# Using References to Create Complex Structures

The array and hash composers



# Anonymous array composer

- You can create a hard reference to an ***anonymous array*** using square brackets.

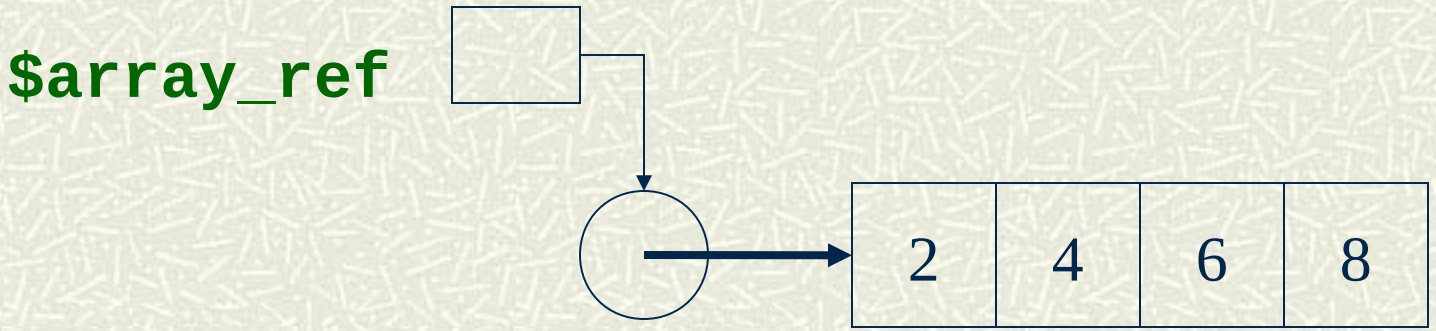
```
$array_ref = [ 2,4,6,8 ];
```

  creates an unnamed array with the values 2,4,6,8, and creates a reference to the array and assigns it to **`$array_ref`**.

- The operator **`[ ]`** is called the ***anonymous array composer.***

# Reference to an anonymous array

- The picture below shows what gets created by the preceding anonymous array composer.




- It shows that **`$array_ref`** contains a reference to an array, represented by the circular object.

# Anonymous arrays

- In general, `[ ]` creates a reference to a list:

  `[ list of scalars ]`

  is a reference to an unnamed array consisting of the list of values inside the brackets.
- The values inside must be scalars, but since references themselves are scalars, this is how you can build arbitrarily complex structures – ***just make it a list of references to other "thingies."*** *(A "thingy" is the term Larry Wall uses for things like variables and literals. )*

# Creating multi-dimensional arrays

- You can create two-dimensional array for example:

```
$ref2by3array = [ [1,4], [2,5], [3,6] ];
```

This is a reference to an array of 3 references to 2-element arrays. It is more clearly written:

```
$ref2by3array  = [
                    [1, 4],
                    [2, 5],
                    [3, 6]
                  ];
```

# Anonymous arrays (cont'd.)

- You can also create a 2D array by making anonymous 1D arrays inside of a ***list literal***:

```
@matrix = ( [1,2], [3,4], [5,6] );
```

in which case `@matrix` is an array of 3 refs to anonymous arrays of 2 elements each. This is because it is of the form

It is better style to write this as

```
@matrix = (
            [1,2],
            [3,4],
            [5,6]
          );
```

# More on arrays

- You can also create structures like

```
$studref =  [
              $name,
              [$hwk1, $hwk2, $hwk3]
            ];
```

  which can be thought of as an array with a name and a list of grades associated with the name.

- Soon you'll see how to fill them, use them, print them and so on.

# Some warnings

- Do not confuse the square bracket array composer with the subscript operator or the slice operator. These square brackets appear where expressions are expected, such as on the right hand side of an assignment operator. When they appear after an identifier, they are the subscript operator.
- Remember too that you are creating ***references***, not actual arrays, and that you must ***dereference*** them to use them.

# Building and using lists of lists

- Unlike most languages, Perl does not require that you specify the size of an array. The rows don't have to have the same number of elements either. If you reference an element that is not in the list yet, Perl extends the list and inserts it. This makes building complex structures very simple.
- The next few slides illustrate how.

# Building lists of lists

- To grow a list of lists by reading each row from a line of a file:

```
while (<>) {
    @temp = split;  # split $_ into list
    # now create a ref to the list and
    # append this ref to @ListOfLists
    push @ListOfLists, [ @temp ];
    }
```

# Using lists of lists

- You can print the preceding array, one row per line with

```
for ($i = 0; $i < @ListOfLists; $i++ ) {
    print "@{$ListOfLists[$i]} \n";
}
```

- The expression `{$ListOfLists[$i]}` is actually a ***BLOCK*** of code, since it is of the form `{…}`. It returns a value, namely the last expression evaluated in the block. This expression is the value of the array component `$ListOfLists[$i]`, which is a reference to a list at index `$i` in `$ListOfLists`. The argument to `print` is therefore of the form *@array_ref*, which evaluates to the list itself, which `print` prints.

# Accessing the individual list elements

- We can also access the elements individually. If we wanted to print every other element in each row, starting with the first, we could use

```
for ($i = 0; $i < @ListOfLists; $i++ ) {
    for ($j=0; $j < @{$ListOfLists[$i]}; $j+=2) {
        print "${$ListOfLists[$i]}[$j]  ";
    }
    print "\n";
}
```

# The `->` operator

- The `->` operator is the ***member dereferencing operator*** (like the one in C, but it is more general.) It can be used to access members of referenced arrays:

```
$ref2by3 = [ [1,4], [2,5], [3,6] ];
print  $ref2by3->[0]->[1];  #prints 4
print  ${$ref2by3->[0]}[1]; #prints 4
print $$ref2by3[0][1];      #prints 4
print $ref2by3->[0][1];     #prints 4
print ${${$ref2by3}[0]}[1]; #prints 4
```

# The Anonymous Hash Composer

- You can create a hard reference to an anonymous hash using the ***anonymous hash composer*** **`{ }`**.

```
$pairs = {
            'tea'   => 'two',
            'me'    => 'you',
            'you'   => 'me'
};
```

creates an unnamed hash with the three key-value pairs and assigns a reference to its thingy to **`$pairs`**.

# The `->` Operator in Hashes

- Using the preceding definition of the `$pairs` reference:

```
$pairs->{tea} = 'Three';
```

  is short for these, which are equivalent:

```
$$pairs{tea} = 'Three';
${$pairs}{tea} = 'Three';
```

- The way to read this is, `$pairs` is a reference to an anonymous hash, which either already has the key `'tea'` and is getting a new value `'Three'`, or the hash did not have the key `'tea'` and a new pair is added to it.

# Hashes and Arrays

- The anonymous array composer allows us to create hashes whose values are not just simple scalars, but are references to lists, as in the following example of a hash named **%Cities**:

```
%Cities =  (
   'Italy'  =>
       [ 'Firenze', 'Padua', 'Milano' ],
   'Japan' =>
       ['Nagano', 'Tokyo', 'Kyoto' ],
   'Canada' =>
       ['Victoria', 'Banff', 'Toronto']
);
```

# More about multidimensional arrays

- Given the list of references to lists

```
@zoo = (
        ["mongoose","capybara"],
        ["tiger","lion", "ocelot"],
        ["boa","cobra"]
      );
print $zoo[1][2];      # prints ocelot.
```

is a shorthand for

```
print $zoo[1]->[2];    #same thing
```

which is really a shorthand for

```
print ${$zoo[1]}[2];   # same thing
```

# Lists of lists

- If we changed the preceding example to use `[]` instead of `()`:

```
$zoo_ref = [
           ["mongoose","capybara"],
           ["tiger","lion", "ocelot"],
           ["boa","cobra"]
           ];
```

we'd have to print using:

```
print $zoo_ref->[1][2]; # prints ocelot.
```

or `print ${${$zoo_ref}[1]}[2];`

or `print $zoo_ref->[1]->[2];`

# Hashes containing arrays

- There are some obvious applications of hashes of array references

```
%CSci132_grades = (
        Obdoli   => [ 78, 65, 94 ],
        Palermo  => [ 84, 90, 74 ],
        Sanchez  => [ 70, 76, 88 ]
        );
```

would be a way to store grades for students with names given as keys.

# Hashes containing arrays (cont'd.)

- To access a grade, I could use

```
print $CSci132_grades{Sanchez}[1];
```

  which prints the 2nd grade of Sanchez. More generally,

```
print $CSci132_grades{$name}[$exam];
```

  prints the grade of the person whose name and exam number are given.

- Note that the hash is searched by name, making this a very elegant method of accessing records. You would otherwise have to write a search routine, if it were an ordinary array.

# Hashes as records or structures

- Hashes are a very convenient way to implement records (and objects), because the field names can be keys and the field values, their values. And hashes can be arbitrarily complex. Some fields can be arrays, some constants, some methods.
- Also, you do not need to enclose strings in quote marks when they are used as hash keys.
- The next slide illustrates.

# Hashes as records (2)

```
($n, $elmt, $res ) = split(" ",<STDIN>);
%atom = (
   NUM       => $n,
   ELEMENT   => $elmt,
   RESIDUE   => $res
   );
```

- The keys serve as names of "fields"; the above code could be used in a loop, in which a table of atom records is filled.

# Hashes as records (3)

```
$atom_ref = {
   NUM        => $n,
   ELEMENT    => $elmt,
   RESIDUE    => $res
   };
```

- Instead of creating a hash, we can create a reference to a hash, and this can be assigned to an array:

```
push((@atoms, $atom_ref)
```

# Hash of complex types

```
$obj_ref = {
   NAME          => $namestring,
   SCORES        => [@list_of_numbers ],
   PERSONAL_DATA => {%list_of_keys_values},
   SAVE_METHOD   => \&save_function,
   RESTORE_METHOD => \&restore_function
   };
```

is an object that has references to string, list, a hash, and some functions for saving and retrieving the saved data. This is conceptual – you have to replace the values with actual data.

# Hash of complex type (cont'd.)

- To access the various components of this object, you would use statements such as

```
print $obj_ref->{NAME};
print "Age: $obj_ref->{PERSONAL_DATA}->{AGE} \n";
for $i ( @{$obj_ref->{SCORES}} ) {
   print " Score $n : $i \n";
   $n++;
   }
&{$obj_ref->{SAVE_METHOD}} ($filename);
```

# Creating complex structures

- The preceding structure can be read piece by piece from a file with code such as

```
$obj_ref = {};              # empty hash
chop( $name  =  <> );       # get name from input
$obj_ref{NAME} = $name; # assign to hash NAME
# read pairs of the form key value from input
# and add to PERSONAL_DATA hash
while (<>) {
    ($key, $value) = split /[\s]+/;
    $obj_ref{PERSONAL_DATA}{$key} = $value;
  }
```

# Subroutine references

- References to subroutines are useful for creating dispatch tables.
  Suppose **show**, **help**, and **quit** are subs. Given:

```
my %dispatch_table = (
        show => \&show,
        help => \&help,
        quit => \&quit
        );
```

- If **$key** contains a user supplied name matching a key then

```
$dispatch_table{$key}->()
```

- invokes the matching function.

# Anonymous subroutine composer

- You can also create references to ***anonymous subroutines*** by using the sub operator without a name after it:

```
$coderef = sub { print "Stop!\n" };
```

  creates a reference to a piece of code that prints "Stop!".

  The "`;`" after the sub body is necessary because of the assignment statement's syntax. Ordinarily a sub declaration has no terminating semicolon.

- ***Anonymous subroutines are called closures in Perl***. Closures have the ability to carry their lexical environments around with them. See the code example.