的動作，所以顯示為** 1。

```
[root@flash 1-2]# perl decrease.pl
0
0
1
0
```

1-2-8 **字串運算子**

使用’.’**字串連接運算元來連接左右兩邊的字串。**

範例：stringoperators.pl

我們在第六行使用’.’**字串連接運算元來連接左右兩邊的字串。**

```
1 #!/usr/bin/perl
2 $a="你好";
3 $b="大家好";
4 print   $a+$b;
5 print   "\n";
6 print   $a.$b;
7 print   "\n";
```

這是執行的情況。

```
[root@flash 1-2]# perl stringoperators.pl
0
你好大家好
```

1-2-9 **檔案測試運算子**

檔案測試運算子可以用來測試檔案的存在、是否可以讀寫、是否為文字檔、資料夾或字元特定檔。

範例：file.pl

第二行-e **的運算子是測試** STDIN **檔案是否存在。**

第四行-z **的運算子是測試** STDIN **檔案的大小是否為** 0。

```
1 #!/usr/bin/perl
2 print -e STDIN;
3 print "\n";
4 print -z STDIN;
5 print "\n";
```

這是執行的情況。

```
[root@flash 1-2]# perl file.pl
1
1
```

1-2-10 **運算子的優先順序**

我們在作數學運算時，規則是先乘除後加減。例如 1+2*5 **答案大家都知到，** 11。**程式的設計規則也是一樣，可以把運算子看作是乘和加，是有先後順序的處理。**

運算子	**功能**	**用法**
->	**選取某個成員**	Object->member
[ ]	**下標**	Variable[exp]
{ }	**函式呼叫**	Name(expr_list)
++	**後置遞增**	$a++
--	**後置遞減**	$a--
++	**前置遞增**	++$a
--	**後置遞增**	--$a
*	**乘**	2*3
/	**除**	6/2
%	**餘數**	5%3
+	**加**	1+1
-	**減**	2-1
<、<=、>、>=	**小於、小於等於、大於**	3<5、3<=5、5>=2
==、!=	**等於、不等於**	2==2、1!=3
=	**指派運算子**	5=3+2

範例 order.pl

第二行 5 **先乘** 2 **再加** 3，**再加上** 15 **先除以** 5 **在對** 2 **取餘數**。

第四行括號裏面的先作，5 **乘** 2 **為** 10，5 **對** 2 **取餘數為** 1，15 **再除** 1

```
1 #!/usr/bin/perl
2 print 3+5*2+15/5%2;
3 print "\n";
4 print 3+(5*2)+15/(5%2);
5 print "\n";
```

```
[root@flash 1-2]# perl order.pl
14
28
```

1-3 控制結構

控制結構(control structure)控制程式或函數的執行流程，控制結構可將個別的指令組成單一的邏輯單元，有進入點就有出口點。

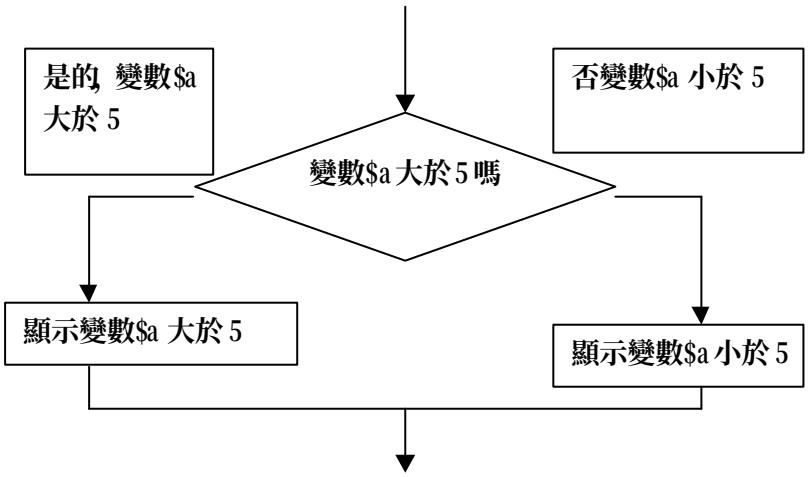
有三種控制結構可以控制執行的流程，分別是循序結構、選取結構、和迴圈結構。一般程式的行程都是一行接著一行的循序執行，這稱為循序結構。而選取結構為可在不同的程式敘述中作選擇。迴圈結構則為重複執行同樣的幾行程式，直到條件滿足而跳出迴圈。

循序結構：

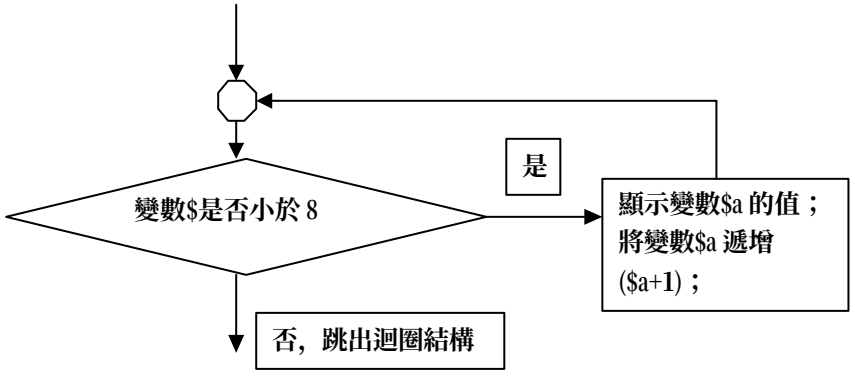
程式碼第一行;
程式碼第二行;
程式碼第三行;
、、、、、、
、、、、、、
、、、、、、

循序結構，就是程式一行一行的由上而下循序執行。

選取結構：

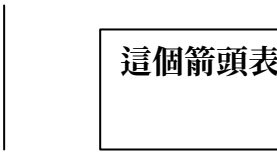


迴圈結構：

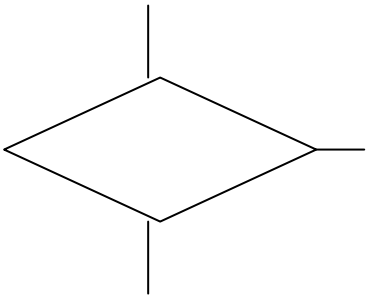


圖形解釋：

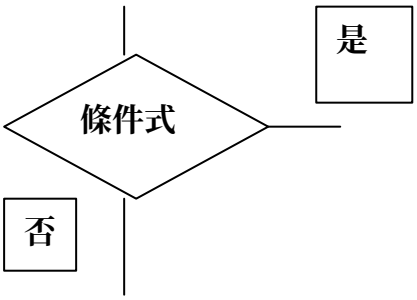
我們在作大型系統或軟體開發時會使用統一模型(UML)語言，因此使用圖示模型來表示程式的資料與程序，我們在這一章將用到許多的模型圖示來表示。



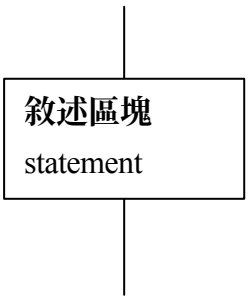
這個箭頭表示程式行進的方向。



這個菱形四邊形，他的進入有一個方向，走出去有兩個方向，



這個菱形四邊形，他的進入口有一個方向，當條件式成立時，會往是的方向走；當條件式不成立時，會往否的方向走。



這個長方形為敘述區塊，是由好幾行程式碼所組成。他的進入口只有一個方向，他的出口也只有一個方向。

範例：control.pl

Perl**程式從** 2 **到** 7 **行，為循序結構，一行一行的由上到下執行。**

Perl**程式從** 9 **到** 15 **行為選取結構，當變數**$a **大於** 5 **時就會顯示變數大於** 5**；當變數**$a **小於** 5 **時就會顯示**$a **小於** 5。

Perl**從第** 16 **行到第** 19 **行為重覆結構。一開始變數**$a=5**，而檢查**$a **小於** 8 **成立，顯示變數**$a **的值** 5**，再執行**$a++(**讓變數**$a+1)。**一值重覆這樣的迴圈，直到變數**$a **小於** 8 **的直不成立而跳出迴圈。**

```
1 #!/usr/bin/perl
2 $a="你好";
3 $b="大家好";
4 print   $a+$b;
5 print   "\n";
6 print   $a.$b;
7 print   "\n";
8 $a=3;
9 if ($a>5){
10 print   "變數\$a大於5";
11 print   "\n";
12 }else{
13 print   "變數\$a小於5";
14 print   "\n";
15 }
16 for($a=5;$a<8;$a++){
17 print   "顯示變數\$a的值".$a;
18 print   "\n";
19 }
```

循序結構所顯示順序，先執行$a+$b **值為** 0**，在顯示字串你好大家好。**

因為變數$a **小於** 5 **所以選取這一行來顯示，這就是選取結構。**

重複執行顯示變數，直到變數$a **不小於** 8 **而跳出迴圈。**

```
[root@flash 1-2]# perl control.pl
0
你好大家好
變數$a小於5
顯示變數$a的值5
顯示變數$a的值6
顯示變數$a的值7
```

1-3-1 **選取結構** if

如何要用 Perl **運算式來表達問句呢？變數**$a **是否大於** 5 **呢**?**在** Perl **中，** if **敘述是主要的選取結構**。

(1)一個選擇的 if 敘述

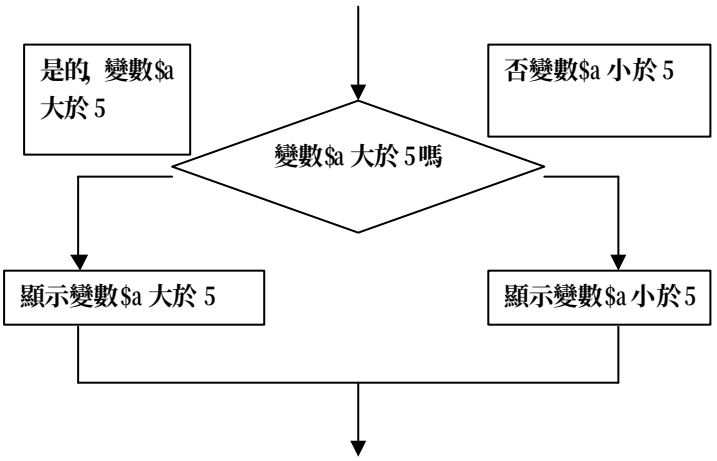
這裡的 condition **指的是條件是否成立；**statement **指的是程式敘述**。

語法：

```
if(condition) {
  statement;
}
```

其中 condition **必須封包在小括號內，可以是個運算式，例如**

```
if($a>5){
print "變數$a 大於 5";
}
```



範例：if.pl

第二行將 8 **給變數**$a**，所以在第三行判別會成立，因此執行第四行**。

```
1 #!/usr/bin/perl
2 $a=8;
3 if ($a>5) {
4 print "變數\$a大於5";
5 }
6 print "\n";
```

```
[root@flash 1-2]# perl if.pl
變數$a大於5
```

範例：if_false.pl

第二行因為 false(0)**所以沒有執行第三行，而到第五行去了。**

```
#!/usr/bin/perl
if(0){
 print("因爲假所以不會列印這一行");
}
print "\n";
```

這是執行的情況。

```
[root@flash 1-3]# perl if_false.pl
```

範例：if_true.pl

第二行因為真 true **所以執行第三行列印"因為真所以列印這一行"。**

```
#!/usr/bin/perl
if(1){
 print("因爲真所以列印這一行");
}
print "\n";
```

```
[root@flash 1-3]# perl if_ture.pl
因爲真所以列印這一行
```

範例：if1.pl

第二行因為真 true(1)**所以執行第三行列印"因為真所以列印這一行"。**

(2)有兩種選擇的 if 敘述

當條件 condition **為真**(1)**時執行第一個敘述**(statement1)，**當條件為假**(0)**時，執行第二個敘述**(statement2)。

語法：

if (condition) {

statement1};

else{

statement2

}

範例：if2.pl
因為變數$a **為** 3，**所以第三行**$a>5 **為假，所以執行第六行。**

```
1 #!/usr/bin/perl
2 $a=3;
3 if ($a>5){
4 print "變數\$a大於5";
5 }else{
6 print "變數\$a小於5";
7 }
8 print "\n";
```

這是執行的結果。

```
[root@flash 1-3]# perl if2.pl
變數$a小於5
```

範例：if2_false.php
因為變數$a **為** 8，**所以第三行**$a>5 **為真，所以執行第四行**"**變數**\$a **大於** 5"。

```
1 #!/usr/bin/perl
2 $a=8;
3 if ($a>5){
4 print "變數\$a大於5";
5 }else{
6 print "變數\$a小於5";
7 }
8 print "\n";
```

這是執行的情況。

```
[root@flash 1-3]# perl if2_false.pl
變數$a大於5
```

(3) 有複合敘述的 if 敘述
在真或假的判別後，就會執行相關的程式敘述，而經常我們是用大括號包住這些複合敘述。複合敘述就是由好幾行程式碼所組成的。
注意事項：條件 condition **是由小括號 () 包起來的，敘** statement **是由大括號 { } 包起來。當條件** condition **為真** true **時，就會執行** true statements 1；**當條件** condition **為假** false **時，就會執行** false statement 2。

```
If(condition){
  true statements 1;
}else{
  false statements 2;
}
```

範例：copoun.php

因為由第三行得知變數$a **小於** 5 **時，就會執行第七行和第八行。**

```
1 #!/usr/bin/perl
2 $a=3;
3 if ($a>5){
4   print   "變數\$a大於5";
5   print   "\n";
6 }else{
7   print   "變數\$a小於5";
8   print   "\n";
9 }
10 print "\n";
```

```
[root@flash 1-3]# perl copoun.pl
變數$a小於5
```

範例：even_var.pl

第二行因為變數$var **為** 9**，是奇數，所以第三行** 9 **取** 2 **的餘數為** 1**，和** 0 **不相等，所以條件的結果為** false**，所以執行第** 8 **行。**

第十一行因為變數$var **為** 8**，是偶數，所以第十二行** 8 **取** 2 **的餘數為** 0**，和** 0 **相等，所以條件的結果為** true**，所以執行第十六行**"**變數**$var1 **是偶數**"**。**

```
1 #!/usr/bin/perl
2 $var=9;
3 if($var %2 ==0){
4   if($var !=0){
5      print ("變數\$var是偶數\n");
6   }
7 }else{
8  print("變數\$var是奇數\n");
9 }
10 $var1=8;
11 if($var1 %2 ==0){
12   if($var1 !=0){
13      print ("變數\$var1是偶數\n");
14   }
15 }else{
16  print("變數\$var1是奇數\n");
17 }
```

```
[root@flash 1-3]# perl even_var.pl
變數$var是奇數
變數$var1是偶數
```

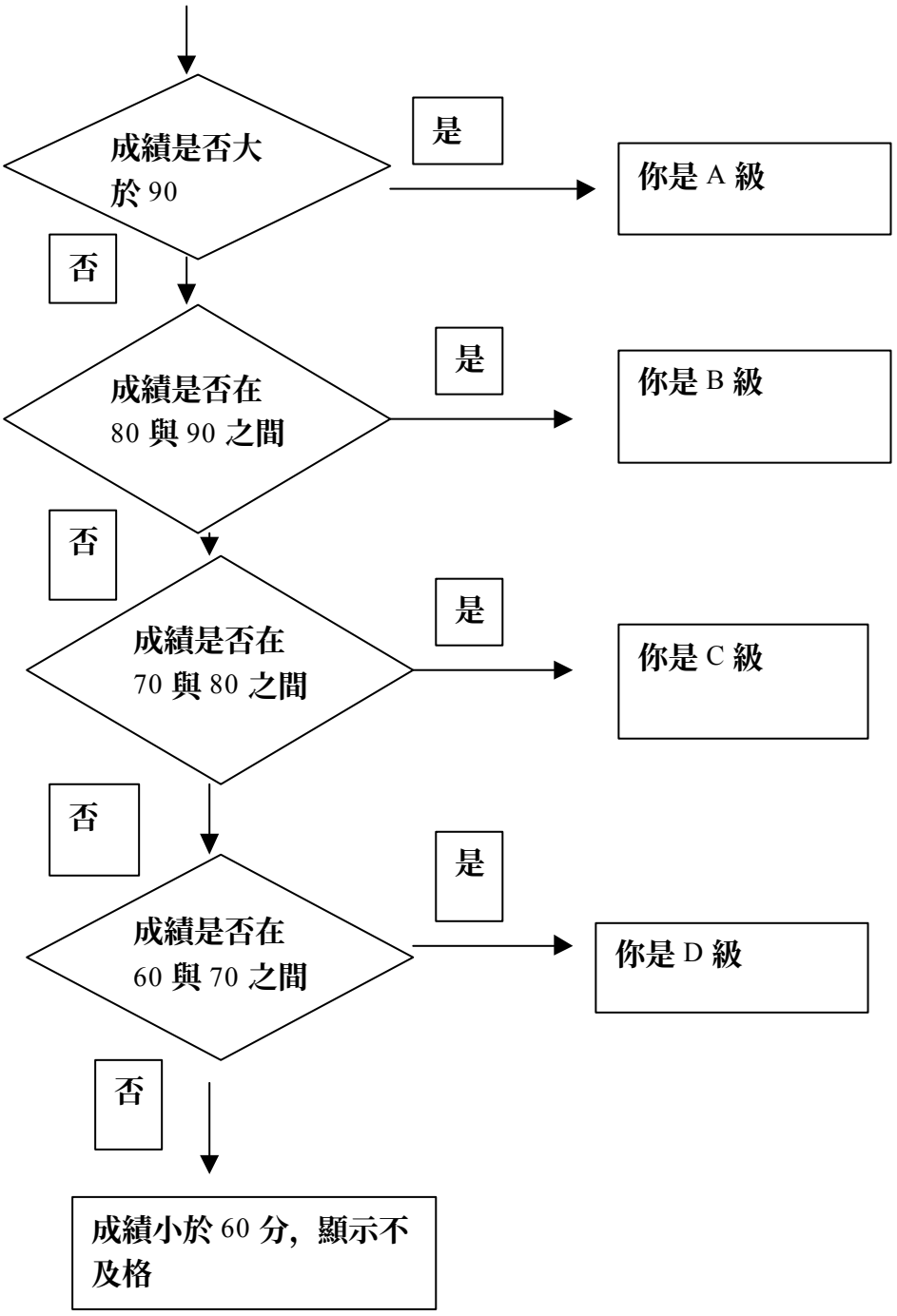
範例：zero.pl
第二行將 0 **給變數**$var。
第三行因為 0 **取** 2 **的餘數為** 0 **和** 0 **相等，所以執行第四行到第六行**。
第四行因為 0!=0 **為** false **假，所以不會執行第五行**。
If**判別式的敘述裏面還有** if **的子判別式，我們稱為巢狀** IF **判別**。

```
1 #!/usr/bin/perl
2 $var=0;
3 if($var %2 ==0){
4   if($var !=0){
5      print ("變數\$var是偶數<br>");
6   }
7 }else{
8  print("變數\$var是奇數<br>");
9 }
```

```
[root@flash 1-3]# perl zero.pl
[root@flash 1-3]#
```

(4)巢狀的 if 敘述

我們已經看過 if 敘述 1 個或 2 個的選擇，在此要用巢狀 if 敘述來撰寫多重選擇決策。例如我們的成績評等，90 到 100 分的為 A 級，80 到 89 的為 B 級，70 到 79 的為 C 級，60 到 69 的為 D 級，59 分以下的為 F 級。如果我有一位學生分數為 82 分，他的分數在 80 到 89 分之間，則他的等級為 B 級。在這個例子中有 5 個等級來做選擇，就叫做多重選擇。



多重的選擇決策

語法：

```
if(condition1){
   statement1;
}elsif(condition2){
   statement2;
}elsif(condition3){
   statement3;
}elsif(condition4){
   statement4;
. . . . .
. . . . .
. . . . .
}else{
statement n;
}
```

範例：netif.pl

多重選擇 i 就會選擇哪一個條件適合來執行。

因為第二行$score=95 **大於** 90 **所以會執行第四行。**

```
#!/usr/bin/perl
$score=95;
if ($score > 90){
 print   "你是A級";
} elsif (($score > 80) && ($score <= 89)){
 print   "你是B級";
} elsif ($score > 70 && $score <= 79){
 print   "你是C級";
} elsif ($score > 60 && $score <= 69){
 print   "你是D級";
} else {
 print   "不及格啦F級";
}
print "\n";
```

```
[root@flash 1-3]# perl netif.pl
你是A級
```

範例：elseif_day.pl

這是 perl **的程式語法，這是多種選擇的條件**。

```
#!/usr/bin/perl
$day=0;
if($day==0){
  print  "禮拜日";
}elsif($day==1){
  print  "禮拜一";
}elsif($day==2){
  print  "禮拜二";
}elsif($day==3){
  print  "禮拜三";
}elsif($day==4){
  print  "禮拜四";
}elsif($day==5){
  print  "禮拜五";
}elsif($day==6){
  print  "禮拜六";
}else{
  print  "請輸入數字";
}
print "\n";
```

因為$day=0 **所以執行第四行禮拜日**。

```
[root@flash 1-3]# perl elseif_day.pl
禮拜日
```

(4)unless 語法

unless **是** if **的相反結構**。

這裡的 condition **指的是條件是否成立，當條件不成立時就會執行掛號內的** statement **敘述，** statement **指的是程式敘述**。

語法：

```
unless(condition) {
  statement;
}
```

其中 condition **必須封包在小括號內，可以是個運算式，例如**

```
unless($a>5){
echo “變數$a 大於 5”;
}
```

範例：unless.pl

因為第三行的條件成立(1)**，此時就不會執行掛號內第四行的** statement **敘述**。

```
1 #!/usr/bin/perl
2 $a=8;
3 unless ($a>5) {
4 print "變數\$a大於5";
5 }
6 print "\n";
```

這個執行結果和 if **的執行結果相反**。

```
[root@flash 1-3]# perl unless.pl
```

範例：unless_false.pl

unless **因為第二行** false(0)**所以執行第三行**。

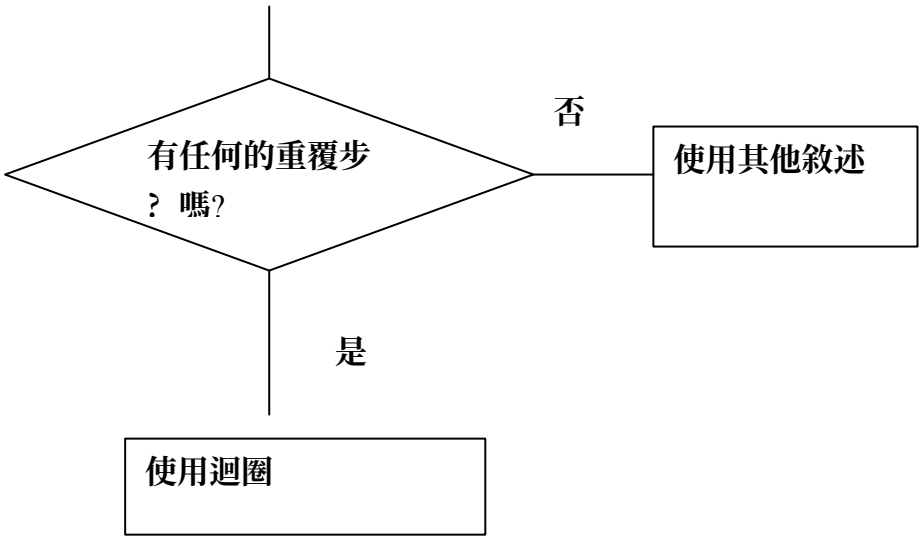
```
1 #!/usr/bin/perl
2 unless(0){
3  print("因爲假所以會列印這一行");
4 }
5 print "\n";
```

這是執行的情況。

```
[root@flash 1-3]# perl unless_false.pl
因爲假所以會列印這一行
```

1-3-2 **迴圈結構**

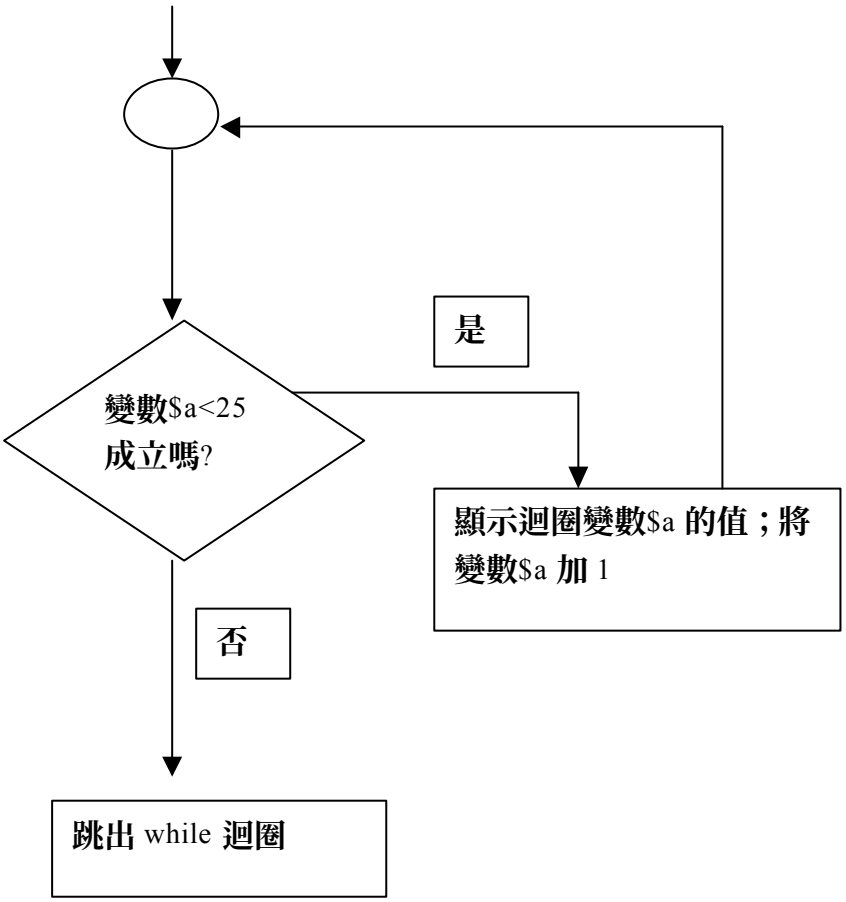
選取結構和循序結構，都只執行程式敘述一次，如果我們要讓同一行程式重複執行好幾遍則要用迴圈敘述。迴圈敘述可以重複執行某一段程式好幾遍，直到條件的不成立才跳出這個迴圈。shell **的迴圈敘述有** WHILE **迴圈和** FOR **迴圈。**



(1)while **迴圈**

在 if **敘述中, 條件後的敘述只執行一次, 而在** while **敘述中, 則可執行一次以上。**

While **敘述的程序圖形中**



語法：

```
while(condition){
   statement;
}
```

condition**：重覆迴圈的控制條件**

statement**：迴圈主體**

範例：while_false.pl

因為第二行的 while **迴圈的條件為** false，**所以不會執行第三行的敘述**。

```
1 #!/usr/bin/perl
2 while(0){
3  print "這個while迴圈不會執行";
4 }
```

```
[root@flash 1-3]# perl while_false.pl
[root@flash 1-3]#
```

範例：while_true.pl

因為第二行的 while **迴圈的條件為** true，**所以會一直執行第三行的敘述**。

```
1 #!/usr/bin/perl
2 while(true){
3  print "這個while迴圈會永久執行\n";
4 }
```

這個會一直執行。

```
這個while迴圈會永久執行
這個while迴圈會永久執行
這個while迴圈會永久執行
這個while迴圈會永久執行
這個while迴圈會永久執行
這個while迴圈會永久執行
這個while迴圈會永久執行
這個while迴圈會永久執行
這個while迴圈會永久執行
這個while迴圈會永久執行
這個while迴圈會永久執行
這個while迴圈會永久執行
這個while迴圈會永久執行
```

範例 while.pl

在這個 while **迴圈中，每執行一次迴圈就會讓變數**$a **的值加** 1 **直到變數**$a **大於等於** 25。

第六行變數$a **後置遞增**。

```
1 #!/usr/bin/perl
2 $a=20;
3 while($a<25){
4   print   "迴圈控制的變數\$a值爲".$a;
5   print   "\n";
6   $a++;
7 }
```

While **迴圈讓第四行程式敘述重覆** 5 **次**。

```
[root@flash 1-3]# perl while.pl
迴圈控制的變數$a值爲20
迴圈控制的變數$a值爲21
迴圈控制的變數$a值爲22
迴圈控制的變數$a值爲23
迴圈控制的變數$a值爲24
```

(2)do…while 迴圈

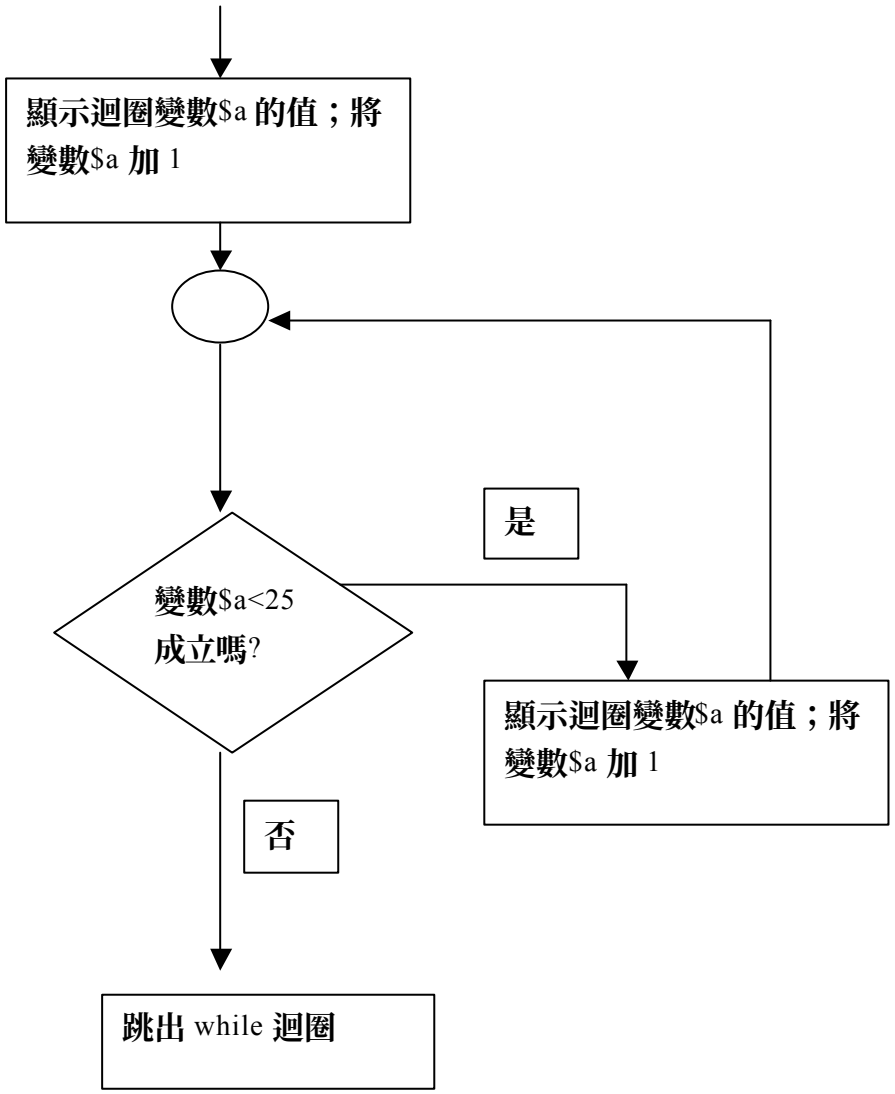
while **敘述與** for **敘述皆在執行迴圈主體前，先計算迴圈重覆條件。在大部分的情況，這是可以使迴圈避免不必要的錯誤，但有時候需先執行一次迴圈主體在測試條件，而** php **提供了我們這** do…..while **迴圈**。

語法：

```
do {
   statement;
}while(condition)
```

do….while **迴圈會先執行** statement **一次再判別條件是否成立**。

在圖中顯示迴圈變數$a 的值;將變數$a 加 1，這是 do….while 的迴圈主體 statement，只要條件(condition)變數$a<25 就會執行迴圈。



範例：dowhile.pl

Do…while 會先做一次迴圈主體。

```
#!/usr/bin/perl
$a=20;
do{
  $a++;
  print   "迴圈控制的變數\$a值爲".$a;
  print   "\n";
}while($a<25);
```

因為已經先做一次迴圈主體，所以變數$a 先加一成為 21 這就是 do…while 迴圈的和 while 不同之處。

```
[root@flash 1-3]# perl dowhile.pl
迴圈控制的變數$a值爲21
迴圈控制的變數$a值爲22
迴圈控制的變數$a值爲23
迴圈控制的變數$a值爲24
迴圈控制的變數$a值爲25
```

範例：do_while_counter.pl

Do…while 會先做一次迴圈主體。第二行$counter 變數是 55，因此在作 do while 迴圈時，會依序執行第四行和第五行。

第四行第一次會顯示資料為 55。

第五行會將變數$counter 的值加 1，第一次將 55 加 1 而成 56。

然後會執行第六行的 while 判別式，因為 56 小於等於 58 為 true，所以會再次執行 do while 迴圈，執到$counter 變數的值大於 58 而跳出。

```
1 #!/usr/bin/perl
2 $counter=55;
3 do{
4   print  "\$counter變數的值是$counter\n";
5   $counter=$counter+1;
6 }while ($counter <=58)
```

第一次變數$counter 的數值為 55，也顯示在第一行。

```
[root@flash 1-3]# perl do_while_counter.pl
$counter變數的值是55
$counter變數的值是56
$counter變數的值是57
$counter變數的值是58
```

(3)for 迴圈

Perl **提供** for **敘述作為迴圈，除了由大括號所包的迴圈主體敘述**(statement)**，還有迴圈的條件結構**。

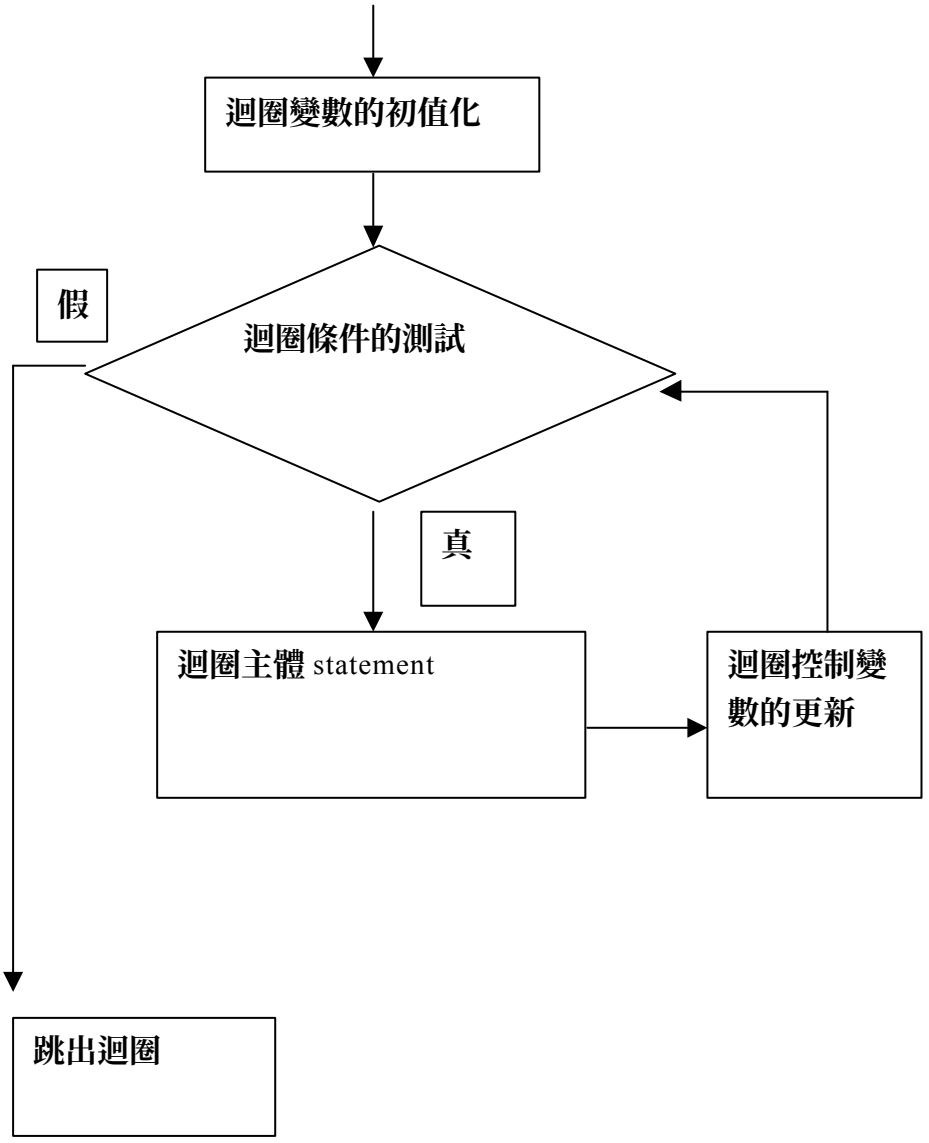
語法：

for(**起始運算式**;**測試條件**;**更新運算式**){

迴圈主體敘述;

}

for **敘述一開始會先執行迴圈變數的初值化**(initialization expression)**，然後執行迴圈條件的測試部分，若條件為真，則執行迴圈主體**(statement)**，若條件為假則不執行迴圈主體，然後再執行迴圈控制變數的更新**(更新運算式)**，執行後在測試迴圈條件的測試，若為真則再執行迴圈主體，直到測試條件不成立而跳出**。



下面的 for 迴圈可以用下列同等的 while 迴圈來表示。

這是 for 迴圈：

```
for(起始運算式;測試條件;更新運算式){
  迴圈主體敘述;
}
```

這是 while 迴圈：

```
起始運算式;
while(測試條件){
   迴圈主體敘述;
   更新條件;
}
```

下面的 for 迴圈沒有初始值，也沒有測試條件，它就和 while(true)的迴圈相同。

```
for(;;){
  迴圈主體敘述;
}
while(true){
   迴圈主體敘述;
}
```

範例 for1.pl

讓變數$i 一開始的值為 1，只要變數$i 小於 5，就會執行顯示變數$i 的迴圈主體，再將變數$i 加 1。

```
#!/usr/bin/perl
for ($i = 1; $i <= 5; $i++) {
    print $i;
    print "\n";
}
```

這是執行的情況。

```
[root@flash 1-3]# perl for1.pl
1
2
3
4
5
```

範例：for2.pl

這題將迴圈條件的測試部分，由迴圈主體的選取結構 if **代替。當執行到第四行的** last **時，會跳出這個迴圈。**

```
1 #!/usr/bin/perl
2 for ($i = 1;;$i++) {
3     if ($i > 10) {
4         last;
5     }
6     print $i;
7     print "\n";
8 }
```

這是執行的情況。

```
[root@flash 1-3]# perl for2.pl
1
2
3
4
5
6
7
8
9
10
```

範例 for3.pl

這個 for **迴圈把變數**$I **的初始值寫再** for **迴圈外面，而將迴圈條件測試部分寫再** for **迴圈裡面，而且也把迴圈控制的更新寫再迴圈主體。**

```
 1 #!/usr/bin/perl
 2 $i = 1;
 3 for (;;) {
 4     if ($i > 10) {
 5         last;
 6     }
 7     print $i;
 8     print "\n";
 9     $i++;
10 }
```

一樣可以顯示變數$I 的值

```
[root@flash 1-3]# perl for3.pl
1
2
3
4
5
6
7
8
9
10
```

範例：for5.pl

我們也可以在 for **迴圈中使用逗號來增加好幾個子句。在測試條件中，是以最後一個子句的測試為主，當最後一個子句測試通過，** for **迴圈就繼續執行，當最後一個子句測試不通過，其** for **迴圈就不執行。**

變數$z **會執行變數**$z **為** 0 **時，變數**$z **為** 5 **時，變數**$z **為** 10 **時、變數**$z **為** 15 **時，變數**$z **為** 20**。當變數**$z **為** 20 **時，更新運算式將變數**$z **加到** 25**，因此再執行測試條件時，將不通過而跳出** for **迴圈。**

```
#!/usr/bin/perl
for($x=0,$y=0,$z=0;
     $x<=20,$y<=20,$z<=20;
     $x=$x+5,$y=$y+5,$z=$z+5){
  print "變數\$x=$x,變數\$y=$y,變數\$z=$z\n";
}
```

這是執行的情況。

```
[root@flash 1-3]# perl for5.pl
變數$x=0,變數$y=0,變數$z=0
變數$x=5,變數$y=5,變數$z=5
變數$x=10,變數$y=10,變數$z=10
變數$x=15,變數$y=15,變數$z=15
變數$x=20,變數$y=20,變數$z=20
```

範例：for6.pl

我們也可以在 for **迴圈中使用逗號來增加好幾個子句。在測試條件中，是以最後一個子句的測試為主，當最後一個子句測試通過，** for **迴圈就繼續執行，當最後一個子句測試不通過，其** for **迴圈就不執行。**

變數$z **會執行變數**$z **為** 0 **時，變數**$z **為** 0 **時，變數**$z **為** 10 **時、變數**$z **為** 10 **時，變數**$z **為** 20 **時。當變數**$z **為** 20 **時，更新運算式將變數**$z **加到** 30**，因此再執行測試條件時，將不通過而跳出** for **迴圈。**

```
1 #!/usr/bin/perl
2 for($x=0,$y=0,$z=0;
3      $x<=20,$y<=20,$z<=20;
4      $x=$x+5,$y=$y+10,$z=$z+10){
5   print "變數\$x=$x,變數\$y=$y,變數\$z=$z\n";
6 }
```

多個子句時，測試條件以最後一個子句為主。

```
[root@flash 1-3]# perl for6.pl
變數$x=0,變數$y=0,變數$z=0
變數$x=5,變數$y=10,變數$z=10
變數$x=10,變數$y=20,變數$z=20
```

範例：m.php

這是巢狀迴圈也是兩層的 for **迴圈所組成，他總共執行** 81 **次** print **輸出，讓我們可以看到** 99 **乘法，也就是說重複執行第** 7 **行** 81 **次。第一個** for **迴圈為第** 6 **到第** 8 **行，外圈的** for **迴圈為第** 3 **到第** 10 **行。**

第七行使用字串連接運算子就可以把字串聯起來了。

```
1 #!/usr/bin/perl
2
3 for($a=1;$a<=9;$a++)
4 {
5   print  "\n";
6   for($b=1;$b<=9;$b++){
7     print  $a."*".$b."=".$a*$b."   ";
8   }
9   print  "\n";
10 }
```

這是 99 乘法。

```
[root@flash 1-3]# perl m.pl
1*1=1   1*2=2   1*3=3   1*4=4   1*5=5   1*6=6   1*7=7   1*8=8   1*9=9
2*1=2   2*2=4   2*3=6   2*4=8   2*5=10  2*6=12  2*7=14  2*8=16  2*9=18
3*1=3   3*2=6   3*3=9   3*4=12  3*5=15  3*6=18  3*7=21  3*8=24  3*9=27
4*1=4   4*2=8   4*3=12  4*4=16  4*5=20  4*6=24  4*7=28  4*8=32  4*9=36
5*1=5   5*2=10  5*3=15  5*4=20  5*5=25  5*6=30  5*7=35  5*8=40  5*9=45
6*1=6   6*2=12  6*3=18  6*4=24  6*5=30  6*6=36  6*7=42  6*8=48  6*9=54
7*1=7   7*2=14  7*3=21  7*4=28  7*5=35  7*6=42  7*7=49  7*8=56  7*9=63
8*1=8   8*2=16  8*3=24  8*4=32  8*5=40  8*6=48  8*7=56  8*8=64  8*9=72
9*1=9   9*2=18  9*3=27  9*4=36  9*5=45  9*6=54  9*7=63  9*8=72  9*9=81
```

(4)last 敘述

當我們要離開迴圈時，主要是條件變成 fasle(0)**時就可以跳出迴圈。我們也可以使用** last **指令或** continue **指令來作出改變迴圈結構流程的方法。**

last **指令可以讓我們離開最裏面的迴圈。**

Continue **區塊接在** while、until **迴圈區塊之後，其作用為在每一次迴圈後執行** continue **區塊的敘述。**

我們利用 last **可以跳出** for **迴圈。**

範例：for_last.pl

第二行 for **迴圈的變數**$var **測試是當變數**$var **小於** 10 **時，就會執行第四到第七行，執行完後再將變數**$var **加** 1。

但是，當執行到變數$var **為** 3 **時，因為第四行的判別式為** false(0)**，所以執行第五行，而跳出** for **迴圈。**

```
1 #!/usr/bin/perl
2 for($var=2;$var<10;$var++)
3 {
4   if($var%2!=0){
5     last ;
6   }
7   print $var."\n";
8 }
```

只有執行第一次變數$var **為** 2 **的迴圈。**

```
[root@flash 1-3]# perl for_last.pl
2
```

範例：prime_last.pl

這是求質數的演算方法。我們要求從第一個數($first)**到最後一個數**($last)**之間的所有質數。質數的定義是只有除了可以被** 1 **或自己整除的數值。質數的特性是判別是否可以被** 1 **到該數值開根號來整除，如果是就不是質數。**

因為第四行的 while **迴圈為** true(1)**，所以會一直執行執到第六行判別為** true **時，再執行第七行而跳出最外層的** while **迴圈。**

因為第九行的 while **迴圈為** true(1)**，所以會一直執行直到第十行判別為** true **時，執行第十一行與第十二行而跳出最內層的** while **迴圈，或者執行到第十四行判別為** true **時，執行第十五行而跳出最內層的** while **迴圈。**

```
1 #!/usr/bin/perl
2 $first=2;
3 $last=199;
4 while(1){
5  $div=2;
6  if($first > $last){
7    last ;
8  }
9   while(1){
10   if($div > sqrt($first)){
11       print  "$first";
12       last;
13   }
14   if($first % $div==0){
15       last;
16   }
17       $div=$div+1;
18   }
19  $first=$first+1;
20 }
21 print "\n";
```

這是從 2 **到** 199 **的質數。**

```
[root@flash 1-3]# perl prime_last.pl
2357111317192329313741434753596167717379838997101103107109113127131137139149151157163167173179181191193197199
```

1-4 副程式與函數

結構化程式設計，在程式的模組化和由上而下的程式設計。在程式設計時，我們常將較大的程式分成數個較小的功能，每個小功能都能夠很容易的分析，並且撰寫。我們可以將這些小功能寫成副程式。而且我們可以將程式中重複的程式寫成一個副程式，當我們需要時再呼叫這副程式，這樣可以易於維護程式碼。在 Perl 中內建的數學函式允許程式設計師執行一般的數學運算。而我們也可以自己定義函數，這就稱為使用者自訂函數，或稱為使用者自訂副程式。

語法：

```
sub 名子(參數 1,參數 2,參數 3,,,,,)
{
 敘述區塊(statement);
}
```

當我們要使用函數時，只要呼叫函數就可以了。

範例：sqrt.pl

第二行使用 Perl 中內建的數學函式 sqrt( )回傳將所輸入的數值開根號。

```
1 #!/usr/bin/perl
2 print sqrt(16);
3 print "\n";
```

```
[root@flash 1-4]# perl sqrt.pl
4
```

範例：math.pl

第三行使用 Perl 中內建的開根號數學函式 sqrt();。

第五行使用 Perl 中內建的 exp( )函數。

第七行將 5 開 log。

```
1 #!/usr/bin/perl
2 $a=-2;
3 print sqrt(3);
4 print "\n";
5 print exp($a);
6 print "\n";
7 print log(5);
8 print "\n"
```

```
[root@flash 1-4]# perl math.pl
1.73205080756888
0.135335283236613
1.6094379124341
```

範例：sub.pl

我們在第四到六行定義 subroutine1( )**副程式，我們在第八到第十一行定義** subroutine2( )**副程式。我們在第二行呼叫** subroutine1( )**副程式，這樣它就會執行第四到第七行。我們在第三行呼叫** subroutin2(**副程式，這樣它就會執行第八到第十一行**。

```
1 #!/usr/bin/perl
2 subroutine1();
3 subroutine2();
4 sub subroutine1
5 {
6   print "呼叫 subroutine1\n";
7 }
8 sub subroutine2
9 {
10  print "呼叫 subroutine2\n";
11 }
```

這是執行的情況。

```
[root@flash 1-4]# perl sub.pl
呼叫 subroutine1
呼叫 subroutine2
```

1-4-1 參數串列

許多程式都須要傳遞參數到函數中來幫助函數執行它們的任務。傳送給函數的串列參數存在於陣列變數@_中。

範例：argument.pl

副程式能夠接收任何數量的參數。

在第二行呼叫副程式 displayArguments 和帶進去六個參數。

在第六行會列印出所有的參數，我們使用陣列變數@_來輸出串列參數。

第八行的$_[$i]會存取陣列變數@_的第 i 個元素。

```
1 #!/usr/bin/perl
2 displayArguments( "佳諺", "界良", 2, 15, 73, 2.79 );
3
4 sub displayArguments
5 {
6    print "All arguments: @_\n";
7    for ( $i = 0; $i < @_; ++$i ) {
8       print "Argument $i: $_[ $i ]\n";
9    }
10 }
```

這是執行的情況。

```
[root@flash 1-4]# perl argument.pl
All arguments: 佳諺 界良 2 15 73 2.79
Argument 0: 佳諺
Argument 1: 界良
Argument 2: 2
Argument 3: 15
Argument 4: 73
Argument 5: 2.79
```

1-4-2 回傳值

當副程式完成它的工作, 資料會被傳回呼叫的副程式, 我們可以使用關鍵字 return 來回傳。

範例：return.pl

第六行到第十行定義副程式 square()。

第二行到第四行的 for 迴圈十次。

第三行是呼叫 square()副程式,

第八行的 shift 函數讓陣列@_移除第一個元素並且將該元素給變數$value。

第九行的 return 會回傳陣列元素值的平方。

```
1 #!/usr/bin/perl
2 for ( 1 .. 10 ) {
3     print square( $_ ), " ";
4 }
5 print "\n";
6 sub square
7 {
8     $value = shift;
9     return $value ** 2;
10 }
```

這是執行的情況。

```
[root@flash 1-4]# perl return.pl
1 4 9 16 25 36 49 64 81 100
```

範例：hello.pl

Perl 提供 wantarray 函數，當副程式是由串列內容被呼叫，則會傳回 true，如果副程式是由純量被呼叫，則會傳回 false。

第二行使用回傳 scalarOrList( )副程式的值來初始化@array。

一個陣列的初始化就是串列內容，因此第八行的 wantarray()副程式會回傳 true，因此會執行第九行。

第三行的$”=”\n”會將換行符號加到陣列的元素間，因此每列印一個陣列元素就會換行。

第五行會傳回純量的內容，因此 wantarray(副程式會回傳 false，因此會執行第十二行的”大家好”。

```
1 #!/usr/bin/perl
2 @array = scalarOrList();
3 $" = "\n";
4 print "Returned:\n@array\n";
5 print "\nReturned: " . scalarOrList();
6 sub scalarOrList
7 {
8    if ( wantarray() ) {
9       return 'this', 'is', 'a', 'list', 'of', 'strings';
10   }
11   else {
12      return '大家好';
13   }
14 }
15 print "\n";
```

這是執行的情況。

```
[root@flash 1-4]# perl scalarorlist.pl
Returned:
this
is
a
list
of
strings

Returned: 大家好
```

1-4-3 呼叫副程式的方法

我們除了直接使用副程式的名子來呼叫，Perl 也使用型態識別子來分辯別型態。我們在副程式名子前面加一個&。例如我們使用&subroutine1 來呼叫 subroutine1 副程式。當呼叫沒有參數的副程式時使用這個語法，將接收到呼叫的@_變數。假如任何明確的參數傳到副程式，在這語法中括號( )總是需要的。

語法：

&副程式名子( )

範例：callsub.pl

我們在第二行到第五行定義 definedBeforeWithoutArguments 副程式。

我們在第六行到第九行定義 definedBeforeWithArguments 副程式，這是會列印出陣列元素。

第十二行和第十三行呼叫副程式使用&和括號( )。

第十五行和第十六行呼叫副程式使用括號( )，這兩個副程式正確的執行。

第十八行呼叫 definedBeforeWithoutArguments 和&符號，但是沒有( )括號。&definedBeforeWithoutArguments 副程式不允許在沒有括號( )的情況下加入參數，這樣會發生錯誤。

第二十二行和第二十三行呼叫副程式，同時不使用&符號和( )括號。這個語法只能在副程式先定義前才能呼叫。

接下來是作同樣的呼叫副程式作測試，但是和前面不同的是副程式的定義是在呼叫程式的後面。

第二十六行和第二十七行呼叫副程式使用&和括號( )，當只有括號被使用時，&符號才能使用，這樣語法才會正確。

第二十九行和第三十行呼叫副程式使用括號( )，這兩個副程式正確的執行。

第三十五行到第三十九行，當同時沒有使用括號( )也沒有&符號的副程式時，因為沒有參數，所以不會產生任何作用，因為沒有內容幫助 Perl 決定辨識目標。

我們使用 print 的方式說明錯誤的發生 generates a syntax error 為發生語法的錯誤，causes no action 為不會產生任何動作。

```
#!usr/bin/perl
sub definedBeforeWithoutArguments
{
   print "definedBeforeWithoutArguments\n";
}
sub definedBeforeWithArguments
{
   print "definedBeforeWithArguments: @_\n";
}
print "副程式在使用前先定義\n";
print "Using & and ():\n";
&definedBeforeWithoutArguments();
&definedBeforeWithArguments( 1, 2, 3 );
print "\nUsing only ():\n";
definedBeforeWithoutArguments();
definedBeforeWithArguments( 1, 2, 3 );
print "\nUsing only &:\n";
&definedBeforeWithoutArguments;
print "\"&definedBeforeWithArguments 1, 2, 3\"",
      " 這個副程式呼叫產生語法錯誤\n";
print "\nUsing bareword:\n";
definedBeforeWithoutArguments;
definedBeforeWithArguments 1, 2, 3;
print "將副程式寫在後面\n";
print "Using & and ():\n";
&definedAfterWithoutArguments();
&definedAfterWithArguments( 1, 2, 3 );
print "\nUsing only ():\n";
definedAfterWithoutArguments();
definedAfterWithArguments( 1, 2, 3 );
print "\nUsing only &:\n";
&definedAfterWithoutArguments;
print "\"&definedAfterWithArguments 1, 2, 3\"",
      " 這個副程式呼叫產生語法錯誤\n";
print "\nUsing bareword:\n";
definedAfterWithoutArguments;
print "\"definedAfterWithoutArguments\"這個副程式沒有任何作用\n";
print "\"definedAfterWithArguments 1, 2, 3\"",
      " 這個副程式呼叫產生語法錯誤\n";
sub definedAfterWithoutArguments
{
   print "definedAfterWithoutArguments\n";
}
sub definedAfterWithArguments
{
   print "definedAfterWithArguments: @_\n";
}
```

這是執行的情況。我們使用 print 的方式說明錯誤的發生 generates a syntax error 為發生語法的錯誤，causes no action 為不會產生任何動作。如果將錯誤語法的副程式呼叫執行，則會產生語法錯誤。我們直接使用 print 列印出來，這是為了方便講解。

```
[root@flash 1-4]# perl callsub.pl
副程式在使用前先定義
Using & and ():
definedBeforeWithoutArguments
definedBeforeWithArguments: 1 2 3

Using only ():
definedBeforeWithoutArguments
definedBeforeWithArguments: 1 2 3

Using only &:
definedBeforeWithoutArguments
"&definedBeforeWithArguments 1, 2, 3" 這個副程式呼叫產生語法錯誤

Using bareword:
definedBeforeWithoutArguments
definedBeforeWithArguments: 1 2 3
將副程式寫在後面
Using & and ():
definedAfterWithoutArguments
definedAfterWithArguments: 1 2 3

Using only ():
definedAfterWithoutArguments
definedAfterWithArguments: 1 2 3

Using only &:
definedAfterWithoutArguments
"&definedAfterWithArguments 1, 2, 3" 這個副程式呼叫產生語法錯誤

Using bareword:
"definedAfterWithoutArguments"這個副程式沒有任何作用
"definedAfterWithArguments 1, 2, 3" 這個副程式呼叫產生語法錯誤
```

1-4-4 隨機函數

我們在前面介紹了使用者自訂的副程式，我們 Perl 也有內部就已經設定好的函數給我們使用，在這裏我們介紹 rand( )函數。rand( )函數會產生在 0 到 1 的浮點數。假如 rand( )函數產生 0 到 1 的任何浮點數，則產生任何它們的機率都會相同。我們可以設定 rand( )函數的參數，來選擇性的傳回特定範圍的數值。我們如果輸入 5 的參數到 rand( )函數中，rand(5)，則它的範圍是從 0 到 5。我們可以將 rand( )隨機函數放到 int( )整數函數中，則這樣就會傳回整數的隨機數。

範例：rand.pl

第三行的 rand(5)會產生 0 到 5 的隨機數。

```
1 #!usr/bin/perl
2 print "由random函數所產生的隨機數:\n";
3 print rand(5);
4 print "\n";
```

我們每一次執行 rand(5)函數，它就會出現 0 到 5 的隨機數。

```
[root@flash 1-4]# perl rand.pl
由random函數所產生的隨機數:
1.095496554994 52
[root@flash 1-4]# perl rand.pl
由random函數所產生的隨機數:
2.14814920713708
[root@flash 1-4]# perl rand.pl
由random函數所產生的隨機數:
2.02415877407931
[root@flash 1-4]# perl rand.pl
由random函數所產生的隨機數:
1.903162080930 11
[root@flash 1-4]# perl rand.pl
由random函數所產生的隨機數:
4.262435556524 92
```

範例：random.pl

第四行的 rand( )**函數會出現** 0 **到** 1 **的隨機數**。

第八行的 rand(100)**函數會出現** 0 **到** 100 **的隨機數**。

第十二行的 int(rand(6))**會將隨機數轉成整數**。rand(6)**會出現** 0 **到** 6 **的隨機數，然後再使用** int( )**函數，將它轉成整數。將它加** 1 **後，就會出現** 1 **到** 6 **的隨機整數**。

```
1  #!usr/bin/perl
2  print "由random函數所產生的隨機數:\n";
3  for ( 1 .. 3 ) {
4     print "    ", rand(), "\n";
5  }
6  print "\n由rand( 100 )函數所產生的隨機數:\n";
7  for ( 1 .. 3 ) {
8     print "    ", rand( 100 ), "\n";
9  }
10 print "\n1 + int( rand( 6 ) )的數值:\n";
11 for ( 1 .. 3 ) {
12    print "    ", 1 + int( rand( 6 ) ), "\n";
13 }
```

這是執行的情況。

```
[root@flash 1-4]# perl random.pl
由random函數所產生的隨機數:
    0.158114187027426
    0.8111110815822275
    0.998143547951383

由rand( 100 )函數所產生的隨機數:
    96.7597338460969
    63.8263747922867
    57.9399667665697

1 + int( rand( 6 ) )的數值:
    5
    1
    2
```

範例：generator.pl

我們可以使用 srand( )**函數來設定起始點。**

第四行的 srand(1)**函數會設定每次隨機數都是從** 1 **開始的起點，因此會顯示三次都是從** 1 **開始。**

第十行會設定起始點為隨機數。

```
#!/usr/bin/perl
for ( 1 .. 3 ) {
    print "\n\n設定起始爲1\n";
    srand( 1 );
    for ( 1 .. 3 ) {
        print "  ", 1 + int( rand( 6 ) );
    }
}
print "\n\n重新設定起始點\n";
srand();
for (1 .. 3) {
    print "\n";
    for ( 1 .. 3 ) {
        print "  ", 1 + int( rand( 6 ) );
    }
}
print "\n";
```

第二到第九行的每一次執行都是從 1 開始。
第十行因為從新設定了啟始點，所以第十一行到第十六行的啟始點都未必相同。
這是兩次執行的情況。

```
[root@flash 1-4]# perl generator.pl


設定起始爲1
   1   3   6

設定起始爲1
   1   3   6

設定起始爲1
   1   3   6

重新設定起始點

   1   5   3
   5   4   3
   1   1   6
[root@flash 1-4]# perl generator.pl


設定起始爲1
   1   3   6

設定起始爲1
   1   3   6

設定起始爲1
   1   3   6

重新設定起始點

   3   6   2
   2   2   6
   1   3   4
```

1-4-5 遞迴函數

遞迴函數就是自己呼叫自己函數的意義。對於某一些難以解決的問題，遞回提供了一個自然、簡單的解決方案，因此遞迴真的是很有用的方法。遞迴呼叫和迴圈最大的差異是設計上的簡易。一般遞迴呼叫可以使用直觀的方法來解決問題，而比較難用迴圈來設計程式來解決問題。但是，遞迴函數需要用到堆疊來放置參數，而這會使用到較多的記憶體。解決問題時，如果覺的使用遞迴還解決較簡單則用遞迴，如果使用迴圈較簡單則用迴圈。我們如果要計算 n*(n-1)*(n-2)….1 **時，也就是** n!，**則可以用遞迴和非遞迴的方式來解決。**

範例：cal.pl

這是使用非遞迴的方式來解決 n!**的問題。**

第四行到第六行為 for **迴圈，它會計算** 1*2*3*4*5。

第五行的特別變數$_**會儲存每一個迴圈中目前串列的元素，分別是** 1、2、3、4、5。

第五行我們也可以寫成$factorial *= $_;

```
1 #!/usr/bin/perl
2 $number=5;
3 $factorial=1;
4 foreach(1..$number){
5   $factorial = $factorial*$_;
6 }
7 print $factorial;
8 print "\n";
```

這是執行的情況。

```
[root@flash 1-4]# perl cal.pl
120
```

範例：recursive.pl

我們在第三行到第十二行定義 factorial( )**函數，可以求** n!**的數值。遞迴函數** factorial( )**在第十行呼叫自己，並且將**$n –1 **的值帶入遞迴函數。這樣的遞迴數學式就是** N*(N-1)*((N-1)-1)*….*1。**最後當**$number **為** 1 **時就會回傳** 1。

第五行的關鍵字 my **是區域範圍的語意，它定義**$number **為語意變數，也就是**$number **只有作用在這個** sub **副程式的區塊中，當區塊結束時，語意變數**$number **也就結束了。**

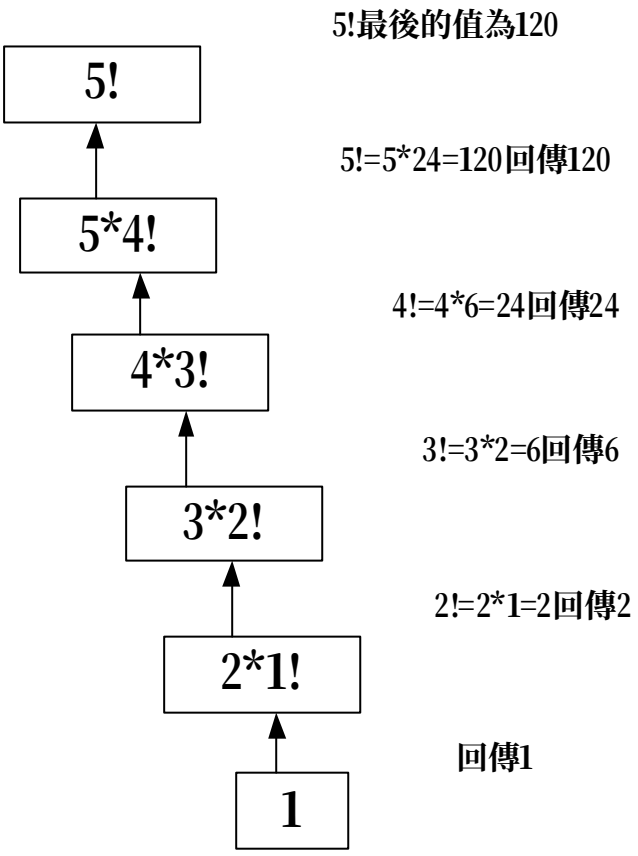
第五行的 shift **函數會移除第一個參數陣列元素**@_**，然後再將該元素的值給**$number **區域變數。**

```
1 #!/usr/bin/perl
2 print factorial(5);
3 sub factorial
4 {
5     my $number = shift;
6
7     if ( $number <= 1 ) {
8         return 1;
9     }else{
10        return $number * factorial( $number - 1 );
11    }
12 }
13 print "\n";
```

這是執行的情況。

```
[root@flash 1-4]# perl recursive.pl
120
```

這是遞迴呼叫 factorial(副程式的過程。每一次 factorial 函數都會呼叫自己，並且將 n-1 代入到 factorial( )副程式中，直到最後的情況為 1 時開始回傳它的值。



範例：fibonacci.pl

費氏系數 fibonacci **是數學上一個很好玩的觀念，我們可以使用遞迴呼叫副程式來解決費式系數的問題。**

我們可以將費氏系數的序列定義寫成

fibonacci(0)=0;

fibonacci(1)=1;

fibonacci(n)=fibonacci(n-1)+fibonacci(n-2)

費氏系數是前面兩個費氏系數的合。

第二行宣告陣列@values**，此陣列有** 0、1、2、3、4、5、10、20、30 **九個的數值。**

第三行到第五行為對於每一個陣列的值，都會在第四行的特別變數$_**列印出。**

第六行到第十四行為定義費氏系數的副程式。

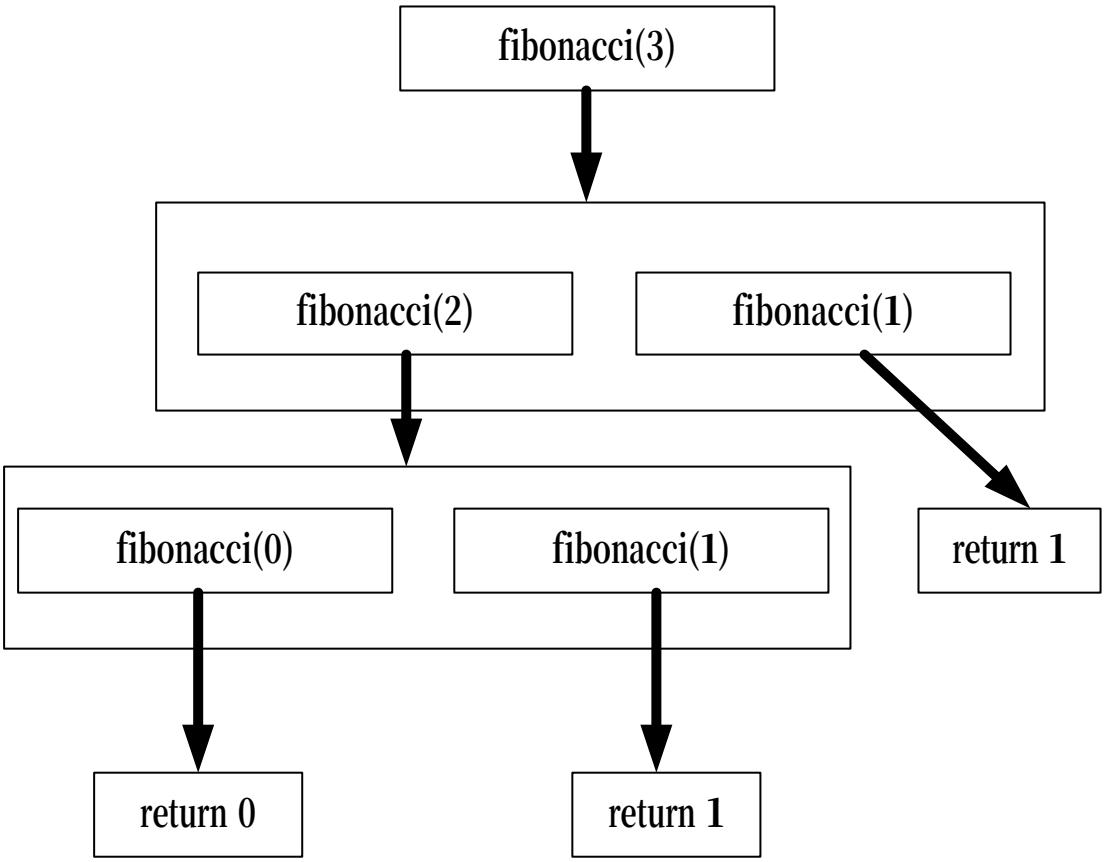
第九行的條件是當費氏系數的參數為 0 **或** 1 **時，就會回傳** 0 **或** 1**。當其值不為** 1 **時就會回傳第十二行的費氏系數遞迴呼叫。**

```
1 #!/usr/bin/perl
2 @values = (0, 1, 2, 3, 4, 5,10,20,30);
3 foreach ( @values ) {
4    print "fibonacci( $_ ) = ", fibonacci( $_ ), "\n";
5 }
6 sub fibonacci
7 {
8    my $number = shift;
9    if ( $number == 0 or $number == 1 ) {
10      return $number;
11   }else{
12      return fibonacci( $number - 1 ) + fibonacci( $number - 2 );
13   }
14 }
```

這是我們執行的情況。當費氏系數為 0 **時，它就會回傳** 0**，也就是執行第十行。當費氏系數為** 1 **時，它就會回傳** 1**，也是執行第十行。。當費氏系數為** 2 **時，它會執行第十二行，則會回傳** fibonacci(1)+fibonacci(0)**，也就是回傳** 1**。當費氏系數為** 3 **時，它會執行第十二行，** fibonacci(2)+fibonacci(1)**，這可以再分解成** {fibonacci(1)+fibonacci(0)}+fibonacci(1)}**，也就是** 2**。當費氏系數為** 4 **時，它就會執行第十二行，** fibonacci(3)+fibonacci(2)**，而這就是** 3**。**

```
[root@flash 1-4]# perl fibonacci.pl
fibonacci( 0 ) = 0
fibonacci( 1 ) = 1
fibonacci( 2 ) = 1
fibonacci( 3 ) = 2
fibonacci( 4 ) = 3
fibonacci( 5 ) = 5
fibonacci( 10 ) = 55
fibonacci( 20 ) = 6765
fibonacci( 30 ) = 832040
```

當費氏系數為 3 時，它會執行第十二行，fibonacci(2)+fibonacci(1)，這可以再分解成{fibonacci(1)+fibonacci(0)}+fibonacci(0)，也就是 2。費氏系數會遞迴呼叫下去。



副程式fibonacci的遞迴呼叫

1-4-6 變數的生存空間

變數可以根據它的生存空間(活動範圍)分為全域變數、語意變數和區域變數。全域變數作用的是整個程式。任何沒有加上關鍵字 local **或** my **的變數都是屬於全域變數。語義變數存在的空間只限定在其定義的區塊內。而區域變數則又稱為動態範圍變數。區域變數則是可以作用在整個定義它的範圍，直到該範圍被摧毀為止。**

關鍵字 our **名確的定義全域變數。關鍵字** my **定義了語意變數。關鍵字** local **定義了區域變數。**

範例：scope.pl

我們在第十三行到第二十行宣告副程式 subroutine1。

我們在第二十一行到第二十五行宣告副程式 subroutine2。

我們在第九行呼叫副程式 subroutine1( )。

我們在第十二行呼叫副程式 subroutine2( )。

我們在第四行使用 our **宣告全域變數**$x。

我們在第五行使用 our **宣告全域變數**$y。

我們在副程式 subroutine1( )**內，第十五行使用** my **關鍵字設定語意變數**$x**，這個變數只存在於此區塊內。**

第十六行我們使用 local**定義了區域變數**$y**，它作用的範圍包含** subroutine1 **整個的副程式，因此也包含呼叫的** surboutine2 **副程式，直到** subroutine1 **結束。**

```
#!/usr/bin/perl
print "沒有全域變數:\n";
subroutine1();
our $x = 3;
our $y = 6;
print "\n有全域變數:";
print "\n全域變數 \$x: $x";
print "\n全域變數 \$y: $y\n";
subroutine1();
print "\n全域變數 \$x: $x";
print "\n全域變數 \$y: $y\n";
subroutine2();
sub subroutine1
{
   my $x = 9;
   local $y = 8;
   print "\$x 副程式subroutine1: $x\t(語意變數設定到副程式subroutine1)\n";
   print "\$y 副程式subroutine1: $y\t(區域變數設定到副程式subroutine1)\n";
   subroutine2();
}
sub subroutine2
{
   print "\$x 副程式 subroutine2: $x\t(全域變數)\n";
   print "\$y 副程式 subroutine2: $y\t(區域變數設定到副程式subroutine1)\n";
}
```

Subroutine1 的變數$x 是語意變數，因此它只作用在 subroutine1 的區塊內，因此 subroutine2 的 print 就不會顯示出來。
Subroutine1 的變數$y 是區域變數，因此它會作用到直到 subroutine1 的副程式結束為止，因此 subroutine2 也會被執行到。
全域變數則是作用到整個程式的範圍，因此在有全域變數之下$x 副程式 subroutine2 會顯示全域變數的值 3。

```
[root@flash 1-4]# perl scope.pl
沒有全域變數:
$x 副程式subroutine1: 9 (語意變數設定到副程式subroutine1)
$y 副程式subroutine1: 8 (區域變數設定到副程式subroutine1)
$x 副程式 subroutine2:  (全域變數)
$y 副程式 subroutine2: 8        (區域變數設定到副程式subroutine1)

有全域變數:
全域變數 $x: 3
全域變數 $y: 6
$x 副程式subroutine1: 9 (語意變數設定到副程式subroutine1)
$y 副程式subroutine1: 8 (區域變數設定到副程式subroutine1)
$x 副程式 subroutine2: 3        (全域變數)
$y 副程式 subroutine2: 8        (區域變數設定到副程式subroutine1)

全域變數 $x: 3
全域變數 $y: 6
$x 副程式 subroutine2: 3        (全域變數)
$y 副程式 subroutine2: 6        (區域變數設定到副程式subroutine1)
```

1-5 物件和模組

類別是抽象的，而物件就是類別的實體。物件導向程式設計(OOP)就是將所有的東西看成是物件。物件導向程式設計封裝資料和函數，並且將它們包裝成一個類別 class。當我們將資料和函數封裝成類別後，我們可以將類別實作成物件。我們可以使用 new 這個建構子來實作物件。當我們實作完類別的物件後，我們可以使用"物件名稱->方法"來呼叫該物件的方法。

範例：file.pl

Perl 提供 FileHandle 模組來處理檔案。我們在使用 Perl 所內建的模組前，我們需使用 use 關鍵字。在 file.pl 中，它會將輸入的檔案 input.txt 內容輸出到 file.txt。

第四行我們使用 use 關鍵字來使用 Perl 的檔案處理模組 FileHandle。

第五行我們使用建構子 new 來建立檔案處理物件，並將它分配給純量變數$write。

第六行我們使用建構子 new 來建立檔案處理物件，並將它分配給純量變數$read。

第七行使用檔案處理 FileHandle 的 open 方法，我們使用$write->open(">file.txt") 來呼叫$write 物件的函數 open( )打開檔案 file.txt。

第八行使用檔案處理 FileHandle 的 open 方法，我們使用$read->open("input.txt") 來呼叫$read 物件的函數 open( )打開檔案 input.txt。

第九行是存取$write 檔案處理物件的變數。

第十一行到第十三行是迴圈。

第十一行使用 getline( )方法來從$read 檔案處理物件讀取檔案的一行。

第十二行使用 print( )方法將所讀取的資料輸出到$write 物件。

```
1 #!/usr/bin/perl
2 use warnings;
3 use strict;
4 use FileHandle;
5 my $write = new FileHandle;
6 my $read  = new FileHandle;
7 $write->open( ">file.txt" ) or die( "Could not open write" );
8 $read->open( "input.txt" ) or  die( "Could not open read" );
9 $write->autoflush( 1 );
10 my $i = 1;
11 while ( my $line = $read->getline() ) {
12    $write->print( $i++, " $line" );
13 }
```

這是我們$read 物件所打開的檔案 input.txt，第八行的$read->open("intput.txt") 會打開這個檔案，第十一行的$read->getline( )會讀取 input.txt 的每一行。

#vi input.txt

吳佳諺 大德 您好 :

您的每日靜思語

原諒別人就是善待自己。

這是我們$write 物件所輸入完成的檔案 file.txt，第十二行的$write->open(“file.txt”)會打開這個檔案，第十一行的$write->print( )會輸入從$read 讀取的每一行到 file.txt 檔案中。
這就是最後的結果。file.pl 會將 input.txt 檔案的內容輸出到 file.txt 中。
#vi file.txt
吳佳諺 大德 您好 :
您的每日靜思語
原諒別人就是善待自己。

1-5-1 實作自訂類別
範例：Date.pm
我們要建立一個 Date 類別套件模組。只要使用 use Date 就可以使用這個 Date 類別套件模組。
我們在第二行宣告這是我們自訂的類別套件模組 Date。
第五行到第十二行是定義 Date 類別套件模組的建構子。在這裏我們在第七行建立雜湊參考物件的資料。我們在第十行使用 bless( )函數將雜湊參考轉為 Date 物件，因此這個物件型態是預設的套件名稱。
我們在第十三行到第三十行建立 year、month 和 day 方法允許存取相關 Date 物件的方法。假如參數傳入 year、month 或 day 方法，則 the_year、the_month 或 the_day 的值可以被改變。
第三十一行到第四十一行的 setDate 方法允許使用者設定所有的方法。第三十三行的 if 判別式會檢查是否有足夠的參數輸入，如果是會執行第三十四到第三十七行設定 mont、year 和 day 屬性資料。
第四十二行到第五十行的 print 方法以 month/day/year 的格式輸出。

```
#!/usr/bin/perl
package Date;
use strict;
use warnings;
sub new
{
    my $date = { the_year => 1000,
                 the_month => 1,
                 the_day => 1, };
    bless( $date );
    return $date;
}
sub year
{
    my $self = shift();
    $self->{ the_year } = shift() if ( @_ );
    return $self->{ the_year };
}
sub month
{
    my $self = shift();
    $self->{ the_month } = shift() if ( @_ );
    return $self->{ the_month };
}
```

```
25 sub day
26 {
27    my $self = shift();
28    $self->{ the_day } = shift() if ( @_ );
29    return $self->{ the_day };
30 }
31 sub setDate
32 {
33    if ( @_ == 4 ) {
34       my $self = shift();
35       $self->month( $_[ 0 ] );
36       $self->day( $_[ 1 ] );
37       $self->year( $_[ 2 ] );
38    }else {
39       print( "請輸入三個參數\n" );
40    }
41 }
42 sub print
43 {
44    my $self = shift();
45    print( $self->month );
46    print( "/" );
47    print( $self->day );
48    print( "/" );
49    print( $self->year );
50 }
51 return 1;
```

1-5-2 使用物件

在前面的自訂類別且在建立 Date 模組後，我們可以使用 use 來將 Date 模組給包含進來檔案中。

範例：day.pl

第二行使用 use Date 將 Date.pm 模組包含進來 day.pl 的檔案中。

第五行使用 new 來建立 Date 的物件$now。

第六行使用 setDate( )函數來設定日期，也就是輸入三個參數。

第七行使用$now->month( )來存取$now 物件的 month 屬性。

第九行使用$now->print( )來運用$now 物件的 print( )方法將日期給顯示出來。

```
1 #!/usr/bin/perl
2 use Date;
3 use strict;
4 use warnings;
5 my $now = new Date;
6 $now->setDate( 11, 10, 2003 );
7 print $now->month() ;
8 print "\n";
9 $now->print();
10 print  "\n" ;
```

這是執行的情況。

```
[root@flash 1-5]# perl day.pl
11
11/10/2003
```