**perl教程**

第一节：  
基本上，简单变量就是一个数据单元，这个单元可以是数字或字符串。   
一、整型   
1、整型   
PERL最常用的简单变量，由于其与其它语言基本相同，不再赘述。   
例：   
$x = 12345;   
if (1217 + 116 == 1333) {   
# statement block goes here   
}   
整型的限制：   
PERL实际上把整数存在你的计算机中的浮点寄存器中，所以实际上被当作浮点数看待。在多数计算机中，浮点寄存器可以存贮约16位数字，长于此的被丢弃。整数实为浮点数的特例。   
2、8进制和16进制数   
8进制以0打头，16进制以0x打头。   
例：$var1 = 047; (等于十进制的39)   
$var2 = 0x1f; (等于十进制的31)   
二、浮点数   
如 11.4 、 -0.3 、.3 、 3. 、 54.1e+02 、 5.41e03   
浮点寄存器通常不能精确地存贮浮点数，从而产生误差，在运算和比较中要特别注意。指数的范围通常为-309到+308。   
例：   
  
#!/usr/local/bin/perl   
$value = 9.01e+21 + 0.01 - 9.01e+21;   
print ("first value is ", $value, "\n");   
$value = 9.01e+21 - 9.01e+21 + 0.01;   
print ("second value is ", $value, "\n");   
  
---------------------------------------------------------   
  
$ program3\_3   
first value is 0   
second value is 0.01   
三、字符串   
惯用C的程序员要注意，在PERL中，字符串的末尾并不含有隐含的NULL字符，NULL字符可以出现在串的任何位置。   
. 双引号内的字符串中支持简单变量替换，例如：   
$number = 11;   
$text = "This text contains the number $number.";   
则$text的内容为："This text contains the number 11."   
  
.双引号内的字符串中支持转义字符   
Table 3.1. Escape sequences in strings.   
  
Escape Sequence Description   
\a Bell (beep)   
\b Backspace   
\cn The Ctrl+n character   
\e Escape   
\E Ends the effect of \L, \U or \Q   
\f Form feed   
\l Forces the next letter into lowercase   
\L All following letters are lowercase   
\n Newline   
\r Carriage return   
\Q Do not look for special pattern characters   
\t Tab   
\u Force next letter into uppercase   
\U All following letters are uppercase   
\v Vertical tab   
  
\L、\U、\Q功能可以由\E关闭掉，如：   
$a = "T\LHIS IS A \ESTRING"; # same as "This is a STRING"   
  
.要在字符串中包含双引号或反斜线，则在其前加一个反斜线，反斜线还可以取消变量替换，如：   
$res = "A quote \" and A backslash \\";   
$result = 14;   
print ("The value of \$result is $result.\n")的结果为：   
The value of $result is 14.   
  
.可用\nnn(8进制)或\xnn(16进制)来表示ASCII字符，如：   
$result = "\377"; # this is the character 255,or EOF   
$result = "\xff"; # this is also 255   
  
.单引号字符串   
单引号字符串与双引号字符串有两个区别，一是没有变量替换功能，二是反斜线不支持转义字符，而只在包含单引号和反斜线时起作用。单引号另一个特性是可以跨多行，如：   
$text = 'This is two   
lines of text   
';   
与下句等效：   
$text = "This is two\nlines of text\n";   
  
.字符串和数值的互相转换   
例1：   
$string = "43";   
$number = 28;   
$result = $string + $number; # $result = 71   
若字符串中含有非数字的字符，则从左起至第一个非数字的字符，如：   
$result = "hello" \* 5; # $result = 0   
$result = "12a34" +1; # $result = 13   
  
.变量初始值   
在PERL中，所有的简单变量都有缺省初始值：""，即空字符。但是建议给所有变量赋初值，否则当程序变得大而复杂后，很容易出现不可预料且很难调试的错误。  
  
  
第二节：  
  
一、算术操作符 ：+(加)、-(减)、\*(乘)、/(除)、\*\*(乘幂)、%(取余)、-(单目负)  
(1)乘幂的基数不能为负，如 (-5) \*\* 2.5 # error；  
(2)乘幂结果不能超出计算机表示的限制，如10 \*\* 999999 # error  
(3)取余的操作数如不是整数，四舍五入成整数后运算；运算符右侧不能为零  
(4)单目负可用于变量： - $y ; # 等效于 $y \* -1  
二、整数比较操作符  
  
  
  
　  
  
Table 3.1. 整数比较操作符  
  
操作符 描述  
< 小于  
> 大于  
== 等于  
<= 小于等于  
>= 大于等于  
!= 不等于  
<=> 比较，返回 1, 0, or -1   
  
  
操作符<=>结果为：  
0 - 两个值相等  
1 - 第一个值大  
1 - 第二个值大  
三、字符串比较操作符  
  
  
  
Table 3.2. 字符串比较操作符  
  
操作符 描述 　  
lt 小于 　  
gt 大于 　  
eq 等于 　  
le 小于等于 　  
ge 大于等于 　  
ne 不等于 　  
cmp 比较，返回 1, 0, or -1 　  
  
  
四、逻辑操作符  
逻辑或：$a || $b 或 $a or $b  
逻辑与：$a && $b 或 $a and $b  
逻辑非：! $a 或 not $a  
逻辑异或：$a xor $b  
五、位操作符  
位与：&  
位或：|  
位非：~  
位异或：^  
左移：$x << 1  
右移：$x >> 2  
注：不要将&用于负整数，因为PERL将会把它们转化为无符号数。  
六、赋值操作符  
  
  
  
Table 3.3. 赋值操作符  
  
操作符 描述  
= Assignment only  
+= Addition and assignment   
-= Subtraction and assignment   
\*= Multiplication and assignment   
/= Division and assignment   
%= Remainder and assignment   
\*\*= Exponentiation and assignment   
&= Bitwise AND and assignment   
|= Bitwise OR and assignment   
^= Bitwise XOR and assignment   
  
  
  
  
Table 3.4. 赋值操作符例子  
  
表达式 等效表达式   
$a = 1; none (basic assignment)  
$a -= 1; $a = $a - 1;   
$a \*= 2; $a = $a \* 2;   
$a /= 2; $a = $a / 2;   
$a %= 2; $a = $a % 2;   
$a \*\*= 2; $a = $a \*\* 2;   
$a &= 2; $a = $a & 2;   
$a |= 2; $a = $a | 2;   
$a ^= 2; $a = $a ^ 2;   
  
  
.=可在一个赋值语句中出现多次，如：  
$value1 = $value2 = "a string";  
.=作为子表达式  
($a = $b) += 3;  
等价于  
$a = $b;  
$a += 3;  
但建议不要使用这种方式。  
七、自增自减操作符 ：++、--(与C++中的用法相同)  
.不要在变量两边都使用此种操作符：++$var-- # error  
.不要在变量自增/减后在同一表达式中再次使用：$var2 = $var1 + ++$var1; # error  
.在PERL中++可用于字符串，但当结尾字符为'z'、'Z'、'9'时进位，如：  
$stringvar = "abc";  
$stringvar++; # $stringvar contains "abd" now  
  
$stringvar = "aBC";  
$stringvar++; # $stringvar contains "aBD" now  
  
$stringvar = "abz";  
$stringvar++; # $stringvar now contains "aca"  
  
$stringvar = "AGZZZ";  
$stringvar++; # $stringvar now contains "AHAAA"  
  
$stringvar = "ab4";  
$stringvar++; # $stringvar now contains "ab5"  
  
$stringvar = "bc999";  
$stringvar++; # $stringvar now contains "bd000"  
.不要使用--，PERL将先将字符串转换为数字再进行自减  
$stringvar = "abc";  
$stringvar--; # $stringvar = -1 now  
  
.如果字符串中含有非字母且非数字的字符，或数字位于字母中，则经过++运算前值转换为数字零，因此结果为1，如：  
$stringvar = "ab\*c";  
$stringvar++;  
$stringvar = "ab5c";  
$stringvar++;  
八、字符串联结和重复操作符  
联接: .  
重复：x  
联接且赋值(类似+=)： .=  
例：  
$newstring = "potato" . "head";  
$newstring = "t" x 5;  
$a = "be";  
$a .= "witched"; # $a is now "bewitched"  
九、逗号操作符  
其前面的表达式先进行运算，如：  
$var1 += 1, $var2 = $var1;  
等价于  
$var1 += 1;  
$var2 = $var1;  
使用此操作符的唯一理由是提高程序的可读性，将关系密切的两个表达式结合在一起，如：  
$val = 26;  
$result = (++$val, $val + 5); # $result = 32  
注意如果此处没有括号则意义不同：  
$val = 26;  
$result = ++$val, $val + 5; # $result = 27  
十、条件操作符  
与C中类似，条件?值1:值2，当条件为真时取值1，为假时取值2，如：  
$result = $var == 0 ? 14 : 7;  
$result = 43 + ($divisor == 0 ? 0 : $dividend / $divisor);  
PERL 5中，还可以在赋值式左边使用条件操作符来选择被赋值的变量，如：  
$condvar == 43 ? $var1 : $var2 = 14;  
$condvar == 43 ? $var1 = 14 : $var2 = 14;  
十一、操作符的次序  
  
  
  
Table 3.6. 操作符次序  
  
操作符 描述  
++, -- 自增，自减  
-, ~, ! 单目  
\*\* 乘方  
=~, !~ 模式匹配  
\*, /, %, x 乘，除，取余，重复  
+, -, . 加，减，联接  
<<, >> 移位  
-e, -r, etc. 文件状态  
<, <=, >, >=, lt, le, gt, ge 不等比较  
==, !=, <=>, eq, ne, cmp 相等比较  
& 位与  
|, ^ 位或，位异或  
&& 逻辑与  
|| 逻辑或  
.. 列表范围  
? and : 条件操作符  
=, +=, -=, \*=, 赋值  
and so on 　  
, 逗号操作符  
not Low-precedence logical NOT  
and Low-precedence logical AND  
or, xor Low-precedence logical OR and XOR  
  
.操作符结合性(associativity)：  
  
  
Table 3.7. 操作符结合性  
  
操作符 结合性  
++, -- 无  
-, ~, ! Right-to-left  
\*\* Right-to-left  
=~, !~ Left-to-right  
\*, /, %, x Left-to-right  
+, -, . Left-to-right  
<<, >> Left-to-right  
-e, -r, 无  
<, <=, >, >=, lt, le, gt, ge Left-to-right  
==, !=, <=>, eq, ne, cmp Left-to-right  
& Left-to-right  
|, ^ Left-to-right  
&& Left-to-right  
|| Left-to-right  
.. Left-to-right  
? and : Right-to-left  
=, +=, -=, \*=, Right-to-left  
and so on 　  
, Left-to-right  
not Left-to-right  
and Left-to-right  
or, xor Left-to-right  
  
  
  
建议：  
1、当你不确定某操作符是否先执行时，一定要用括号明确之。  
2、用多行、空格等方式提高程序的可读性。  
  
  
第三节：  
  
一、列表  
列表是包含在括号里的一序列的值，可以为任何数值，也可为空，如：(1, 5.3 , "hello" , 2)，空列表：()。  
注：只含有一个数值的列表(如：(43.2) )与该数值本身(即：43.2 )是不同的，但它们可以互相转化或赋值。  
列表例：  
(17, $var, "a string")  
(17, 26 << 2)  
(17, $var1 + $var2)  
($value, "The answer is $value")  
二、数组--列表的存贮  
列表存贮于数组变量中，与简单变量不同，数组变量以字符"@"打头，如：  
@array = (1, 2, 3);  
注：  
(1)数组变量创建时初始值为空列表：()。  
(2)因为PERL用@和$来区分数组变量和简单变量，所以同一个名字可以同时用于数组变量和简单变量，如：  
$var = 1;  
@var = (11, 27.1 , "a string");  
但这样很容易混淆，故不推荐。  
1、数组的存取  
.对数组中的值通过下标存取，第一个元素下标为0。试图访问不存在的数组元素，则结果为NULL，但如果给超出数组大小的元素赋值，则数组自动增长，原来没有的元素值为NULL。如：  
@array = (1, 2, 3, 4);  
$scalar = $array[0];  
$array[3] = 5; # now @array is (1,2,3,5)  
$scalar = $array[4]; # now $scalar = null;  
$array[6] = 17; # now @array is (1,2,3,5,"","",17)  
.数组间拷贝  
@result = @original;  
.用数组给列表赋值  
@list1 = (2, 3, 4);  
@list2 = (1, @list1, 5); # @list2 = (1, 2, 3, 4, 5)  
.数组对简单变量的赋值  
(1) @array = (5, 7, 11);  
($var1, $var2) = @array; # $var1 = 5, $var2 = 7, 11被忽略  
(2) @array = (5, 7);  
($var1, $var2, $var3) = @array; # $var1 = 5, $var2 = 7, $var3 ="" (null)  
.从标准输入(STDIN)给变量赋值  
$var = <STDIN>;  
@array = <STDIN>; # ^D为结束输入的符号  
2 、字符串中的方括号和变量替换  
"$var[0]" 为数组@var的第一个元素。  
"$var[0]" 将字符"["转义，等价于"$var". "[0]"，$var被变量替换，[0]保持不变。  
"${var}[0]" 亦等价于"$var" ."[0]"。  
"${var}"则取消了大括号的变量替换功能，包含文字：${var}.  
3、列表范围：  
(1..10) = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10)  
(2, 5..7, 11) = (2, 5, 6, 7, 11)  
(3..3) = (3)  
.用于实数  
(2.1..5.3) = (2.1, 3.1 ,4.1, 5.1)  
(4.5..1.6) = ()  
.用于字符串  
("aaa".."aad") = ("aaa","aab", "aac", "aad")  
@day\_of\_month = ("01".."31")  
.可包含变量或表达式  
($var1..$var2+5)  
.小技巧：  
$fred = "Fred";  
print (("Hello, " . $fred . "!n") x 2);  
其结果为：  
Hello, Fred!  
Hello, Fred!  
4、数组的输出：  
(1) @array = (1, 2, 3);  
print (@array, "n");  
结果为：  
123  
(2) @array = (1, 2, 3);  
print ("@arrayn");  
结果为：  
1 2 3  
5、列表/数组的长度  
当数组变量出现在预期简单变量出现的地方，则PERL解释器取其长度。  
@array = (1, 2, 3);  
$scalar = @array; # $scalar = 3,即@array的长度  
($scalar) = @array; # $scalar = 1,即@array第一个元素的值  
注：以数组的长度为循环次数可如下编程：  
$count = 1;  
while ($count <= @array) {  
print ("element $count: $array[$count-1]n");  
$count++;  
}  
6、子数组  
@array = (1, 2, 3, 4, 5);  
@subarray = @array[0,1]; # @subarray = (1, 2)  
@subarray2 = @array[1..3]; # @subarray2 = (2,3,4)  
@array[0,1] = ("string", 46); # @array =("string",46,3,4,5) now  
@array[0..3] = (11, 22, 33, 44); # @array = (11,22,33,44,5) now  
@array[1,2,3] = @array[3,2,4]; # @array = (11,44,33,5,5) now  
@array[0..2] = @array[3,4]; # @array = (5,5,"",5,5) now  
可以用子数组形式来交换元素：  
@array[1,2] = @array[2,1];  
7、有关数组的库函数  
(1)sort--按字符顺序排序  
@array = ("this", "is", "a","test");  
@array2 = sort(@array); # @array2 = ("a","is", "test", "this")  
@array = (70, 100, 8);  
@array = sort(@array); # @array = (100, 70, 8) now  
( 2)reverse--反转数组  
@array2 = reverse(@array);  
@array2 = reverse sort (@array);  
(3)chop--数组去尾  
chop的意义是去掉STDIN（键盘）输入字符串时最后一个字符--换行符。而如果它作用到数组上，则将数组中每一个元素都做如此处理。  
@list = ("rabbit", "12345","quartz");  
chop (@list); # @list = ("rabbi", "1234","quart") now  
( 4)join/split--连接/拆分  
join的第一个参数是连接所用的中间字符，其余则为待连接的字符数组。  
$string = join(" ", "this", "is","a", "string"); # 结果为"this is a string"  
@list = ("words","and");  
$string = join("::", @list, "colons"); #结果为"words::and::colons"  
@array = split(/::/,$string); # @array = ("words","and", "colons") now  
  
  
　  
  
第四节：  
  
一、条件判断   
if ( <expression>) {   
<statement\_block\_1>   
}   
elsif ( <expression> ) {   
<statement\_block\_2>   
}   
...   
else{   
<statement\_block\_3>   
}   
二、循环：   
1、while循环   
while ( <expression> ) {   
<statement\_block>   
}   
2、until循环   
until ( <expression> ) {   
<statement\_block>   
}   
3、类C的for循环 ，如   
for ($count=1; $count <= 5; $count++) {   
# statements inside the loop go here   
}   
下面是在for循环中使用逗号操作符的例子：   
for ($line = <STDIN>, $count = 1; $count <= 3; $line = <STDIN>, $count++) {   
print ($line);   
}   
它等价于下列语句：   
$line = <STDIN>;   
$count = 1;   
while ($count <= 3) {   
print ($line);   
$line = <STDIN>;   
$count++;   
}   
4、针对列表(数组)每个元素的循环：foreach，语法为：   
foreach localvar (listexpr) {   
statement\_block;   
}   
例：   
foreach $word (@words) {   
if ($word eq "the") {   
print ("found the word 'the'n");   
}   
}   
注：   
(1)此处的循环变量localvar是个局部变量，如果在此之前它已有值，则循环后仍恢复该值。   
(2)在循环中改变局部变量，相应的数组变量也会改变，如：   
@list = (1, 2, 3, 4, 5);   
foreach $temp (@list) {   
if ($temp == 2) {   
$temp = 20;   
}   
}   
此时@list已变成了(1, 20, 3, 4, 5)。   
5、do循环   
do {   
statement\_block   
} while\_or\_until (condexpr);   
do循环至少执行一次循环。   
6、循环控制   
退出循环为last，与C中的break作用相同；执行下一个循环为next，与C中的continue作用相同；PERL特有的一个命令是redo，其含义是重复此次循环，即循环变量不变，回到循环起始点，但要注意，redo命令在do循环中不起作用。   
7、传统的goto label;语句。   
  
三、单行条件   
语法为statement keyword condexpr。其中keyword可为if、unless、while或until，如：   
print ("This is zero.n") if ($var == 0);   
print ("This is zero.n") unless ($var != 0);   
print ("Not zero yet.n") while ($var-- > 0);   
print ("Not zero yet.n") until ($var-- == 0);   
虽然条件判断写在后面，但却是先执行的。

四、引用和数组  
关于Perl语言应该记住的最重要的一点可能是：Perl中的数组和哈希表始终是一维的。因此，数组和哈希表只保存标量值，不直接存贮数组或其它的复杂数据结构。数组的成员要么是数（或字符串）要么是引用。  
对数组和哈希表可以象对简单变量一样使用反斜线操作符，数组的引用如下：  
  
  
1 #!/usr/bin/perl  
2 #  
3 # Using Array references  
4 #  
5 $pointer = @ARGV;  
6 printf "n Pointer Address of ARGV = $pointern";  
7 $i = scalar(@$pointer);  
8 printf "n Number of arguments : $i n";  
9 $i = 0;  
10 foreach (@$pointer) {  
11 printf "$i : $$pointer[$i++]; n";  
12 }  
  
运行结果如下：  
  
  
$ test 1 2 3 4  
Pointer Address of ARGV = ARRAY(0x806c378)  
Number of arguments : 4  
0 : 1;  
1 : 2;  
2 : 3;  
3 : 4;  
  
第5行将引用$pointer指向数组@ARGV，第6行输出ARGV的地址。$pointer返回数组第一个元素的地址，这与C语言中的数组指针是类似的。第7行调用函数scalar()获得数组的元素个数，该参数亦可为@ARGV，但用指针则必须用@$pointer的形式指定其类型为数组，$pointer给出地址，@符号说明传递的地址为数组的第一个元素的地址。第10行与第7行类似，第11行用形式$$pointer[$i]列出所有元素。  
对关联数组使用反斜线操作符的方法是一样的--把所有关联数组名换成引用$poniter。注意数组和简单变量（标量）的引用显示时均带有类型--ARRAY和SCALAR，哈希表（关联数组）和函数也一样，分别为HASH和CODE。下面是哈希表的引用的例子。  
  
  
#!/usr/bin/perl  
1 #  
2 # Using Associative Array references  
3 #  
4 %month = (  
5 '01', 'Jan',  
6 '02', 'Feb',  
7 '03', 'Mar',  
8 '04', 'Apr',  
9 '05', 'May',  
10 '06', 'Jun',  
11 '07', 'Jul',  
12 '08', 'Aug',  
13 '09', 'Sep',  
14 '10', 'Oct',  
15 '11', 'Nov',  
16 '12', 'Dec',  
17 );  
18  
19 $pointer = %month;  
20  
21 printf "n Address of hash = $pointern ";  
22  
23 #  
24 # The following lines would be used to print out the  
25 # contents of the associative array if %month was used.  
26 #  
27 # foreach $i (sort keys %month) {  
28 # printf "n $i $$pointer{$i} ";  
29 # }  
30  
31 #  
32 # The reference to the associative array via $pointer  
33 #  
34 foreach $i (sort keys %$pointer) {  
35 printf "$i is $$pointer{$i} n";  
36 }  
  
结果输出如下：  
  
  
$ mth  
Address of hash = HASH(0x806c52c)  
01 is Jan  
02 is Feb  
03 is Mar  
04 is Apr  
05 is May  
06 is Jun  
07 is Jul  
08 is Aug  
09 is Sep  
10 is Oct  
11 is Nov  
12 is Dec  
  
与数组类似，通过引用访问哈希表的元素形式为$$pointer{$index}，当然，$index是哈希表的键值，而不仅是数字。还有几种访问形式，此外，构建哈希表还可以用=>操作符，可读性更好些。下面再看一个例子：  
  
  
1 #!/usr/bin/perl  
2 #  
3 # Using Array references  
4 #  
5 %weekday = (  
6 '01' => 'Mon',  
7 '02' => 'Tue',  
8 '03' => 'Wed',  
9 '04' => 'Thu',  
10 '05' => 'Fri',  
11 '06' => 'Sat',  
12 '07' => 'Sun',  
13 );  
14 $pointer = %weekday;  
15 $i = '05';  
16 printf "n ================== start test ================= n";  
17 #  
18 # These next two lines should show an output  
19 #  
20 printf '$$pointer{$i} is ';  
21 printf "$$pointer{$i} n";  
22 printf '${$pointer}{$i} is ';  
23 printf "${$pointer}{$i} n";  
24 printf '$pointer->{$i} is ';  
25  
26 printf "$pointer->{$i}n";  
27 #  
28 # These next two lines should not show anything 29 #  
30 printf '${$pointer{$i}} is ';  
31 printf "${$pointer{$i}} n";  
32 printf '${$pointer->{$i}} is ';  
33 printf "${$pointer->{$i}}";  
34 printf "n ================== end of test ================= n";  
35  
  
结果输出如下：  
  
  
================== start test =================  
$$pointer{$i} is Fri  
${$pointer}{$i} is Fri  
$pointer->{$i} is Fri  
${$pointer{$i}} is  
${$pointer->{$i}} is  
================== end of test =================  
  
可以看到，前三种形式的输出显示了预期的结果，而后两种则没有。当你不清楚是否正确时，就输出结果看看。在Perl中，有不明确的代码就用print语句输出来实验一下，这能使你清楚Perl是怎样解释你的代码的。  
五、多维数组  
语句@array = list;可以创建数组的引用，中括号可以创建匿名数组的引用。下面语句为用于画图的三维数组的例子：  
$line = ['solid' , 'black' , ['1','2','3'] , ['4','5','6']];  
此语句建立了一个含四个元素的三维数组，变量$line指向该数组。前两个元素是标量，存贮线条的类型和颜色，后两个元素是匿名数组的引用，存贮线条的起点和终点。访问其元素语法如下：  
  
  
$arrayReference->[$index] single-dimensional array  
$arrayReference->[$index1][$index2] two-dimensional array  
$arrayReference->[$index1][$index2][$index3] three-dimensional array  
  
可以创建在你的智力、设计经验和计算机的内存允许的情况下极尽复杂的结构，但最好对可能读到或管理你的代码的人友好一些--尽量使代码简单些。另一方面，如果你想向别人炫耀你的编程能力，Perl给你足够的机会和能力编写连自己都难免糊涂的代码。http://www.linuxeden.com/forum/images/smilies/default/smile.gif  
建议：当你想使用多于三维的数组时，最好考虑使用其它数据结构来简化代码。  
下面为创建和使用二维数组的例子：  
  
  
1 #!/usr/bin/perl  
2 #  
3 # Using Multi-dimensional Array references  
4 #  
5 $line = ['solid', 'black', ['1','2','3'] , ['4', '5', '6']];  
6 print "$line->[0] = $line->[0] n";  
7 print "$line->[1] = $line->[1] n";  
8 print "$line->[2][0] = $line->[2][0] n";  
9 print "$line->[2][1] = $line->[2][1] n";  
10 print "$line->[2][2] = $line->[2][2] n";  
11 print "$line->[3][0] = $line->[3][0] n";  
12 print "$line->[3][1] = $line->[3][1] n";  
13 print "$line->[3][2] = $line->[3][2] n";  
14 print "n"; # The obligatory output beautifier.  
  
结果输出如下：  
  
  
$line->[0] = solid  
$line->[1] = black  
$line->[2][0] = 1  
$line->[2][1] = 2  
$line->[2][2] = 3  
$line->[3][0] = 4  
$line->[3][1] = 5  
$line->[3][2] = 6  
  
那么三维数组又如何呢？下面是上例略为改动的版本。  
  
  
1 #!/usr/bin/perl  
2 #  
3 # Using Multi-dimensional Array references again  
4 #  
5 $line = ['solid', 'black', ['1','2','3', ['4', '5', '6']]];  
6 print "$line->[0] = $line->[0] n";  
7 print "$line->[1] = $line->[1] n";  
8 print "$line->[2][0] = $line->[2][0] n";  
9 print "$line->[2][1] = $line->[2][1] n";  
10 print "$line->[2][2] = $line->[2][2] n";  
11 print "$line->[2][3][0] = $line->[2][3][0] n";  
12 print "$line->[2][3][1] = $line->[2][3][1] n";  
13 print "$line->[2][3][2] = $line->[2][3][2] n";  
14 print "n";  
  
结果输出如下：  
  
  
$line->[0] = solid  
$line->[1] = black  
$line->[2][0] = 1  
$line->[2][1] = 2  
$line->[2][2] = 3  
$line->[2][3][0] = 4  
$line->[2][3][1] = 5  
$line->[2][3][2] = 6  
  
访问第三层元素的方式形如$line->[2][3][0]，类似于C语言中的Array\_pointer[2][3][0]。本例中，下标均为数字，当然亦可用变量代替。用这种方法可以把数组和哈希表结合起来构成复杂的结构，如下：  
  
  
1 #!/usr/bin/perl  
2 #  
3 # Using Multi-dimensional Array and Hash references  
4 #  
5 %cube = (  
6 '0', ['0', '0', '0'],  
7 '1', ['0', '0', '1'],  
8 '2', ['0', '1', '0'],  
9 '3', ['0', '1', '1'],  
10 '4', ['1', '0', '0'],  
11 '5', ['1', '0', '1'],  
12 '6', ['1', '1', '0'],  
13 '7', ['1', '1', '1']  
14 );  
15 $pointer = %cube;  
16 print "n Da Cube n";  
17 foreach $i (sort keys %$pointer) {  
18 $list = $$pointer{$i};  
19 $x = $list->[0];  
20 $y = $list->[1];  
21 $z = $list->[2];  
22 printf " Point $i = $x,$y,$z n";  
23 }  
  
结果输出如下：  
  
  
Da Cube  
Point 0 = 0,0,0  
Point 1 = 0,0,1  
Point 2 = 0,1,0  
Point 3 = 0,1,1  
Point 4 = 1,0,0  
Point 5 = 1,0,1  
Point 6 = 1,1,0  
Point 7 = 1,1,1  
  
这是一个定义立方体的例子。%cube中保存的是点号和坐标，坐标是个含三个数字的数组。变量$list获取坐标数组的引用：$list = $$ pointer{$i}; 然后访问各坐标值：$x = $list->[0]; ... 也可用如下方法给$x、$y和$z赋值：($x,$y,$z) = @$list;  
使用哈希表和数组时，用$和用->是类似的，对数组而言下面两个语句等效：  
$$names[0] = "kamran";  
$names->[0] = "kamran";  
对哈希表而言下面两个语句等效：  
$$lastnames{"kamran"} = "Husain";  
$lastnames->{"kamran"} = "Husain";  
Perl中的数组可以在运行中创建和扩展。当数组的引用第一次在等式左边出现时，该数组自动被创建，简单变量和多维数组也是一样。如下句，如果数组contours不存在，则被创建：  
$contours[$x][$y][$z] = &xlate($mouseX, $mouseY);  
六、子程序的引用  
perl中子程序的引用与C中函数的指针类似，构造方法如下：  
$pointer\_to\_sub = sub {... declaration of sub ...};  
通过所构造的引用调用子程序的方法为：  
&$pointer\_to\_sub(parameters);  
子程序模板  
子程序的返回值不仅限于数据，还可以返回子程序的引用。返回的子程序在调用处执行，但却是在最初被创建的调用处被设置，这是由Perl对Closure处理的方式决定的。Closure意即如果你定义了一个函数，它就以最初定义的内容运行。(Closure详见OOP的参考书)下面的例子中，设置了多个错误信息显示子程序，这样的子程序定义方法可用于创建模板。  
  
  
#!/usr/bin/perl  
sub errorMsg {  
my $lvl = shift;  
#  
# define the subroutine to run when called.  
#  
return sub {  
my $msg = shift; # Define the error type now.  
print "Err Level $lvlhttp://www.linuxeden.com/forum/images/smilies/default/shy.gifmsgn"; }; # print later.  
}  
$severe = errorMsg("Severe");  
$fatal = errorMsg("Fatal");  
$annoy = errorMsg("Annoying");  
  
&$severe("Divide by zero");  
&$fatal("Did you forget to use a semi-colon?");  
&$annoy("Uninitialized variable in use");  
  
结果输出如下：  
  
  
Err Level Severehttp://www.linuxeden.com/forum/images/smilies/default/biggrin.gifivide by zero  
Err Level Fatalhttp://www.linuxeden.com/forum/images/smilies/default/biggrin.gifid you forget to use a semi-colon?  
Err Level Annoying:Uninitialized variable in use  
  
上例中，子程序errorMsg使用了局域变量$lvl，用于返回给调用者。当errorMsg被调用时，$lvl的值设置到返回的子程序内容中，虽然是用的my函数。三次调用设置了三个不同的$lvl变量值。当errorMsg返回时，$lvl的值保存到每次被声明时所产生的子程序代码中。最后三句对产生的子程序引用进行调用时$msg的值被替换，但$lvl的值仍是相应子程序代码创建时的值。  
很混淆是吗？是的，所以这样的代码在Perl程序中很少见。  
七、数组与子程序  
数组利于管理相关数据，本节讨论如何向子程序传递多个数组。前面我们讲过用@\_传递子程序的参数，但是@\_是一个单维数组，不管你传递的参数是多少个数组，都按序存贮在@\_中，故用形如my(@a,@b)=@\_; 的语句来获取参数值时，全部值都赋给了@a，而@b为空。那么怎么把一个以上的数组传递给子程序呢？方法是用引用。见下例：  
  
  
#!/usr/bin/perl  
@names = (mickey, goofy, daffy );  
@phones = (5551234, 5554321, 666 );  
$i = 0;  
sub listem {  
my ($a,$b) = @\_;  
foreach (@$a) {  
print "a[$i] = " . @$a[$i] . " " . "tb[$i] = " . @$b[$i] ."n";  
$i++;  
}  
}  
&listem(@names, @phones);  
  
结果输出如下：  
  
  
a[0] = mickey b[0] = 5551234  
a[1] = goofy b[1] = 5554321  
a[2] = daffy b[2] = 666  
注意：  
  
1、当想传递给子程序的参数是多于一个的数组时一定要使用引用。  
2、一定不要在子程序中使用形如 (@variable)=@\_; 的语句处理参数，除非你想把所有参数集中到一个长的数组中。  
  
　  
  
八、文件句柄的引用  
有时，必须将同一信息输出到不同的文件，例如，某程序可能在一个实例中输出到屏幕，另一个输出到打印机，再一个输出到记录文件，甚至同时输出到这三个文件。相比较于每种处理写一个单独的语句，可以有更好的实现方式如下：  
spitOut(\*STDIN);  
spitOut(\*LPHANDLE);  
spitOut(\*LOGHANDLE);  
其中子程序spitOut的代码如下：  
  
  
sub spitOut {  
my $fh = shift;  
print $fh "Gee Wilbur, I like this lettucen";  
}  
  
注意其中文件句柄引用的语法为\*FILEHANDLE。

第五节：  
  
一、定义   
子程序即执行一个特殊任务的一段分离的代码，它可以使减少重复代码且使程序易读。PERL中，子程序可以出现在程序的任何地方。定义方法为：   
sub subroutine{   
statements;   
}   
二、调用   
调用方法如下：   
1、用&调用   
&subname;   
...   
sub subname{   
...   
}   
2、先定义后调用 ，可以省略&符号   
sub subname{   
...   
}   
...   
subname;   
3、前向引用 ，先定义子程序名，后面再定义子程序体   
sub subname;   
...   
subname;   
...   
sub subname{   
...   
}   
4、用do调用   
do my\_sub(1, 2, 3);等价于&my\_sub(1, 2, 3);   
三、返回值   
缺省的，子程序中最后一个语句的值将用作返回值。语句return (retval);也可以推出子程序并返回值retval，retval可以为列表。   
四、局部变量   
子程序中局部变量的定义有两种方法：my和local。其区别是：my定义的变量只在该子程序中存在；而local定义的变量不存在于主程序中，但存在于该子程序和该子程序调用的子程序中(在PERL4中没有my)。定义时可以给其赋值，如：   
my($scalar) = 43;   
local(@array) = (1, 2, 3);   
五、子程序参数传递   
1、形式   
&sub1(&number1, $number2, $nubmer3);   
...   
sub sub1{   
my($number1, $number2, $number3) = @\_;   
...   
}   
2、传送数组   
&addlist (@mylist);   
&addlist ("14", "6", "11");   
&addlist ($value1, @sublist, $value2);   
...   
sub addlist {   
my (@list) = @\_;   
...   
}   
参数为数组时，子程序只将它赋给一个数组变量。如   
sub twolists {   
my (@list1, @list2) = @\_;   
}   
中@list2必然为空。但简单变量和数组变量可以同时传递：   
&twoargs(47, @mylist); # 47赋给$scalar，@mylist赋给@list   
&twoargs(@mylist); # @mylist的第一个元素赋给$scalar，其余的元素赋给@list   
...   
sub twoargs {   
my ($scalar, @list) = @\_;   
...   
}   
六、递归子程序   
PERL中，子程序可以互相调用，其调用方法与上述相同，当调用该子程序本身时，即成了递归子程序。递归子程序有两个条件：1、除了不被子程序改变的变量外，所有的变量必须的局部的；2、该子程序要含有停止调用本身的代码。   
七、用别名传递数组参数   
1、用前面讲到的调用方法&my\_sub(@array)将把数组@array的数据拷贝到子程序中的变量@\_中，当数组很大时，将会花费较多的资源和时间，而用别名传递将不做这些工作，而对该数组直接操作。形式如：   
@myarray = (1, 2, 3, 4, 5);   
&my\_sub(\*myarray);   
sub my\_sub {   
my (\*subarray) = @\_;   
}   
2、此方法类似于C语言中的传递数组的起始地址指针，但并不一样，在定义数组的别名之后，如果有同名的简单变量，则对该变量也是起作用的。如：   
$foo = 26;   
@foo = ("here's", "a", "list");   
&testsub (\*foo);   
...   
sub testsub {   
local (\*printarray) = @\_;   
...   
$printarray = 61;   
}   
当子程序执行完，主程序中的$foo的值已经成了61，而不再是26了。   
3、用别名的方法可以传递多个数组，如：   
@array1 = (1, 2, 3);   
@array2 = (4, 5, 6);   
&two\_array\_sub (\*array1, \*array2);   
sub two\_array\_sub {   
my (\*subarray1, \*subarray2) = @\_;   
}   
在该子程序中，subarray1是array1的别名，subarray2是array2的别名。   
八、预定义的子程序   
PERL5预定义了三个子程序，分别在特定的时间执行，它们是：BEGIN子程序在程序启动时被调用；END子程序在程序结束时被调用；AUTOLOAD子程序在找不到某个子程序时被调用。你可以自己定义它们，以在特定时间执行所需要的动作。如：   
BEGIN {   
print("Hi! Welcome to Perl!n");   
}   
AUTOLOAD{   
print("subroutine $AUTOLOAD not foundn"); # 变量$AUTOLOAD即未找到的子程序名   
print("arguments passed: @\_n");   
}   
若同一个预定义子程序定义了多个，则BEGIN顺序执行，END逆序执行。  
  
　  
  
第六节：  
  
一、数组变量的限制  
在前面讲的数组变量中，可以通过下标访问其中的元素。例如，下列语句访问数组@array的第三个元素：  
$scalar = $array[2];  
虽然数组很有用，但它们有一个显著缺陷，即很难记住哪个元素存贮的什么内容。假如我们来写一个程序计算某文件中首字母大写的单词出现的次数，用数组来实现就比较困难，程序代码如下：  
  
1 : #!/usr/local/bin/perl  
2 :  
3 : while ($inputline = <STDIN>) {  
4 : while ($inputline =~ /b[A-Z]S+/g) {  
5 : $word = $&;  
6 : $word =~ s/[;.,:-]$//; # remove punctuation  
7 : for ($count = 1; $count <= @wordlist;  
8 : $count++) {  
9 : $found = 0;  
10: if ($wordlist[$count-1] eq $word) {  
11: $found = 1;  
12: $wordcount[$count-1] += 1;  
13: last;  
14: }  
15: }  
16: if ($found == 0) {  
17: $oldlength = @wordlist;  
18: $wordlist[$oldlength] = $word;  
19: $wordcount[$oldlength] = 1;  
20: }  
21: }  
22: }  
23: print ("Capitalized words and number of occurrences:n");  
24: for ($count = 1; $count <= @wordlist; $count++) {  
25: print ("$wordlist[$count-1]: $wordcount[$count-1]n");  
26: }  
  
运行结果如下：  
  
  
Here is a line of Input.  
This Input contains some Capitalized words.  
^D  
Capitalized words and number of occurrences:  
Here: 1  
Input: 2  
This: 1  
Capitalized: 1  
  
这个程序每次从标准输入文件读一行文字，第四行起的循环匹配每行中首字母大写的单词，每找到一个循环一次，赋给简单变量$word。在第六行中去掉标点后，查看该单词是否曾出现过，7~15行中在@wordlist中挨个元素做此检查，如果某个元素与$word相等，@wordcount中相应的元素就增加一个数。如果没有出现过，即@wordlist中没有元素与$word相等，16~20行给@wordlist和@wordcount增加一个新元素。  
二、定义  
正如你所看到的，使用数组元素产生了一些问题。首先，@wordlist中哪个元素对应着哪个单词并不明显；更糟的是，每读进一个新单词，程序必须检查整个列表才能知道该单词是否曾经出现过，当列表变得较大时，这是很耗费时间的。  
这些问题产生的原因是数组元素通过数字下标访问，为了解决这类问题，Perl定义了另一种数组，可以用任意简单变量值来访问其元素，这种数组叫做关联数组，也叫哈希表。  
为了区分关联数组变量与普通的数组变量，Perl使用%作为其首字符，而数组变量以@打头。与其它变量名一样，%后的第一个字符必须为字母，后续字符可以为字母、数字或下划线。  
三、访问关联数组的元素  
关联数组的下标可以为任何简单/标量值，访问单个元素时以$符号打头，下标用大括号围起来。例如：  
  
$fruit{"bananas"}  
$number{3.14159}  
$integer{-7}  
  
简单变量也可作为下标，如：  
$fruit{$my\_fruit}  
四、增加元素  
创建一个关联数组元素最简单的方法是赋值，如语句$fruit{"bananas"} = 1； 把1赋给关联数组%fruit下标为bananas的元素，如果该元素不存在，则被创建，如果数组%fruit从未使用过，也被创建。  
这一特性使得关联数组很容易用于计数。下面我们用关联数组改写上面的程序，注意实现同样的功能此程序简化了许多。  
  
  
1 : #!/usr/local/bin/perl  
2 :  
3 : while ($inputline = ) {  
4 : while ($inputline =~ /b[A-Z]S+/g) {  
5 : $word = $&;  
6 : $word =~ s/[;.,:-]$//; # remove punctuation  
7 : $wordlist{$word} += 1;  
8 : }  
9 : }  
10: print ("Capitalized words and number of occurrences:n");  
11: foreach $capword (keys(%wordlist)) {  
12: print ("$capword: $wordlist{$capword}n");  
13: }  
  
运行结果如下：  
  
  
Here is a line of Input.  
This Input contains some Capitalized words.  
^D  
Capitalized words and number of occurrences:  
This: 1  
Input: 2  
Here: 1  
Capitalized: 1  
  
你可以看到，这次程序简单多了，读取输入并存贮各单词数目从20行减少到了7行。  
本程序用关联数组%wordlist跟踪首字母大写的单词，下标就用单词本身，元素值为该单词出现的次数。第11行使用了内嵌函数keys()。这个函数返回关联数组的下标列表，foreach语句就用此列表循环。  
注：关联数组总是随机存贮的，因此当你用keys()访问其所有元素时，不保证元素以任何顺序出现，特别值得一提的是，它们不会以被创建的顺序出现。  
要想控制关联数组元素出现的次序，可以用sort()函数对keys()返回值进行排列，如：  
  
foreach $capword (sort keys(%wordlist)) {  
print ("$capword: $wordlist{$capword}n");  
}  
  
　  
  
五、创建关联数组  
可以用单个赋值语句创建关联数组，如：  
%fruit = ("apples",17,"bananas",9,"oranges","none");  
此语句创建的关联数组含有下面三个元素：  
  
下标为apples的元素，值为17  
下标为bananas的元素，值为9  
下标为oranges的元素，值为none  
注：用列表给关联数组赋值时，Perl5允许使用"=>"或","来分隔下标与值，用"=>"可读性更好些，上面语句等效于：  
%fruit = ("apples"=>17,"bananas"=>9,"oranges"=>"none");  
六、从数组变量复制到关联数组  
与列表一样，也可以通过数组变量创建关联数组，当然，其元素数目应该为偶数，如：  
@fruit = ("apples",17,"bananas",9,"oranges","none");  
%fruit = @fruit;  
反之，可以把关联数组赋给数组变量，如：  
%fruit = ("grapes",11,"lemons",27);  
@fruit = %fruit;  
注意，此语句中元素次序未定义，那么数组变量@fruit可能为("grapes",11,"lemons",27)或("lemons",27,"grapes",11)。  
关联数组变量之间可以直接赋值，如：%fruit2 = %fruit1; 还可以把数组变量同时赋给一些简单变量和一个关联数组变量，如：  
($var1, $var2, %myarray) = @list;  
此语句把@list的第一个元素赋给$var1，第二个赋给$var2，其余的赋给%myarray。  
最后，关联数组可以通过返回值为列表的内嵌函数或用户定义的子程序来创建，下例中把split()函数的返回值--一个列表--赋给一个关联数组变量。  
  
1: #!/usr/local/bin/perl  
2:  
3: $inputline = <STDIN>;  
4: $inputline =~ s/^s+|s+n$//g;  
5: %fruit = split(/s+/, $inputline);  
6: print ("Number of bananas: $fruit{"bananas"}n");  
  
运行结果如下：  
  
oranges 5 apples 7 bananas 11 cherries 6  
Number of bananas: 11  
  
　  
  
七、元素的增删  
增加元素已经讲过，可以通过给一个未出现过的元素赋值来向关联数组中增加新元素，如$fruit{"lime"} = 1;创建下标为lime、值为1的新元素。  
删除元素的方法是用内嵌函数delete，如欲删除上述元素，则：  
delete ($fruit{"lime"});  
注意：  
  
1、一定要使用delete函数来删除关联数组的元素，这是唯一的方法。  
2、一定不要对关联数组使用内嵌函数push、pop、shift及splice，因为其元素位置是随机的。  
  
　  
  
八、列出数组的索引和值  
上面已经提到，keys()函数返回关联数组下标的列表，如：  
  
  
%fruit = ("apples", 9,  
"bananas", 23,  
"cherries", 11);  
@fruitsubs = keys(%fruits);  
这里，@fruitsubs被赋给apples、bananas、cherries构成的列表，再次提请注意，此列表没有次序，若想按字母顺序排列，可使用sort()函数。  
@fruitindexes = sort keys(%fruits);  
这样结果为("apples","bananas","cherries")。类似的，内嵌函数values()返回关联数组值的列表，如：  
  
  
%fruit = ("apples", 9,  
"bananas", 23,  
"cherries", 11);  
@fruitvalues = values(%fruits);  
这里，@fruitvalues可能的结果为(9,23.11)，次序可能不同。  
九、用关联数组循环  
前面已经出现过利用keys()函数的foreach循环语句，这种循环效率比较低，因为每返回一个下标，还得再去寻找其值，如：  
  
  
foreach $holder (keys(%records)){  
$record = $records{$holder};  
}  
  
Perl提供一种更有效的循环方式，使用内嵌函数each()，如：  
  
  
%records = ("Maris", 61, "Aaron", 755, "Young", 511);  
while (($holder, $record) = each(%records)) {  
# stuff goes here  
}  
  
each()函数每次返回一个双元素的列表，其第一个元素为下标，第二个元素为相应的值，最后返回一个空列表。  
注意：千万不要在each()循环中添加或删除元素，否则会产生不可预料的后果。  
十、用关联数组创建数据结构  
用关联数组可以模拟在其它高级语言中常见的多种数据结构，本节讲述如何用之实现：链表、结构和树。  
1、(单)链表  
链表是一种比较简单的数据结构，可以按一定的次序存贮值。每个元素含有两个域，一个是值，一个是引用（或称指针），指向链表中下一个元素。一个特殊的头指针指向链表的第一个元素。  
在Perl中，链表很容易用关联数组实现，因为一个元素的值可以作为下一个元素的索引。下例为按字母顺序排列的单词链表：  
  
  
%words = ("abel", "baker",  
"baker", "charlie",  
"charlie", "delta",  
"delta", "");  
$header = "abel";  
上例中，简单变量$header含有链表中第一个单词，它同时也是关联数组第一个元素的下标，其值baker又是下一个元素的下标，依此类推。  
下标为delta的最后一个元素的值为空串，表示链表的结束。  
在将要处理的数据个数未知或其随程序运行而增长的情况下，链表十分有用。下例用链表按字母次序输出一个文件中的单词。  
  
  
1 : #!/usr/local/bin/perl  
2 :  
3 : # initialize list to empty  
4 : $header = "";  
5 : while ($line = <STDIN>) {  
6 : # remove leading and trailing spaces  
7 : $line =~ s/^s+|s+$//g;  
8 : @words = split(/s+/, $line);  
9 : foreach $word (@words) {  
10: # remove closing punctuation, if any  
11: $word =~ s/[.,;:-]$//;  
12: # convert all words to lower case  
13: $word =~ tr/A-Z/a-z/;  
14: &add\_word\_to\_list($word);  
15: }  
16: }  
17: &print\_list;  
18:  
19: sub add\_word\_to\_list {  
20: local($word) = @\_;  
21: local($pointer);  
22:  
23: # if list is empty, add first item  
24: if ($header eq "") {  
25: $header = $word;  
26: $wordlist{$word} = "";  
27: return;  
28: }  
29: # if word identical to first element in list,  
30: # do nothing  
31: return if ($header eq $word);  
32: # see whether word should be the new  
33: # first word in the list  
34: if ($header gt $word) {  
35: $wordlist{$word} = $header;  
36: $header = $word;  
37: return;  
38: }  
39: # find place where word belongs  
40: $pointer = $header;  
41: while ($wordlist{$pointer} ne "" &&  
42: $wordlist{$pointer} lt $word) {  
43: $pointer = $wordlist{$pointer};  
44: }  
45: # if word already seen, do nothing  
46: return if ($word eq $wordlist{$pointer});  
47: $wordlist{$word} = $wordlist{$pointer};  
48: $wordlist{$pointer} = $word;  
49: }  
50:  
51: sub print\_list {  
52: local ($pointer);  
53: print ("Words in this file:n");  
54: $pointer = $header;  
55: while ($pointer ne "") {  
56: print ("$pointern");  
57: $pointer = $wordlist{$pointer};  
58: }  
59: }  
  
运行结果如下：  
  
  
Here are some words.  
Here are more words.  
Here are still more words.  
^D  
Words in this file:  
are  
here  
more  
some  
still  
words  
  
此程序分为三个部分：  
  
  
主程序：读取输入并转换到相应的格式。  
子程序：add\_word\_to\_list，建立排序单词链表。  
子程序：print\_list，输出单词链表  
第3~17行为主程序，第4行初始化链表，将表头变量$header设为空串，第5行起的循环每次读取一行输入，第7行去掉头、尾的空格，第8行将句子分割成单词。9~15行的内循环每次处理一个单词，如果该单词的最后一个字符是标点符号，就去掉。第13行把单词转换成全小写形式，第14行传递给子程序add\_word\_to\_list。  
子程序add\_word\_to\_list先在第24行处检查链表是否为空。如果是，第25行将单词赋给$header，26行创建链表第一个元素，存贮在关联数组%wordlist中。如果链表非空，37行检查第一个元素是否与该单词相同，如果相同，就立刻返回。下一步检查这一新单词是否应该为链表第一个元素，即其按字母顺序先于$header。如果是这样，则：  
1、创建一个新元素，下标为该新单词，其值为原第一个单词。  
2、该新单词赋给$header。  
如果该新单词不该为第一个元素，则40~44行利用局域变量$pointer寻找其合适的有效位置，41~44行循环到$wordlist{$pointer}大于或等于$word为止。接下来46行查看该单词是否已在链表中，如果在就返回，否则47~48行将其添加到链表中。首先47行创建新元素$wordlist{$word}，其值为$wordlist{$pointer}，这时$wordlist{$word}和$wordlist{$pointer}指向同一个单词。然后，48行将$wordlist{$pointer}的值赋为$word，即将$wordlist{$pointer}指向刚创建的新元素$wordlist{$word}。  
最后当处理完毕后，子程序print\_list()依次输出链表，局域变量$pointer含有正在输出的值，$wordlist{$pointer}为下一个要输出的值。  
注：一般不需要用链表来做这些工作，用sort()和keys()在关联数组中循环就足够了，如：  
  
  
foreach $word (sort keys(%wordlist)) {  
# print the sorted list, or whatever }  
  
但是，这里涉及的指针的概念在其它数据结构中很有意义。  
2、结构  
许多编程语言可以定义结构(structure)，即一组数据的集合。结构中的每个元素有其自己的名字，并通过该名字来访问。  
Perl不直接提供结构这种数据结构，但可以用关联数组来模拟。例如模拟C语言中如下的结构：  
  
  
struce{  
int field1;  
int field2;  
int field3; }mystructvar;  
  
我们要做的是定义一个含有三个元素的关联数组，下标分别为field1、field2、field3，如：  
  
  
%mystructvar = ("field1" , "" ,  
"field2" , "" ,  
"field3" , "" ,);  
  
像上面C语言的定义一样，这个关联数组%mystrctvar有三个元素，下标分别为field1、field2、field3，各元素初始值均为空串。对各元素的访问和赋值通过指定下标来进行，如：  
$mystructvar{"field1"} = 17;  
3、树  
另一个经常使用的数据结构是树。树与链表类似，但每个节点指向的元素多于一个。最简单的树是二叉树，每个节点指向另外两个元素，称为左子节点和右子节点（或称孩子），每个子节点又指向两个孙子节点，依此类推。  
注：此处所说的树像上述链表一样是单向的，每个节点指向其子节点，但子节点并不指向父节点。  
树的概念可以如下描述：  
  
  
因为每个子节点均为一个树，所以左/右子节点也称为左/右子树。（有时称左/右分支）  
第一个节点（不是任何节点的子节点的节点）称为树的根。  
没有孩子（子节点）的节点称为叶节点。  
有多种使用关联数组实现树结构的方法，最好的一种应该是：给子节点分别加上left和right以访问之。例如，alphaleft和alpharight指向alpha的左右子节点。下面是用此方法创建二叉树并遍历的例程：  
  
  
1 : #!/usr/local/bin/perl  
2 :  
3 : $rootname = "parent";  
4 : %tree = ("parentleft", "child1",  
5 : "parentright", "child2",  
6 : "child1left", "grandchild1",  
7 : "child1right", "grandchild2",  
8 : "child2left", "grandchild3",  
9 : "child2right", "grandchild4");  
10: # traverse tree, printing its elements  
11: &print\_tree($rootname);  
12:  
13: sub print\_tree {  
14: local ($nodename) = @\_;  
15: local ($leftchildname, $rightchildname);  
16:  
17: $leftchildname = $nodename . "left";  
18: $rightchildname = $nodename . "right";  
19: if ($tree{$leftchildname} ne "") {  
20: &print\_tree($tree{$leftchildname});  
21: }  
22: print ("$nodenamen");  
23: if ($tree{$rightchildname} ne "") {  
24: &print\_tree($tree{$rightchildname});  
25: }  
26: }  
  
结果输出如下：  
  
  
grandchild1  
child1  
grandchild2  
parent  
grandchild3  
child2  
grandchild4  
  
　  
  
第七节：  
  
一、引用简介  
二、使用引用  
三、使用反斜线()操作符  
四、引用和数组  
五、多维数组  
六、子程序的引用  
子程序模板  
七、数组与子程序  
八、文件句柄的引用  
  
一、引用简介  
引用就是指针，可以指向变量、数组、哈希表（也叫关联数组）甚至子程序。Pascal或C程序员应该对引用（即指针）的概念很熟悉，引用就是某值的地址，对其的使用则取决于程序员和语言的规定。在Perl中，可以把引用称为指针，二者是通用的，无差别的。引用在创建复杂数据方面十分有用。  
Perl5中的两种引用类型为硬引用和符号引用。符号引用含有变量的名字，它对运行时创建变量名并定位很有用，基本上，符号引用就象文件名或UNIX系统中的软链接。而硬引用则象文件系统中的硬链接。  
Perl4只允许符号引用，给使用造成一些困难。例如，只允许通过名字对包的符号名哈希表（名为\_main{}）建立索引。Perl5则允许数据的硬引用，方便多了。  
硬引用跟踪引用的计数，当其数为零时，Perl自动将被引用的项目释放，如果该项目是对象，则析构释放到内存池中。Perl本身就是个面向对象的语言，因为Perl中的任何东西都是对象，包和模块使得对象更易于使用。  
简单变量的硬引用很简单，对于非简单变量的引用，你必须显式地解除引用并告诉其应如何做，详见《第 章Perl中的面向对象编程》。  
二、使用引用  
本章中，简单变量指像$pointer这样的变量，$pointer仅含一个数据项，其可以为数字、字符串或地址。  
任何简单变量均可保存硬引用。因为数组和哈希表含有多个简单变量，所以可以建立多种组合而成的复杂的数据结构，如数组的数组、哈希表的数组、子程序的哈希表等等。只要你理解其实只是在用简单变量在工作，就应该可以正确的在最复杂的结构中正确地解除引用。  
首先来看一些基本要点。  
如果$pointer的值为一个数组的指针，则通过形式@$pointer来访问数组中的元素。形式@$pointer的意义为“取出$pointer中的地址值当作数组使用”。类似的，%$pointer为指向哈希表中第一个元素的引用。  
有多种构建引用的方法，几乎可以对任何数据建立引用，如数组、简单变量、子程序、文件句柄，以及--C程序员会感兴趣的--引用。Perl使你有能力写出把自己都搞糊涂的极其复杂的代码。http://www.linuxeden.com/forum/images/smilies/default/smile.gif  
下面看看Perl中创建和使用引用的方法。  
三、使用反斜线()操作符  
反斜线操作符与C语言中传递地址的操作符&功能类似。一般是用创建变量又一个新的引用。下面为创建简单变量的引用的例子：  
$variavle = 22;  
$pointer = $variable;  
$ice = "jello";  
$iceprt = $ice;  
引用$pointer指向存有$variable值的位置，引用$iceptr指向"jello"。即使最初的引用$variable销毁了，仍然可以通过$pointer访问该值，这是一个硬引用，所以必须同时销毁$pointer和$variable以便该空间释放到内存池中。  
在上面的例子中，引用变量$pointer存的是$variable的地址，而不是值本身，要获得值，形式为两个$符号，如下：  
  
  
#!/usr/bin/perl  
$value = 10;  
$pointer = $value;  
printf "n Pointer Address $pointer of $value n";  
printf "n What Pointer \*($pointer) points to $$pointern";  
  
结果输出如下：  
  
  
Pointer Address SCALAR(0x806c520) of 10  
What Pointer \*(SCALAR(0x806c520)) points to 10  
  
每次运行，输出结果中的地址会有所改变，但可以看到$pointer给出地址，而$$pointer给出$variable的值。  
看一下地址的显示，SCALAR后面一串十六进制，SCALAR说明该地址指向简单变量（即标量），后面的数字是实际存贮值的地址。  
注意：指针就是地址，通过指针可以访问该地址处存贮的数据。如果指针指向了无效的地址，就会得到不正确的数据。通常情况下，Perl会返回NULL值，但不该依赖于此，一定要在程序中把所有的指针正确地初始化，指向有效的数据项。