Data Structures

- A *data structure* is how a computer stores and indexes pieces of information and connections between them.
- Data structures may be designed for storage efficiency, speed of general or common operations, speed of certain specific operations, understandability of use, or understandability of implementation.
  - Typically one is emphasized at the expense of the others.
  - For example, hashes are designed for speed of a specific operation (lookup). Speed of other operations, storage space, and understandability (particularly of implementation) are not emphasized.
- Perl natively supports two major data structures (besides "scalar"): arrays and hashes.
- As we have seen, you cannot create multidimensional arrays, hashes with keys that point to arrays, or similar structures, because arrays and hashes flatten instead of nest.
- However, with the addition of *references*, these types of data structures (as well as many others) can be implemented.

References

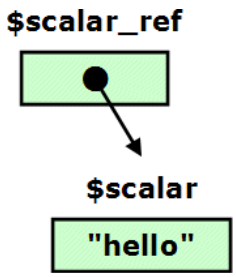
- A reference is similar to a *pointer*.
- However, in most languages, a pointer is simply a memory address that can be used as if it pointed to anything (directly or through *casting*).
  - Some languages store some additional information along with pointers, such as a the *type* of variable or structure being pointed at.
- Perl's references store much more information, and are strongly typed.
- References are another type of scalar value and are stored in scalar variables.
  - Therefore, they can be stored in arrays or as hash values.
- The variable being referenced (pointed to) by a reference is called the *referent*.

Creating a Reference

- Create a reference to another variable (a scalar, a hash, or an array) using the reference operator, spelled "backslash" ("\"). For example, the code

```
my $scalar = "hello";
my $scalar_ref = \$scalar;
```

  creates the following structures in memory:

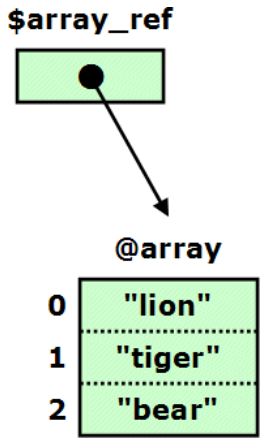


  - After these operations are complete, **`$scalar_ref`** contains a reference to **`$scalar`**. If you print **`$scalar_ref`** or otherwise force it into string context, you get a string like "SCALAR(0x1291fc)" which is the type of the thing being referenced and its memory address.
    - All variables that contain references return true in Boolean context. This is useful for checking whether a reference has been defined, for instance by a subroutine that returns a reference or **`undef`** in the case of failure.
      - The fact that a defined reference is always "true" tells you **nothing** about the truth of the referent!
    - If you force a reference into numeric context, it evaluates as the memory address. Unlike in C, this is never useful, because Perl has no facility for "pointer arithmetic".

- You can create a references to an array (called an *arrayref*) with code like this:

```
my @array = ("lion", "tiger", "bear");
my $array_ref = \@array;
```

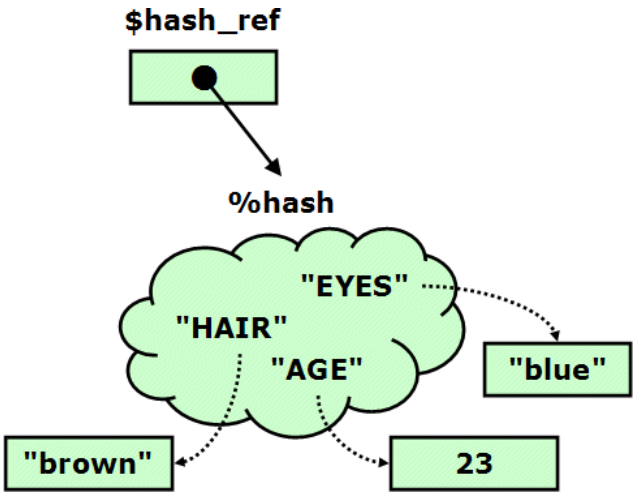
  which generates memory structures like this:



- You can create a references to an hash (called a *hashref*) with code like this:

```
my %hash = (HAIR => "brown", EYES => "blue", AGE => 23);
my $hash_ref = \%hash;
```

  which generates memory structures like this:

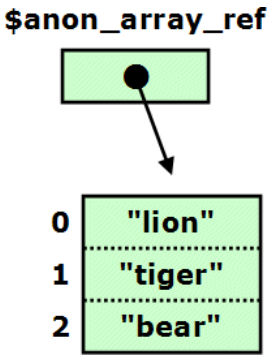


- You can make references to references by taking a reference of a reference or by using two (or more) backslashes. This is rarely useful.

- You can create a reference to an *anonymous* array or hash as well. This means you create a reference that points to data in memory, but that data does not correspond to a plain variable. In other words, there is no way to get to it without going through the reference.
  - The term "*anonymous arrayref*" is a shortened version of "reference to an anonymous array".
  - This code create an anonymous arrayref:

```
my $anon_array_ref = [ "lion", "tiger", "bear" ];
```

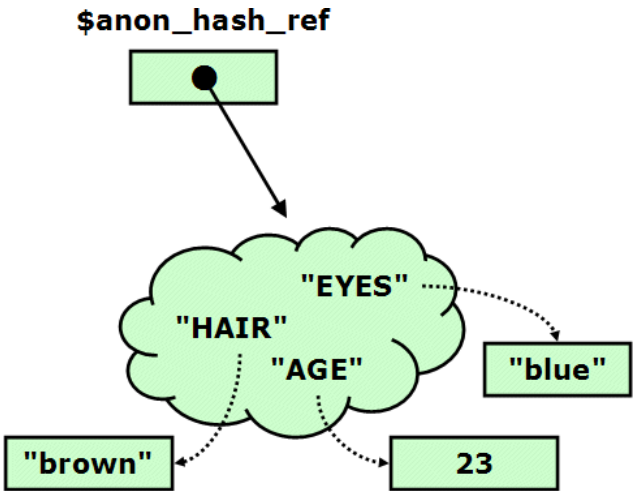
which looks like this:



- This code create an anonymous hashref:

```
my $anon_hash_ref = { HAIR => "brown", EYES => "blue", AGE => 23 };
```

which looks like this:



  - Yet another use for braces!
- You **cannot** create an anonymous arrayref using code like this:

```
$array_ref = \($var1, $var2, $var3);
```

  - This actually distributes the reference operator, and is equivalent to

```
$array_ref = (\$var1, \$var2, \$var3);
```

  - This is a scalar assignment; hence the commas are comma operators, not array element separators.
  - This is rarely if ever useful.
  - You cannot create anonymous hashrefs this way either.
  - The following code **will** work:

```
$array_ref = [ $var1, $var2, $var3 ];
```

  - Similar code works to create hashrefs.

- Functionally, there is no difference between anonymous references and references to variables.

Dereferencing a Reference

- To *dereference* a reference variable (i.e. to access the data it is pointing to), prepend the appropriate sigil:

```
@keys = keys %$hash_ref;
push @$array_ref, "value";
print $$scalar_ref, "\n";
```

  - You can also enclose everything after the dereferencing sigil in braces for *disambiguation*:

```
@keys = keys %{$hash_ref};
push @{$array_ref}, "value";
print ${$scalar_ref}, "\n";
```

    - Disambiguation is not necessary for simple dereferences like this, but consider an array of references. Each element such as `$array[7]` is a reference, but it is not possible to simply dereference that element with `$$array[7]`. Instead you need to use `${$array[7]}`.

  - Dereferencing a reference to a reference to a scalar produces a reference to a scalar. You can "double-dereference" a reference to a reference to a scalar to get a scalar:

```
1 my $scalar = "blah";
2 my $scalar_ref = \$scalar;
3 my $dbl_ref1 = \$scalar_ref;
4 my $dbl_ref2 = \\$scalar;
5
6 print $scalar, "\n";
7 print $$scalar_ref, "\n";
8 print $$$dbl_ref1, "\n";
9 print $$$dbl_ref2, "\n";
```

    - That's one dollar sign for each "dereference a scalar ref", plus one for the variable being accessed.
    - Output:

```
blah
blah
blah
blah
```

- To dereference and index into an arrayref or hashref in order to get or set values of elements, use a *thin arrow* (spelled "`->`") in between the ref variable and the brackets or braces:

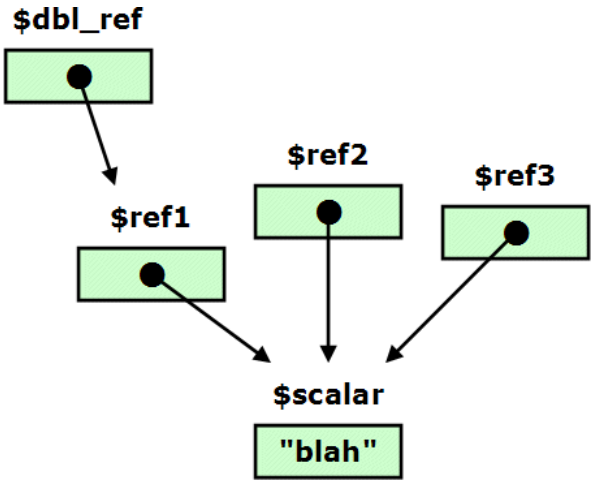
```
$hash_ref->{GREETING} = "hello";
print $array_ref->[6];
```

Multiplicity

- If a reference variable is copied or referenced, or two references are taken to the same variable, modifying either referent (or the original variable) affects the other referent (and the original variable). Consider the code

```
 1 my $scalar = "blah";
 2 my $ref1 = \$scalar;  # create a ref
 3 my $ref2 = \$scalar;  # create another ref to the same thing
 4 my $ref3 = $ref2;     # copy a ref, which creates another ref to the same thing
 5 my $dbl_ref = \$ref1; # create a ref to a ref
 6
 7 print "              Before: ($scalar/$$ref1/$$ref2/$$ref3/$$$dbl_ref)\n";
 8 $scalar .= "!";
 9 print "   Modifying \$scalar: ($scalar/$$ref1/$$ref2/$$ref3/$$$dbl_ref)\n";
10 $$ref1 .= "?";
11 print "    Modifying \$\$ref1: ($scalar/$$ref1/$$ref2/$$ref3/$$$dbl_ref)\n";
12 $$$dbl_ref = uc $$$dbl_ref;
13 print "Modifying \$\$\$dbl_ref: ($scalar/$$ref1/$$ref2/$$ref3/$$$dbl_ref)\n";
```

Lines 1-5 of this program create the following structures in memory:



- Note that there is no functional difference between a reference that was copied from **`$ref1`** (in this case, **`$ref2`**), and one that was constructed in the same way as **`$ref1`** (in this case, **`$ref3`**).

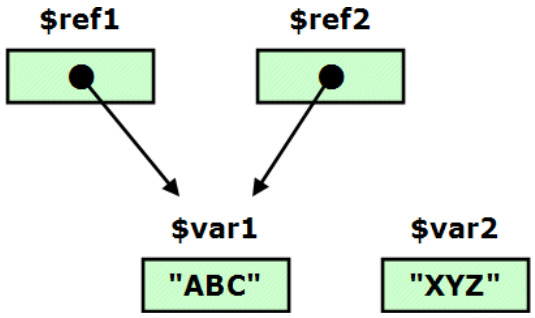
The program outputs:

```
            Before: (blah/blah/blah/blah/blah)
   Modifying $scalar: (blah!/blah!/blah!/blah!/blah!)
    Modifying $$ref1: (blah!?/blah!?/blah!?/blah!?/blah!?)
Modifying $$$dbl_ref: (BLAH!?/BLAH!?/BLAH!?/BLAH!?/BLAH!?)
```

- Notice in this example, the **referent** (or double-referent) or the original variable was being modified.
- If instead of modifying the **referent** you modify the **reference**, the other referent will be unaffected, because all you are doing is making the second reference point someplace else:
  - For example, consider the code

```
 1 my $var1 = "ABC";
 2 my $var2 = "XYZ";
 3
 4 my $ref1 = \$var1;
 5 my $ref2 = $ref1;
 6
 7 print "\$ref1 and \$ref2 point to the same place;\n";
 8 print "changes to \$var1, \$\$ref1, or \$\$ref2 affect each other:\n";
 9
10 print "  \$var1=$var1; \$\$ref1=$$ref1, \$\$ref2=$$ref2\n";
11 print "Changing \$var1:\n";
12 $var1 = "YEE";
13 print "  \$var1=$var1; \$\$ref1=$$ref1, \$\$ref2=$$ref2\n\n";
14
15 print "Now we will modify \$ref2 (NOT \$\$ref2):\n";
16 $ref2 = \$var2;
17 print "  \$var1=$var1; \$\$ref1=$$ref1, \$\$ref2=$$ref2\n\n";
18
19 print "Changes to \$var1 or \$\$ref1 no longer affect \$\$ref2, or vice-versa:\n";
20 $var1 = "Blah";
21 print "  \$var1=$var1; \$\$ref1=$$ref1, \$\$ref2=$$ref2\n";
```

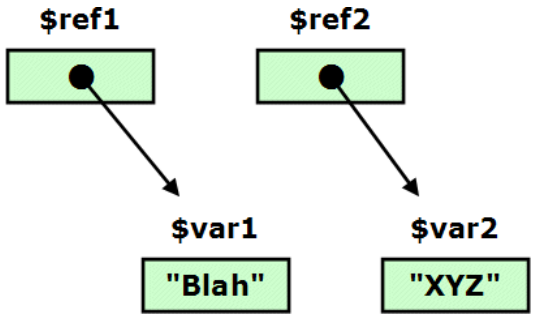
The memory structures created by lines 1-5 are:



The code outputs

```
$ref1 and $ref2 point to the same place;
changes to $var1, $$ref1, or $$ref2 affect each other:
  $var1=ABC; $$ref1=ABC, $$ref2=ABC
Changing $var1:
  $var1=YEE; $$ref1=YEE, $$ref2=YEE

Now we will modify $ref2 (NOT $$ref2):
  $var1=YEE; $$ref1=YEE, $$ref2=XYZ

Changes to $var1 or $$ref1 no longer affect $$ref2, or vice-versa:
  $var1=Blah; $$ref1=Blah, $$ref2=XYZ
```

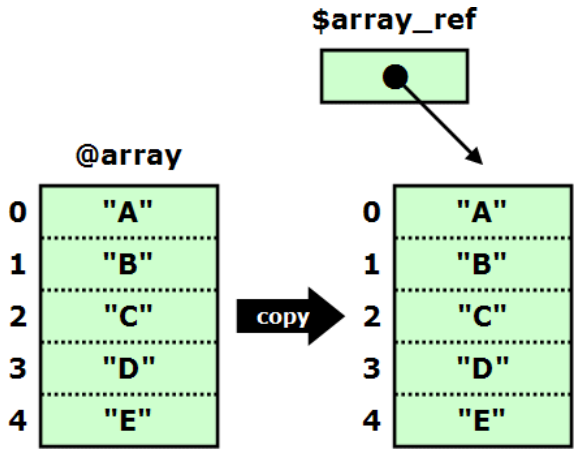
After the code executes, the memory structures look like this:



- You can also create an anonymous hashref or arrayref that includes variables using code like:
  `$array_ref = [ @array ];`
  or
  `$array_ref = [ $var1, $var2, $var3 ];`
  This **does** make copies of those variables for its elements because assignment is a copy operation. The reference here is to the array structure in memory, not to the individual elements. Modifying the elements of the anonymous array will not affect the variables they were copied from. So the code

```
1 my @array = ("A", "B", "C", "D", "E");
2 my $array_ref = [@array];
3
4 $array[2] = "NEW";
5 push @{$array_ref}, "PUSHED";
6
7 print "   array = (@array)\n";
8 print "arrayref = [@{$array_ref}]\n";
```

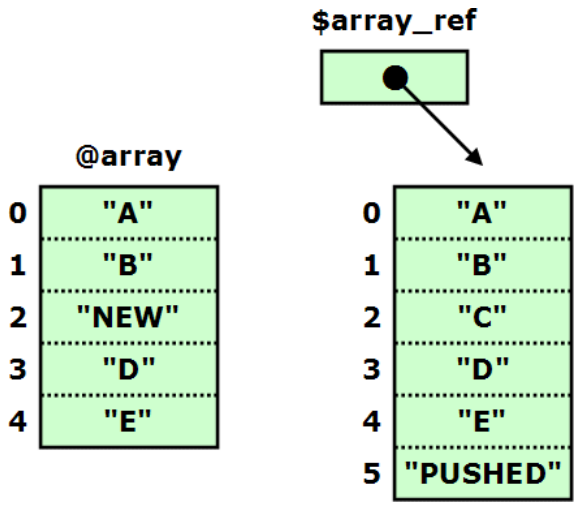
creates the memory structures



and outputs

```
   array = (A B NEW D E)
arrayref = [A B C D E PUSHED]
```

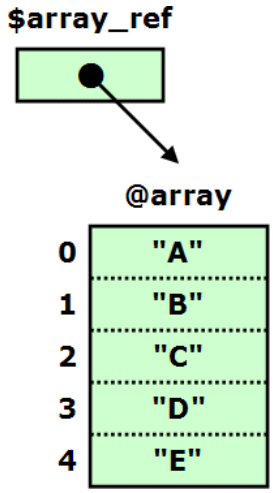
- Changes made to the reference and the original array **do not affect** each other.
- The final memory structures look like this:



- As opposed to code like
  `$array_ref = \@array;`
  which makes a reference to the whole `@array` structure. Any changes to `@array`, including adding or removing elements or modifying existing elements, will be reflected in `$array_ref` (and vice-versa). Consider the code

```
1 my @array = ("A", "B", "C", "D", "E");
2 my $array_ref = \@array;
3
4 $array[2] = "NEW";
5 push @{$array_ref}, "PUSHED";
6
7 print "   array = (@array)\n";
8 print "arrayref = [@{$array_ref}]\n";
```

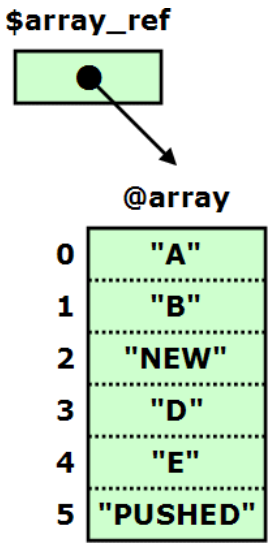
This code is identical to the code above except in the manner the arrayref is constructed. The resulting memory structure is the familiar:



And the code outputs

```
   array = (A B NEW D E PUSHED)
arrayref = [A B NEW D E PUSHED]
```

- Changes made to the reference and the referent **do affect** each other.
- After the code is executed, the memory structure is:



- Code like
  ```
  $array_ref1 = [ @array ];
  $array_ref2 = $array_ref1;
  ```
  copies the elements in `@array` into an anonymous array that `$array_ref1` points to. Changes to `@array` and `$array_ref1` are independent. However, `$array_ref2` is a copy of `$array_ref1`; or in other words, they point to the same thing (the anonymous array). Changes made to the array referenced by `$array_ref1` **will** be reflected in the array referenced by `$array_ref2` (because they are the same array).
- Note that all of the points made here about array refs apply equally to hashrefs (and scalar refs).

Reference Compatibility

- In Perl, references are strongly typed. Despite that fact that plain hashes and arrays are more-or-less compatible, you cannot use a hashref as if it were an arrayref (at least not without jumping through a lot of syntactic hoops).
  - Attempting to do so results in an error message. So the code

    ```
    my $array_ref = [ 1, 2, 3 ];
    print keys %{$array_ref};
    ```

    outputs

    ```
    Can't coerce array into hash at incompatible_refs.pl line 2.
    ```
- It should go without saying that you cannot use a non-reference (i.e. a plain scalar) or a undefined scalar as a reference without incurring errors.
- You **can** assign an arrayref onto a hashref, because this clobbers whatever was there before.
- You can also assign an array or a dereferenced arrayref to a hash or dereferenced hashref, or vice versa. So the following are allowed, and work as expected:
  - `%hash = @array` (plain old assignment)
  - `@array = %hash` (also plain assignment)
  - `%hash = @{$arrayref}`
  - `@array = %{$hashref}`
  - `%{$hashref} = @array`
  - `@{$arrayref} = %hash`
  - `%{$hashref} = @{$arrayref}`
  - `@{$arrayref} = %{$hashref}`
    - These all work because by the time the assignment happens, there are no references involved—all of the references were dereferenced.
- In case you need to check what type of reference you are dealing with (very often the case when checking arguments to a subroutine), use the `ref` function.
  - `ref` returns an empty string if passed a plain scalar and the type of reference if passed another type of reference.
  - Example:

    ```
     1 my $scalar = 1;
     2 my @array = (1, 2, 3);
     3 my %hash = (A=>1, B=>2, C=>3);
     4
     5 my @variables = (
     6   $scalar,
     7   \$scalar,
     8   \\$scalar,
     9   \@array,
    10   \%hash,
    11   \\@array
    12 );
    13
    14 foreach my $possible_ref (@variables) {
    15   my $ref_type = ref $possible_ref;
    16   print "ref returned \"$ref_type\"\n";
    17   if ($ref_type eq "REF") {
    18     my $ref_of_ref = ref ${$possible_ref};
    19     print "  REF TO $ref_of_ref REF\n";
    20   }
    21 }
    ```

outputs

```
ref returned ""
ref returned "SCALAR"
ref returned "REF"
  REF TO SCALAR REF
ref returned "ARRAY"
ref returned "HASH"
ref returned "REF"
  REF TO ARRAY REF
```

Using Arrayrefs and Hashrefs

- Using the rules above, most operations on a simple arrayref or hashref should be intuitive. Here are the most common ones:

Arrayref Operations

Operation	Array	Arrayref
create	`@array = (1, 2, 3)`	`$array_ref = [1, 2, 3]` `$array_ref = \@array`
access whole array	`@array`	`@{$array_ref}`
print whole array	`print "@array"`	`print "@{$array}"`
print element	`print $array[7]`	`print $array_ref->[7]`
set element	`$array[7] = "hello"`	`$array_ref->[7] = "hello"`
push element	`push @array, $value`	`push @{$array_ref}, $value`
pop element	`$value = pop @array`	`$value = pop @{$array_ref}`
count array elements	`scalar @array`	`scalar @{$array_ref}`
get highest index	`$#array`	`$#{$array_ref}`
sort array	`@array = sort @array`	`@{$array_ref} = sort @{$array_ref}`
array slice	`@array[0 .. 2]`	`@{$array_ref}[0 .. 2]`

- Note: you can check whether an array is empty by evaluating `@array` (i.e. the number of elements) in Boolean context. To check whether an array**ref** is empty, remember you must evaluate `@{$arrayref}` in Boolean context.
  - Evaluating `$arrayref` in Boolean context will not work. Recall that references are always true in Boolean context unless they are undefined (which is **not** the same as being defined but referencing an empty array).
- Discussion question: what does each of the following statements do?
  - `$foo = [1,2,3]; @foo = (1,2,3); $foo = (1,2,3); @foo = [1,2,3];`

Hashref Operations

Operation	hash	hashref
create	`%hash = (A=>1, B=>2, C=>3)`	`$hash_ref = {A=>1, B=>2, C=>3}` `$hash_ref = \%hash`
access whole hash	`%hash`	`%{$hash_ref}`
print element	`print $hash{A}`	`print $hash_ref->{A}`
set element	`$hash{A} = "hello"`	`$hash_ref->{A} = "hello"`
access hash keys	`keys %hash`	`keys %{$hash_ref}`
delete element	`delete $hash{A}`	`delete $hash_ref->{A}`
test for element	`exists $hash{A}`	`exists $hash_ref->{A}`
hash slice	`@hash{'A', 'B'}`	`@{$hash_ref}{'A', 'B'}`

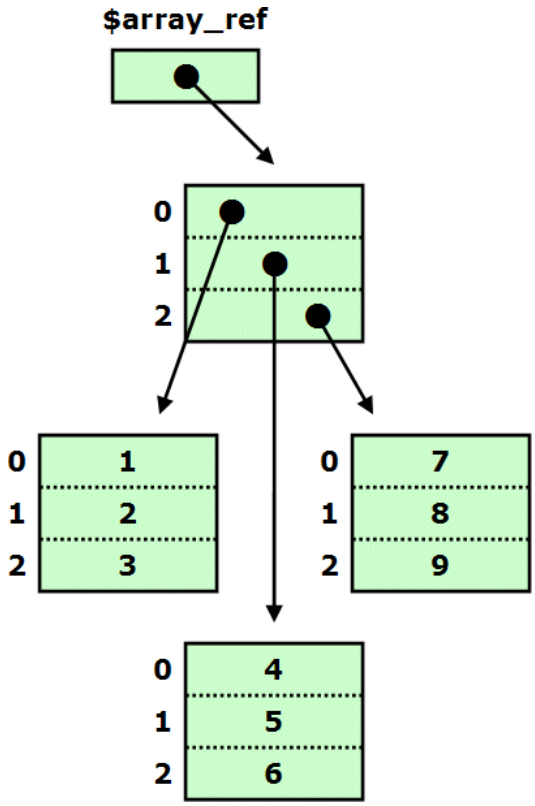
- Note: the same rules for when you need quotes in a hash lookup apply when dereferencing/looking up into a hashref.
- Remember that hash **values** can be any type of scalar, but anything used as a **key** will be forced into string context. Don't use references as hash keys.

Multidimensional Arrays

- As mentioned repeatedly, there is no such thing as native multi-dimensional arrays in Perl because arrays are flattened.
- But creating an array or arrayref with arrayrefs as elements allows us to get there.
  - There is no functional difference between using an array and an arrayref as the base of a multi-dimensional array. However, I find that using an arrayref as the base variable simplifies the syntax and therefore clarifies the resulting code. I will use this convention in all examples.
  - In general, if your program is going to use both arrays and arrayrefs (and/or hashes and hashrefs) it will probably be easier and clearer to use references exclusively, except for plain scalar variables.
- You can define an 2-D array in one step like this:

```
1 my $array_ref = [
2                   [ 1, 2, 3 ],
3                   [ 4, 5, 6 ],
4                   [ 7, 8, 9 ]
5                 ];
6
7 print Dumper $array_ref;
```

which create the memory structure



and outputs

```
$VAR1 = [
          [
            1,
            2,
            3
          ],
          [
            4,
            5,
            6
          ],
          [
            7,
            8,
            9
          ]
        ];
```

or build it step by step, like this:

```
my $array_ref = [ ]; # start with an empty array ref
foreach my $row ( 0 .. 3 ) {
  foreach my $col ( 0 .. 2 ) {
    $array_ref->[$row]->[$col] = "R$row/C$col";
  }
}

print Dumper $array_ref;
```

which outputs

```
$VAR1 = [
          [
            'R0/C0',
            'R0/C1',
            'R0/C2'
          ],
          [
            'R1/C0',
            'R1/C1',
            'R1/C2'
          ],
          [
            'R2/C0',
            'R2/C1',
            'R2/C2'
          ],
          [
            'R3/C0',
            'R3/C1',
            'R3/C2'
          ]
        ];
```

- Some notes:
  - Remember that a "2-dimensional array" is really just an arrayref of arrayrefs.
  - It doesn't really matter whether you consider the first level the "row" or "x" and the second level the "column" or "y", or vice versa. Just be consistent.
  - Note the two-level dereference/index structure in the second example:
    `$array->[$row]->[$col]`
    - You may also see this with only the first arrow:
      `$array->[$row][$col]`
    - Either is fine.
  - Intermediate elements are *auto-vivified* when any of their sub-elements are referenced (even for reading).
  - Declaring `$array` as an empty arrayref to start is not required (i.e. "`my $arr = [ ];`" instead of just "`my $arr;`"). The only requirement is that it be an arrayref or undefined when array operations are executed.
    - However, I find this a helpful reminder of what the variable will be used for later.

- To get the highest index of the first level, or to treat the first level as a plain array (for example to get the number of elements or to **push** onto it), use a regular dereference as shown in the table above:
  `$#{$array_ref}`
  or
  `@{$array_ref}`
  - To get the highest index of one of the sub-levels, or to treat it as a plain array, dereference **one level** and then perform the dereference/operation:
    `$#{$array_ref->[0]}`
    or
    `@{$array_ref->[0]}`
  - NOTE: The subarrays do **not** have to be the same size and there is no mechanism for enforcing this (as there is in C or Pascal). If you want your array to be *orthogonal* you must check this yourself.
    - Discussion question: how would you check this?

The Substitution Principle

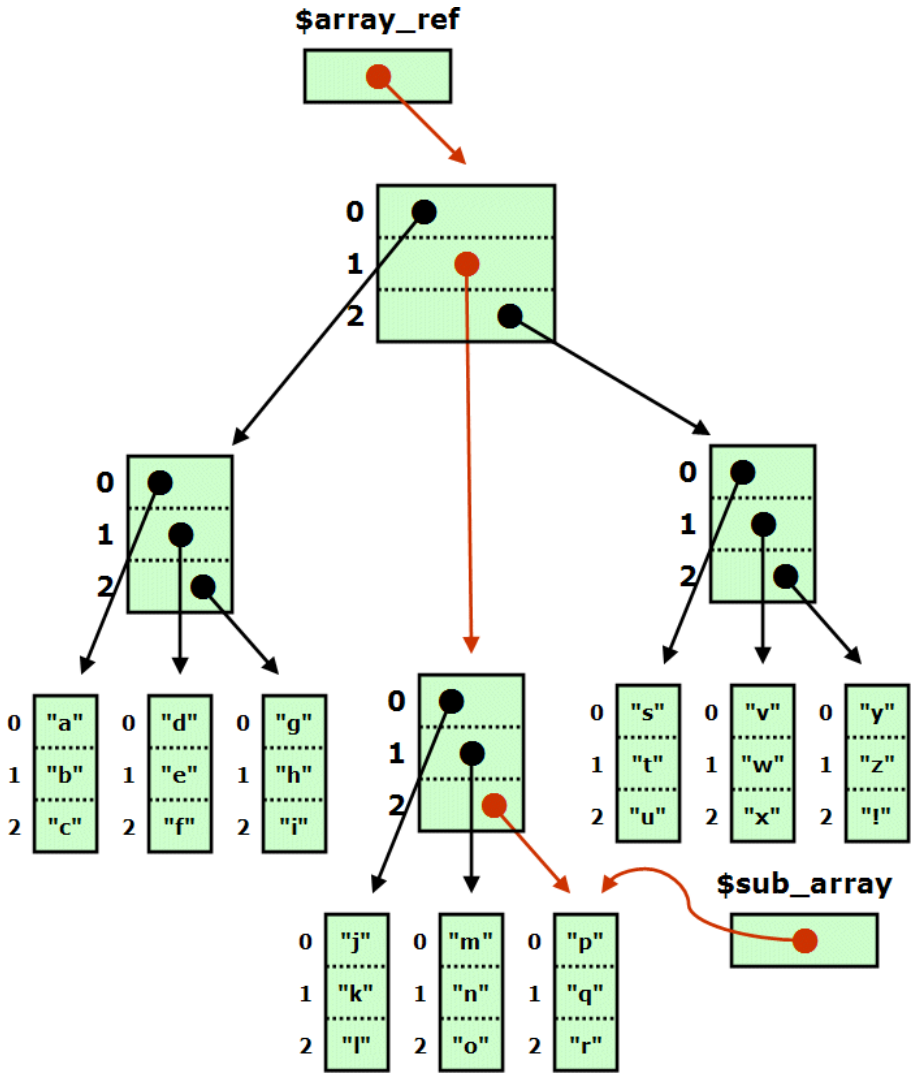
> ***Values that are equivalent should be able to be substituted for one another without altering the value of an expression.***

- Suppose we have the arrayref:

```
$array_ref = [
               [
                 ["a", "b", "c"],
                 ["d", "e", "f"],
                 ["g", "h", "i"]
               ],
               [
                 ["j", "k", "l"],
                 ["m", "n", "o"],
                 ["p", "q", "r"]
               ],
               [
                 ["s", "t", "u"],
                 ["v", "w", "x"],
                 ["y", "z", "!"]
               ]
             ];
```

- Note that `$array_ref->[1]->[2]` is the arrayref `["p", "q", "r"]`.
- Now suppose you set `$sub_array = $array_ref->[1]->[2]`. Because `$array_ref->[1]->[2]` is a reference, when you set `$sub_array` equal to (a copy of) it, `$sub_array` will have the same referent.

Thus, the result in memory is this:



- Now `$sub_array` can be substituted for `$array_ref->[1]->[2]`, because the two expressions are equivalent.
  - Therefore: because
    `$array_ref->[1]->[2]->[0]`
    is `"p"`,
    `$sub_array->[0]`
    is also `"p"`.
- This very useful for simplifying operations, particularly when you are using more complex data structures.
  - Suppose you want to sum the members of a 2-D array that is stored 3 levels down in a 5-level structure:
  - The code would look like this:

```
for my $r (0 .. $#{$data->[$level1]->[$level2]->[$level3]}) {
  for my $c (0 .. $#{$data->[$level1]->[$level2]->[$level3]->[$r]}) {
    $sum += $data->[$level1]->[$level2]->[$level3]->[$r]->[$c];
  }
}
```

  - Now let's take advantage of the fact that `$data->[$level1]->[$level2]->[$level3]` is a "common ancestor" of all of our operations in this code and set `$array` equal to it before we begin the loop.

- Now the code is:

```
my $array = $data->[$level1]->[$level2]->[$level3];
for my $r (0 .. $#{$array}) {
  for my $c (0 .. $#{$array->[$r]}) {
    $sum += $array->[$r]->[$c];
  }
}
```

- And by then substituting `$row` for `$array->[$r]` (which, recall, is actually `$data->[$level1]->[$level2]->[$level3]->[$r]`), you can simplify even further:

```
my $array = $data->[$level1]->[$level2]->[$level3];
for my $r (0 .. $#{$array}) {
  my $row = $array->[$r];
  for my $c (0 .. $#{$row}) {
    $sum += $row->[$c];
  }
}
```

- The original monstrosity has become downright manageable.
- Another advantage is that if you modify the data structure, for instance by adding another level, you only need to modify the line of code that sets up `$array`, instead of modifying every appearance of `$data`.
- If you consider it, you will notice that the "syntactic sugar" of eliding the intermediate arrows (i.e. using `$array->[0][0][0]` instead of `$array->[0]->[0]->[0]`) violates the substitution principle.
  - This is a trade-off that may or may not be worth it.
  - I use the short form because I find it clearer; if you choose to do so you can still use a slightly modified form of the substitution principle, you just need to think about it a little harder when you do.

Complex Data Structures

- As you can nest arrayrefs in arrayrefs to create multidimensional arrays, you can next hashrefs within hashrefs or hashrefs and arrayrefs within each other to create other types of data structures:

level 1	level 2	possible usage	example
arrayref	arrayref	Array of data indexed by x, y coordinates	topographic data
arrayref	hashref	Array of records	student records
hashref	arrayref	Data arrays indexed by name	sensor readings by location
hashref	hashref	Records indexed by name	class schedule

- Of course you can nest as deeply as you want.
- For example, the following is an arrayref of hashrefs, being used as an array of records:

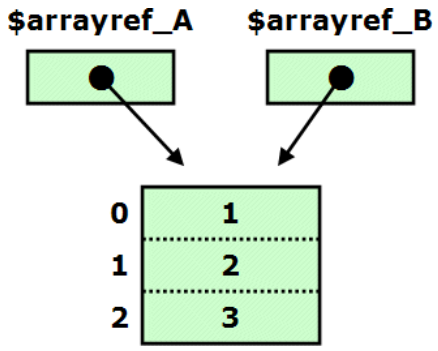
```
$people = [
  { name=>'John Smith', id=>'J37482-SH4', eyes=>'blue',  hair=>'brown' },
  { name=>'Mary Jones', id=>'M67134-JS5', eyes=>'blue',  hair=>'blond' },
  { name=>'Chris Lee',  id=>'C47923-LE1', eyes=>'brown', hair=>'black' },
];

print "The first person's name is $people->[0]{name}\n";

# add a person
push @{$people}, { name=>"Will Johnson", id=>"X93888-QA2', eyes=>'hazel', hair=>'blond' };
```

- As multidimensional arrays do not guarantee orthogonality, the hashrefs within an "arrays of records" are not constrained in any way. All of the following operations, illegal in a C array of `struct`s, are allowed in Perl:
  - `delete $people->[2]{id}`;
  - `$people->[1]{eyes} = [1, 2, 3]`;
  - `unshift @{$people}, "RUTABAGA"`;
- The programmer is responsible for using his or her own data structures in the manner intended.
- Normally this is not a problem, but it is tricky to prevent abuse of your data structures when writing libraries that might be used by others.
  - The short answer is that when writing a module, direct access to data structures should be denied, and functions to access and change the data should be provided.

    - Some approaches will be covered in the section on modules.
- As with 2-D, 3-D, and higher order of multidimensional arrays, to dereference all the way to the scalar elements, you simply follow the data structure with a thin arrow and then a key or index for each level of the data structure, using keys enclosed in braces to go through a hashref and indices wrapped in brackets when the current level is a hashref.
- You may stop before you get to the end, in which case you are referring to an arrayref or a hashref that represents a sub-part of the structure. You can dereference this with the `@{}` or `%{}` constructs to treat it as a plain hash or array, to count elements, get a list of keys for looping, or add/remove values.
    - Any operation applicable to a hash is applicable to a dereferenced hashref, and the same applies to arrayrefs.
    - You may also stop early and assign the result to a plain array or hash to make dealing with it easier.
        - In the above example, you might consider something like `%this_person = %{$people->[0]}`
        - This is not necessarily an application of the substitution principle, because a dereferences hash and a hashref are not the same thing. Therefore you must use some caution:
            - A couple of caveats: first, if you stop dereferencing at the next-to-last level, your result will be a plain array or hash containing scalar elements. If you stop before that, your plain array or hash will contain references for (at least some) elements. This is permissible, especially if you are just looping over the keys or elements, but be sure you know what's contained in a scalar variable that you plucked from deep within a multi-level data structure.
            - Assigning to a temporary array or hash variable as in the `%this_person` example is a copy operation. Therefore, modifications to its elements will not effect the original data structure.
            - However, if a temporary array or hash contains a larger chunk of a data structure than just a next-to-last-level hash or array, its elements may be references. These references will be copies of the references in the original data structure, but a copy of a reference refers to the same thing as the original reference. Therefore, modifying the referent of these elements **will** affect the original data structure.
    - Heavy use of `Data::Dumper` is recommended when developing data structures, especially complex ones. `Dumper` does an excellent job of displaying multi-level structures.

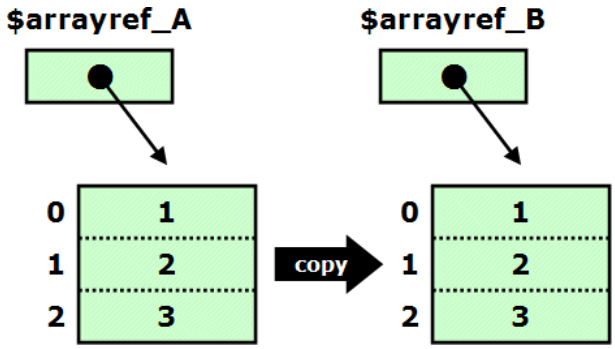
Shallow Copying vs. Deep Copying

- As stated, if you copy a reference:
```
$arrayref_A = [1, 2, 3];
$arrayref_B = $arrayref_A;
```
Your references point to the same places and changes to one affect the other
    - The result in memory is this:



    - If you say
```
$arrayref_A->[0] = 12;
```
      the value of `$arrayref_B->[0]` will now be `12`.
- Also as stated, if you create an anonymous arrayref with a copy of another arrayref in it, the elements in the second arrayref are copies of the elements of the first arrayref:
```
$arrayref_A = [1, 2, 3]
$arrayref_B = [ @{$arrayref_A} ];
```

The result in memory is this:



Therefore, changes the elements of **`$arrayref_A`** have no effect on the elements of **`$arrayref_B`**, and vice-versa.

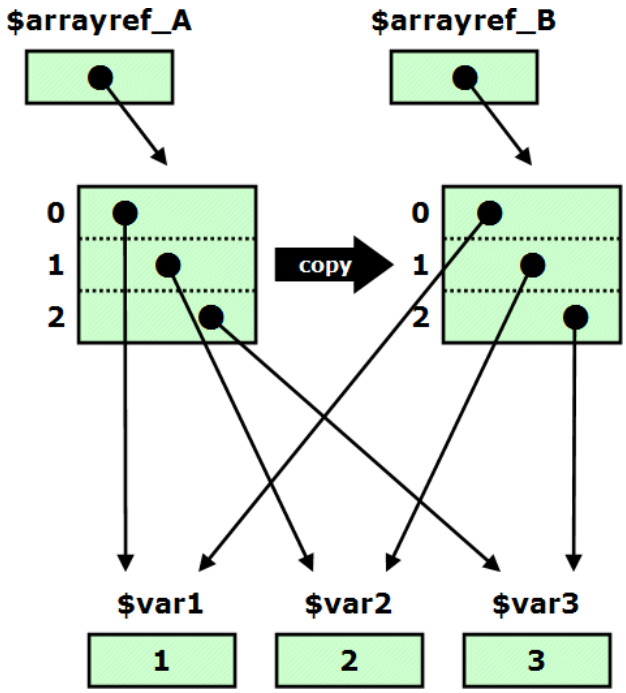
- If you say
  ```
  $arrayref_A->[0] = 12;
  ```
  the value of **`$arrayref_B->[0]`** will still be **`1`**.

- But what if the thing you copy **contains references?**

Suppose you set up the following memory structures:

```
($var1, $var2, $var3) = (1, 2, 3);
$arrayref_A = [ \$var1, \$var2, \$var3 ];
$arrayref_B = [ @{$arrayref_A} ];
```

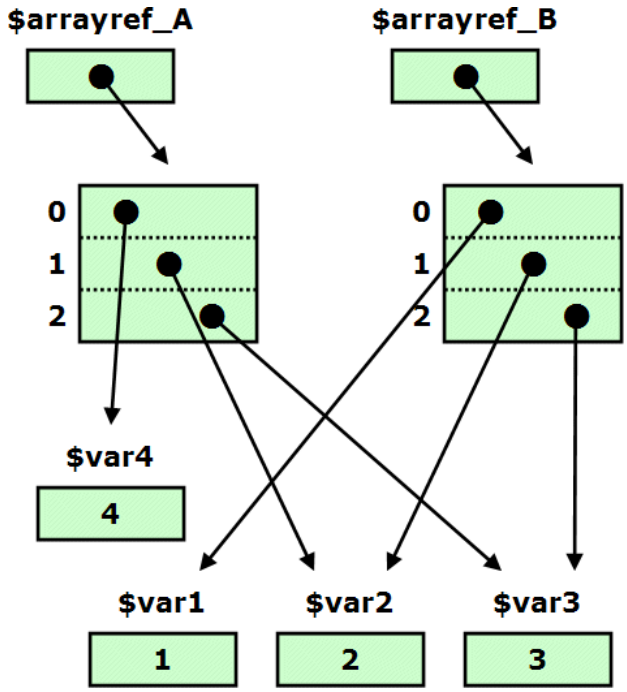
The result in memory is this:



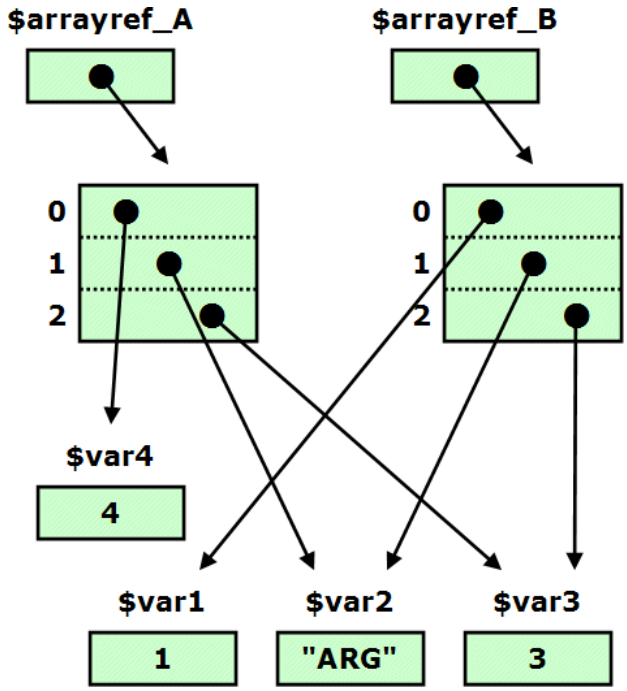
Now what happens if you modify the elements of **`$arrayref_A`**?

- The elements of **`$arrayref_B`** are copies of the elements in **`$arrayref_A`**. Therefore, changes to the elements of **`$arrayref_A`** have no affect on the elements of **`$arrayref_B`**.
- So suppose you say:
  ```
  $arrayref_A->[0] = \$var4;
  ```
  You are changing the value of the first element of **`$arrayref_A`** to point someplace new; the first element of **`$arrayref_B`** will still point at **`$var1`**.

The result in memory is this:



- But remember that a copy of a reference points to the same place as the original reference. So even though `$arrayref_A->[1]` is a **copy** of `$arrayref_B->[1]`, and direct changes to `$arrayref_A->[1]` don't affect `$arrayref_B->[1]`, **if you modify the referent of** `$arrayref_A->[1]` with code like:
  ```
  ${$arrayref_A->[1]} = "ARG"
  ```
  you affect the referent of `$arrayref_A->[1]` (in other words, `$var2`), not `$arrayref_A->[1]` itself, and as a side-effect, **you will also affect the referent of** `$arrayref_B->[1]` (also `$var2`). The result is this:



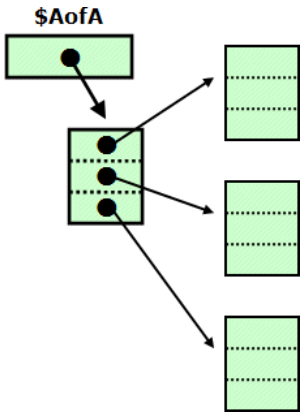
- This "first-level-only" method of copying data structures that Perl employs is called *shallow copying*.
  - When you say
    `$copy_of_complex_data_structure = [ @{$original_complex_data_structure} ];`
    you only copy the elements in the first level—i.e. the direct elements of **`$original_complex_data_structure`**. Any data in the second and subsequent levels pointed to by references in the first level is NOT copied.
    - Therefore changes to the original data structure deeper than the first level will still affect the copy (and vice-versa).
  - Making a complete copy of an entire data structure so that the copy has an identical structure and the same leaf-node data, but no shared references, is called a *deep copy*, and Perl does not support deep copying natively.
    - It becomes hard to define what "deep copy" even means under several circumstances.
      - For example, what if a data structure contains a reference to a system variable, like **`@ARGV`** or **`%ENV`**. A deep-copied data structure should probably preserve that reference as a reference, not make a copy of **`@ARGV`** and have the copied data structure reference that.
        - For that matter, what if the data structure simply includes a reference to an unrelated variable? Should a deep copy make a copy of the unrelated variable or simply copy the reference?
      - A data structure that contains a reference to a subroutine (discussed below) is also hard to deep-copy.
      - A data structure with a non-linear structure, such as one that includes *circular references*, is very difficult to deep-copy.
    - If you assume a data structure that consists only of nested hashrefs and arrayrefs, it is still a non-trivial matter to implement an arbitrary deep copy function, but think about how it might be accomplished.
      - Hint: it requires a recursive function and heavy use of the **`ref`** function.

Complex Data Structures: Reference

NOTES:

- For clarity, it is assumed that all variables shown have been pre-declared.
- Remember that some actions (**`$arrayref2 = [@{$arrayref1}]`**) copy elements and other actions (**`$arrayref2 = $arrayref1`**) only copy the refence.
- Recall that intermediate thin arrows in a dereference like **`$data->{$id}->{SCHEDULE}->[0]`** can be elided, and the previous written as **`$data->{$id}{SCHEDULE}[0]`**. Both forms are shown.
- For this reference, the term "structure" refers to the elements of the base level hash or array (these elements themselves being arrays or hashes); the term "element" refers to an element in the second level (these elements being plain scalars in these examples)

Arrayref of Arrayrefs (2-D Array)

Structure	

Declaration/Initialization	`$AofA = [`
	`  [ 1, 2, 3 ],`
	`  [ 4, 5, 6 ],`
	`  [ 7, 8, 9 ],`
	`];`

Access a Structure	`$sub_arrayref = $AofA->[0];` `@sub_array = @{$AofA->[0]};`

Set/Change a Structure	`$AofA->[0] = [ 101, 102, 103 ];`
	`$AofA->[0] = \@new_array;`
	`$AofA->[0] = [ @new_array ];`

Add a Structure	`push @{$AofA}, [ 101, 102, 103 ];`
	`push @{$AofA}, \@new_array;`
	`push @{$AofA}, [ @new_array ];`

Delete a Structure	`pop @{$AofA};`
	`shift @{$AofA};`
	`splice @{$AofA}, $position, 1;`

Count Structures	`$count = @{$AofA};`
Sort Structures	`$AofA = [ sort @{$AofA} ];`
List Structures' Keys	N/A

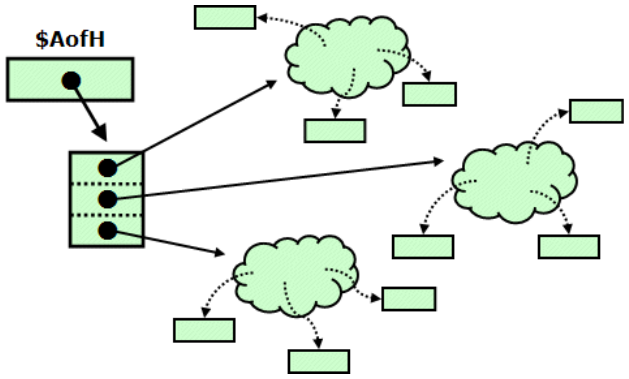
Access an Element	`print $AofA->[0]->[0];`
	`print $AofA->[0][0];`
	—
	`$var = $AofA->[0]->[0];`
	`$var = $AofA->[0][0];`

Set/Change an Element	`$AofA->[0]->[0] = 99;` `$AofA->[0][0] = 99;`
Add an Element to a Structure	`push @{$AofA->[0]}, 100;`

Delete an Element from a Structure	`pop @{$AofA->[0]};`
	`shift @{$AofA->[0]};`
	`splice @{$AofA->[0]}, $position, 1;`

Count Elements in a Structure	`$count = @{$AofA->[0]};`
Sort Elements in a Structure	`$AofA->[0] = [ sort @{$AofA->[0]} ];`
List Elements' Keys in a Structure	N/A

Arrayref of Hashrefs (Array of Records)

Structure	

Declaration/Initialization	`$AofH = [`
	`  { K1 => "A", K2 => "B", K3 => "C" },`
	`  { K4 => "D", K5 => "E", K6 => "F" },`
	`  { K7 => "G", K8 => "H", K9 => "I" },`
	`];`

Access a Structure	`$sub_hashref = $AofH->[0];` `%sub_hash = %{$AofH->[0]};`

Set/Change a Structure	`$AofH->[0] = { K101 => "X", K102 => "Y" };`
	`$AofH->[0] = \%new_hash;`
	`$AofH->[0] = { %new_hash };`

Add a Structure	`push @{$AofH}, { K101 => "X", K102 => "Y" };`
	`push @{$AofH}, \%new_hash;`
	`push @{$AofH}, { %new_hash };`

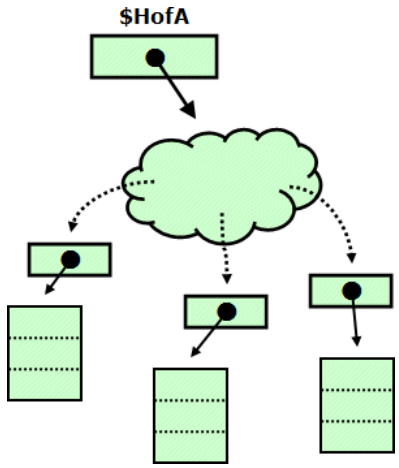
Delete a Structure	`pop @{$AofH};`
	`shift @{$AofH};`
	`splice @{$AofH}, $position, 1;`

Count Structures	`$count = @{$AofH};`
Sort Structures	`$AofH = [ sort @{$AofH} ];`
List Structures' Keys	N/A

Access an Element	`print $AofH->[0]->{K1};`
	`print $AofH->[0]{K1};`
	—
	`$var = $AofH->[0]->{K1};`
	`$var = $AofH->[0]{K1};`

Set/Change an Element	`$AofH->[0]->{K1} = "X";` `$AofH->[0]{K1} = "X";`
Add an Element to a Structure	`$AofH->[0]{K10} = "X";`
Delete an Element from a Structure	`delete $AofH->[0]{K1};`
Count Elements in a Structure	`$count = keys %{$AofH->[0]};`
Sort Elements in a Structure	N/A
List Elements' Keys in a Structure	`@keys = keys %{$AofH->[0]};`

Hashref of Arrayrefs (String-Indexed Arrays)

Structure	

Declaration/Initialization	`$HofA = {`
	`  KeyA => [ 1, 2, 3 ],`
	`  KeyB => [ 4, 5, 6 ],`
	`  KeyC => [ 7, 8, 9 ],`
	`};`

Access a Structure	`$sub_arrayref = $HofA->{KeyA};` `@sub_array = @{$HofA->{KeyA}};`

Set/Change a Structure	`$HofA->{KeyA} = [ 101, 102, 103 ];`
	`$HofA->{KeyA} = \@new_array;`
	`$HofA->{KeyA} = [ @new_array ];`

Add a Structure	`$HofA->{KeyX} = [ 101, 102, 103 ];`
	`$HofA->{KeyX} = \@new_array;`
	`$HofA->{KeyX} = [ @new_array ];`

Delete a Structure	`delete $HofA->{KeyA};`
Count Structures	`$count = keys %{$HofA};`
Sort Structures	N/A
List Structures' Keys	`@keys = keys %{$HofA};`

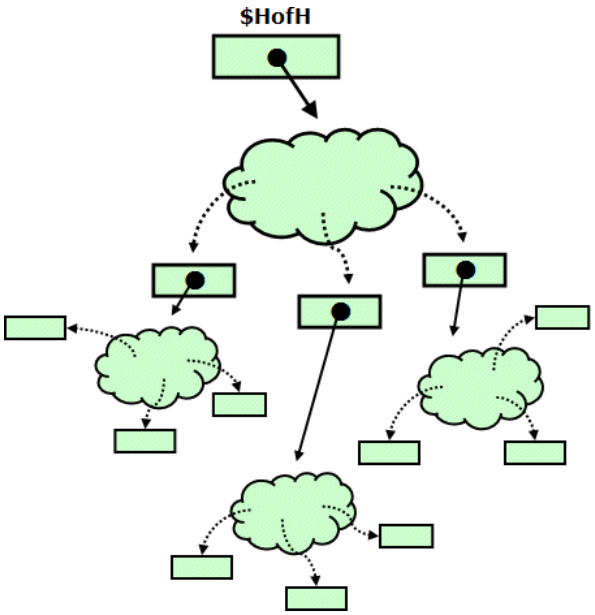
Access an Element	`print $HofA->{KeyA}->[0];`
	`print $HofA->{KeyA}[0];`
	—
	`$var = $HofA->{KeyA}->[0];`
	`$var = $HofA->{KeyA}[0];`

Set/Change an Element	`$HofA->{KeyA}->[0] = 99;` `$HofA->{KeyA}[0] = 99;`
Add an Element to a Structure	`push @{$HofA->{KeyA}}, 100;`

Delete an Element from a Structure	`pop @{$HofA->{KeyA}};`
	`shift @{$HofA->{KeyA}};`
	`splice @{$HofA->{KeyA}}, $position, 1;`

Count Elements in a Structure	`$count = @{$HofA->{KeyA}};`
Sort Elements in a Structure	`$HofA->{KeyA} = [ sort @{$HofA->{KeyA}} ];`
List Elements' Keys in a Structure	N/A

Hashref of Hashrefs (String-Indexed Records)

Structure	

Declaration/Initialization	`$HofH = {`
	`  KeyA => { K1 => "A", K2 => "B", K3 => "C" },`
	`  KeyB => { K4 => "D", K5 => "E", K6 => "F" },`
	`  KeyC => { K7 => "G", K8 => "H", K9 => "H" },`
	`};`

Access a Structure	`$sub_hashref = $HofH->{KeyA};` `%sub_hash = %{$HofH->{KeyA}};`

Set/Change a Structure	`$HofH->{KeyA} = { K101 => "X", K102 => "Y" };`
	`$HofH->{KeyA} = \%new_hash;`
	`$HofH->{KeyA} = { %new_hash };`

Add a Structure	`$HofH->{KeyX} = { K101 => "X", K102 => "Y" };`
	`$HofH->{KeyX} = \%new_hash;`
	`$HofH->{KeyX} = { %new_hash };`

Delete a Structure	`delete $HofH->{KeyA};`
Count Structures	`$count = keys %{$HofH};`
Sort Structures	N/A
List Structures' Keys	`@keys = keys %{$HofH};`

Access an Element	`print $HofH->{KeyA}->{K1};`
	`print $HofH->{KeyA}{K1};`
	—
	`$var = $HofH->{KeyA}->{K1};`
	`$var = $HofH->{KeyA}{K1};`

Set/Change an Element	`$HofH->{KeyA}->{K1} = "X";` `$HofH->{KeyA}{K1} = "X";`
Add an Element to a Structure	`$HofH->{KeyA}{K10} = "X";`
Delete an Element from a Structure	`delete $HofH->{KeyA}{K1};`
Count Elements in a Structure	`$count = keys %{$HofH->{KeyA}};`
Sort Elements in a Structure	N/A
List Elements' Keys in a Structure	`@keys = keys %{$HofH->{KeyA}};`

References and Subroutines

Passing References to Subroutines

- We have already seen that a subroutine's argument list is evaluated in array context, and its elements are therefore flattened, making it impossible to pass two arrays to a subroutine simply by saying
  `compare_arrays(@array1, @array2, @array3)`
  In this case, the `@_` array inside `compare_arrays` will just contain all the elements of `@array1` followed by all the elements of `@array2` and `@array3`. You can get around this by passing some torturous set of arguments like
  `compare_arrays($#array1, @array1, $#array2, @array2, $#array3, @array3)`
  but this is really confusing.
- A much better solution is to use arrayrefs. There are three slightly different ways of accomplishing this.
  - This first is to use arrayrefs as the primary variable and pass them directly:

```
1 $arrayref1 = [1, 2, 3];
2 $arrayref2 = [2, 3, 4];
3 $arrayref3 = [3, 4, 5];
4
5 compare_arrays($arrayref1, $arrayref2, $arrayref3);
```

  - If your primary variables are plain arrays, you can use the reference operator to pass references to these arrays:

```
1 @array1 = (1, 2, 3);
2 @array2 = (2, 3, 4);
3 @array3 = (3, 4, 5);
4
5 compare_arrays(\@array1, \@array2, \@array3);
```

  - You can also pass anonymous arrayrefs derived from your arrays:

```
1 @array1 = (1, 2, 3);
2 @array2 = (2, 3, 4);
3 @array3 = (3, 4, 5);
4
5 compare_arrays([@array1], [@array2], [@array3]);
```

  - It is possible to mix all of these with no problems:

```
compare_arrays(\@array1, [@array2], $arrayref3);
```

    but as always, consistency is key. While this works, it is to be avoided.
    - Unless your variables already exists as a mix of arrays and arrayrefs for some reason.
- Accessing the data inside the subroutine is no different than accessing any other scalar data passed to a subroutine. You just need to know that the variable is a reference:

```
1 sub compare_arrays {
2   my ($arrayref1, $arrayref2, $arrayref3) = @_;
3   print "There are " . @{$arrayref1} . " elements in array #1\n":
4   $arrayref2->[0] = "hello";
5 }
```

- Another reason to write a subroutine that accepts references is because the subroutine wants to modify its elements. Writing the subroutine to accept references is a red flag to the user of the subroutine that the arguments may be modified.
  - This is a supplement, not a substitute, for naming the subroutine in a way that clearly implies it modifies its arguments.

- Here is an example of a subroutine that uses a passed reference to modify its argument:

```
sub set_keys_to_uc {
  my $hashref = $_[0];
  die "First arg to set_keys_to_uc() must be hashref!\n"
    unless "HASH" eq ref $hashref;
  foreach my $key (keys %{$hashref}) {
    if ($key ne uc $key) {
      $hashref->{uc $key} = $hashref->{$key};
      delete $hashref->{$key};
    }
  }
  print "Modified hash: " . (Dumper $hashref);
}
```

This subroutine can be called in three ways:

- Using a reference to an existing hash variable:

```
my %hash = (
  tomato => "fruit",
  Rhubarb => "Vegetable",
  BEEF => "MEAT",
  whopper => "other",
);

print "Hash before: " . (Dumper \%hash);
set_keys_to_uc(\%hash);
print "Hash after: " . (Dumper \%hash);
```

  which outputs

```
Hash before: $VAR1 = {
          'Rhubarb' => 'Vegetable',
          'BEEF' => 'MEAT',
          'tomato' => 'fruit',
          'whopper' => 'other'
        };
Modified hash: $VAR1 = {
          'TOMATO' => 'fruit',
          'BEEF' => 'MEAT',
          'RHUBARB' => 'Vegetable',
          'WHOPPER' => 'other'
        };
Hash after: $VAR1 = {
          'TOMATO' => 'fruit',
          'BEEF' => 'MEAT',
          'RHUBARB' => 'Vegetable',
          'WHOPPER' => 'other'
        };
```

- Using hashref variable:

```
my $hash = {
  tomato => "fruit",
  Rhubarb => "Vegetable",
  BEEF => "MEAT",
  whopper => "other",
};

print "Hash before: " . (Dumper $hash);
set_keys_to_uc($hash);
print "Hash after: " . (Dumper $hash);
```

which outputs

```
Hash before: $VAR1 = {
          'Rhubarb' => 'Vegetable',
          'BEEF' => 'MEAT',
          'tomato' => 'fruit',
          'whopper' => 'other'
        };
Modified hash: $VAR1 = {
          'TOMATO' => 'fruit',
          'BEEF' => 'MEAT',
          'RHUBARB' => 'Vegetable',
          'WHOPPER' => 'other'
        };
Hash after: $VAR1 = {
          'TOMATO' => 'fruit',
          'BEEF' => 'MEAT',
          'RHUBARB' => 'Vegetable',
          'WHOPPER' => 'other'
        };
```

- Using an anonymous hashref with a copy of a hash variable:

```
1 my %hash = (
2   tomato => "fruit",
3   Rhubarb => "Vegetable",
4   BEEF => "MEAT",
5   whopper => "other",
6 );
7
8 print "  Hash before: " . (Dumper \%hash);
9 set_keys_to_uc( {%hash} );
10 print "  Hash after: " . (Dumper \%hash);
```

which outputs

```
  Hash before: $VAR1 = {
          'Rhubarb' => 'Vegetable',
          'BEEF' => 'MEAT',
          'tomato' => 'fruit',
          'whopper' => 'other'
        };
Modified hash: $VAR1 = {
          'TOMATO' => 'fruit',
          'BEEF' => 'MEAT',
          'RHUBARB' => 'Vegetable',
          'WHOPPER' => 'other'
        };
  Hash after: $VAR1 = {
          'Rhubarb' => 'Vegetable',
          'BEEF' => 'MEAT',
          'tomato' => 'fruit',
          'whopper' => 'other'
        };
```

  - In this final example, notice the changes **did not affect** the original hash variable. This is because a copy of all keys/elements was made and passed. This means that modifying `$hashref` inside the `set_keys_to_uc()` subroutine will not affect any data outside the subroutine.

- The above subroutine will not work if it is passed anything other than a hashref as its first argument. Notice lines 3 and 4, which us the `ref` function to confirm the status of `$_[0]` before it is used. If accidentally passed a plain hash:

```
1 my %hash = (
2   tomato => "fruit",
3   Rhubarb => "Vegetable",
4   BEEF => "MEAT",
5   whopper => "other",
6 );
7
8 print "Hash before: " . (Dumper \%hash);
9 set_keys_to_uc(%hash);
10 print "Hash after: " . (Dumper \%hash);
```

the subroutine will `die` with a helpful error message:

```
Hash before: $VAR1 = {
          'Rhubarb' => 'Vegetable',
          'BEEF' => 'MEAT',
          'tomato' => 'fruit',
          'whopper' => 'other'
        };
First arg to set_keys_to_uc() must be hashref!
```

- Also, if a subroutine modifies **the reference itself** (as opposed to dereferencing it and modifying the result), the original hash
  - will not be affected if it is passed as a ref to a hash variable
  - will be modified if it is a hashref scalar in the calling code.

So the code:

```
1  sub mod_hashref {
2    $_[0] = { new => 'hashref' };
3  }
4
5  my %hash = (
6    tomato => "fruit",
7    Rhubarb => "Vegetable",
8    BEEF => "MEAT",
9    whopper => "other",
10 );
11
12 print "Hash before: " . (Dumper \%hash);
13 mod_hashref(\%hash);
14 print "Hash after: " . (Dumper \%hash);
```

outputs

```
Hash before: $VAR1 = {
          'Rhubarb' => 'Vegetable',
          'BEEF' => 'MEAT',
          'tomato' => 'fruit',
          'whopper' => 'other'
        };
Hash after: $VAR1 = {
          'Rhubarb' => 'Vegetable',
          'BEEF' => 'MEAT',
          'tomato' => 'fruit',
          'whopper' => 'other'
        };
```

but the code

```
sub mod_hashref {
  $_[0] = { new => 'hashref' };
}

my $hashref = {
  tomato => "fruit",
  Rhubarb => "Vegetable",
  BEEF => "MEAT",
  whopper => "other",
};

print "Hash before: " . (Dumper $hashref);
mod_hashref($hashref);
print "Hash after: " . (Dumper $hashref);
```

outputs

```
Hash before: $VAR1 = {
          'Rhubarb' => 'Vegetable',
          'BEEF' => 'MEAT',
          'tomato' => 'fruit',
          'whopper' => 'other'
        };
Hash after: $VAR1 = {
          'new' => 'hashref'
        };
```

- Remember that all points made here about hashes and hashrefs apply equally to arrays and arrayrefs.
- As you have surmised, code like
  `$arrayref = [ $arrayref1, $arrayref2 ]`
  or
  `@array = ( $arrayref1, $arrayref2 )`
  does not flatten anything because arrayrefs are scalar and thus not subject to flattening. If you want to flatten arrayrefs, you must dereference them:
  `$arrayref = [ @{$arrayref1}, @{$arrayref2} ]`
  or
  `@array = ( @{$arrayref1}, @{$arrayref2} )`
  - Similar comments apply to overlaying hashes:
    `%hash = ( $hashref1, $hashref2 )`
    will not work, but
    `%hash = ( %{$hashref1}, %{$hashref2} )`
    will.
  - Discussion question: what will the first (incorrect) line actually produce?

Returning References from Subroutines

- References are a perfectly valid return value from subroutines.
- As subroutines have always been able to return nothing, scalars, arrays, and hash-like arrays, this is not as much of a game-changer as passing references to a subroutine is, but be aware that a subroutine can return any type of reference, including a complex data structure.

Coderefs and Anonymous Subroutines

- It is possible to create a reference to a block of code (called a *coderef*) by using the reference operator and the ampersand ("&") sigil that means "subroutine". To make a reference to an existing subroutine, you need only do this:

```
sub sum {
  return eval join "+", @_;
}

my $subref = \&sum;
```

To dereference and call the subroutine, use the arrow dereference operator with parentheses to indicate a subroutine call:

```
my @data = (2, 4, 6, 8, 10);
print $subref->(@data);
```

- The parentheses are part of the dereference-and-call syntax and are therefore always required, even if the subroutine does not take any arguments.

- You can also create a reference to an anonymous subroutine using the **sub** keyword:

```
1 my $sub_sum = sub {
2   return eval join "+", @_;
3 };
4
5 print $sub_sum->(1, 2, 3);
```

  - Note the semicolon at the end of the code block.
  - The calling syntax is the same.
  - This feature has implications for creating private functions in libraries. This will be explored further in the unit on modules.

- Coderefs are scalar values like all other references and can be stored in a scalar variable, as an element of an array/arrayref, or as a value in a hash/hashref. This allows you to do "function lookups". This type of code might look like:

```
 1 my $operators = {
 2   '+' => sub { $_[0] + $_[1] },
 3   '-' => sub { $_[0] - $_[1] },
 4   '*' => sub { $_[0] * $_[1] },
 5   '/' => sub { $_[0] / $_[1] },
 6 };
 7
 8 my @expressions = (
 9   "3 + 27",
10   "100 / 5",
11   "6 * 12"
12 );
13
14 foreach my $expression (@expressions) {
15   my ($operand1, $operator, $operand2) = split " ", $expression;
16   my $result = $operators->{$operator}->($operand1, $operand2);
17   print "$expression = $result\n";
18 }
```

  This code would output:

```
3 + 27 = 30
100 / 5 = 20
6 * 12 = 72
```

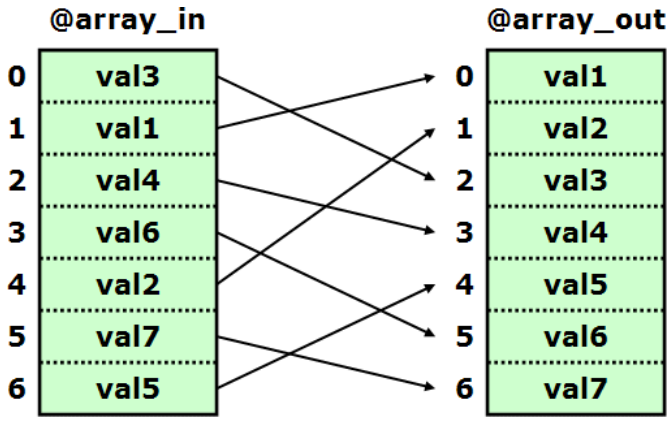
Mind-Bending Array Functions

- Perl includes several functions that accept an array and return a modified array (this operation is called an *array transform*). So far we have seen two of these functions: **sort** (in an abbreviated form) and **reverse**. Here are the rest of them.

Sorting an Array

Will the real `sort` function please step forward?

- The simple version of **sort** merely sorts an array in ASCIIbetical order. This is useful for sorting plain text, but is much less useful for numeric data (and is downright useless for non-scalar data).



- Fortunately, the **sort** function, in its complete form, takes a *comparator* function. This can be
  - the name of a subroutine
  - a literal block of code
  - a plain scalar variable (not a hash or array element) containing a coderef
- The comparator function must be designed to examine the special global variables **$a** and **$b** (not its arguments) and return 1 if **$a** is "greater", -1 if **$b** is "greater", and 0 if they are the same.
  - Recall the handy **<=>** and **cmp** functions.
- Examples:
  - To sort an array of numbers numerically:

```
sub by_num {
  $a <=> $b
}

my @data = (4, 67, 1, 45, 1235, -1234);

@data = sort by_num @data;
```

    - Comparator functions are often named "by_(method)", so the call to **sort** reads more intuitively.
    - Note the lack of comma between the comparator and the data to sort.
  - To sort an array of strings by length:

```
my @data = ("medium", "very long", "short", "tiny");

@data = sort { (length $a) <=> (length $b) } @data;
```

  - To sort an array of strings case insensitively:

```
my $sort_modes = {
  normal   => sub { $a cmp $b },
  num      => sub { $a <=> $b },
  case_ins => sub { (lc $a) cmp (lc $b) },
};

my @data = ("Banana", "PEAR", "apple");

my $sort_mode = $sort_modes->{case_ins};
@data = sort $sort_mode @data;
```

    - Note you have to store the coderef to the comparator in a plain scalar variable.

- To sort an array of data structures based on a value deep within them:

```
$data = [
          sort {
                 $a->{SCHEDULE}{Tuesday}[0]{start_time}
                   <=>
                 $b->{SCHEDULE}{Tuesday}[0]{start_time}
               } @{$data}
        ];
```

- The following is a *cascading* sort. It uses short-circuit evaluation to sort first by one parameter and if they are the same, by a second parameter (and so on).

```
my @data = qw(happy Happy Zanzibar Goodbye Happier apple hoppy HOPPY Hoppy harpy);

@data = sort {
                (length $a) <=> (length $b)
                  or
                (uc $a) cmp (uc $b)
                  or
                $a cmp $b
              } @data;

print "(@data)\n";
```

  This code outputs:

```
(apple Happy happy harpy HOPPY Hoppy hoppy Goodbye Happier Zanzibar)
```

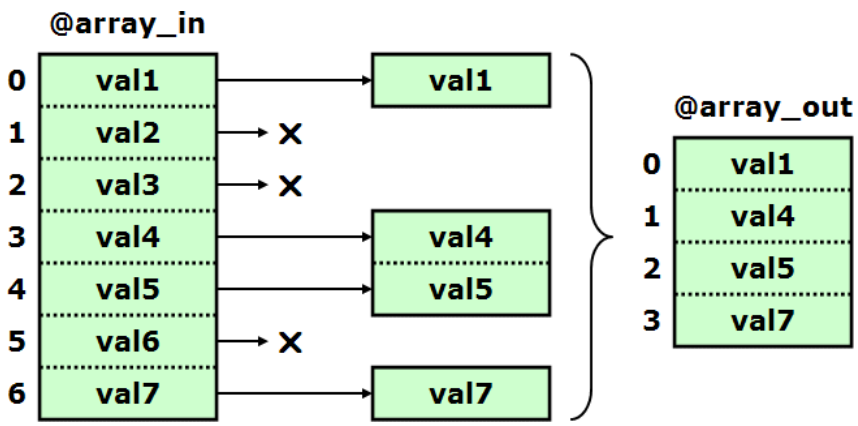
  - Compare to the results of a standard ASCIIbetical sort:

```
(Goodbye HOPPY Happier Happy Hoppy Zanzibar apple happy harpy hoppy)
```

- Remember you can pass `sort` (as well as `grep` and `map`) an array variable, a dereferenced arrayref, or a literal list.
  - You **cannot** pass an arrayref (well, you CAN, but `sort` won't sort the contents of the arrayref).
  - Remember also that `sort` returns a plain array, not an arrayref.
  - In the next-to-last example above, notice that the data structure itself (`$data`) is an arrayref, not an array, so it must be dereferenced when passed to `sort` and the return value of `sort` must be converted back into a reference.
- To reverse any sort, call `reverse` on the result or switch `$a` and `$b` in the comparator.
- `sort` is not destructive; it does not modify the original array.

Filtering an Array

- The `grep` function takes a code block or function like `sort`, called the *filter* function. The filter function uses `$_` to derive a true or false value.
- The result of a call to `grep` is an array consisting of all the elements of the passed array for which the filter, called with `$_` set to the array element in question, returned true.



  - In scalar context, it just returns a count of how many elements "matched".
  - In Boolean context, this is helpfully true if any elements matched and false if none did.
- Like `sort`, you can pass the name of a subroutine, a coderef, or a literal block of code.

- Examples:
  - To determine if a list of people contains a certain person:

```
my @names = qw(Bill Mary John Peter);

foreach my $testname ('John', 'Frankenstein') {
  if (grep { (uc $_) eq (uc $testname) } @names) {
    print "found \"$testname\"\n";
  }
  else {
    print "didn't find \"$testname\"\n";
  }
}
```

    This code outputs

```
found "John"
didn't find "Frankenstein"
```

  - To count the non-negative values in a list of data:

```
sub for_non_neg {
  return $_ >= 0;
}

my @data = (-2, 5, 0, -1.5, 3.5, -6, 0.5, 0.75);
my $count = grep for_non_neg, @data;
print "Found $count non-negative numbers\n";
```

    This code outputs

```
Found 5 non-negative numbers
```

    - Note that the filter function is often named `for_(property)` so that the call to `grep` reads naturally.
  - To strip undefined values from a list:

```
my @data = (1, undef, 3, undef, undef, 6);
print Dumper \@data;

@data = grep defined, @data;
print Dumper \@data;
```

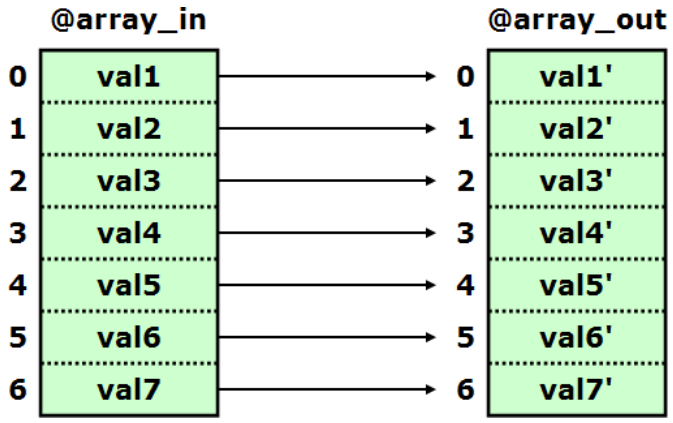
    This code outputs

```
$VAR1 = [
          1,
          undef,
          3,
          undef,
          undef,
          6
        ];
$VAR1 = [
          1,
          3,
          6
        ];
```

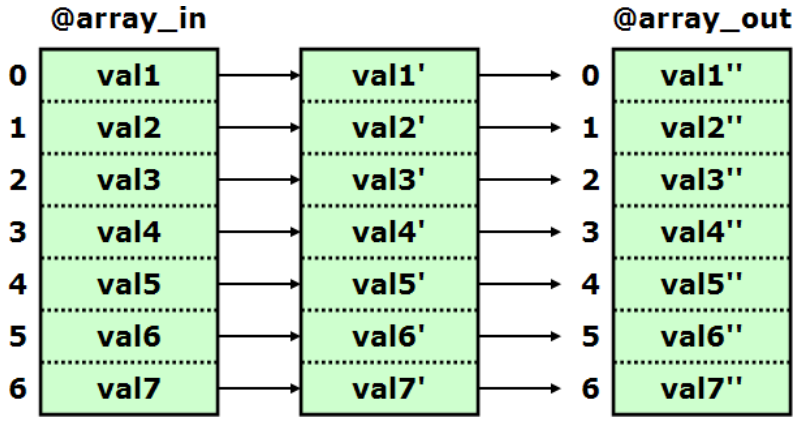
    - Unlike `sort`, the `grep` function needs a comma between a subroutine name and the data to examine.
    - The subroutine can be user-defined or built-in, but it must operate on `$_`.
- **`grep` is destructive**.
  - In each iteration, `$_` is aliased to an element the passed array. Modifying it without copying it first affects the original array element.

Arbitrary Transforms

- **map** evaluates a function or code block on each element of an array and returns an array consisting of the result of each execution.
- **map** is a very useful function.
  - Actually, **map** is **the most powerful function** in all of Perl.
  - Singlehandedly, it allows you to engage in true *functional programming*.
- How does **map** really work?
  - **map** sets **$_** equal to each element of the array passed to **map** in turn and evaluates the code block.
  - Each iteration of the code block may return a single value, an empty list **()**, or a list with multiple elements.
  - **map** collates the results of each iteration and flattens the results into a single array, which is what **map** returns.



  - Note that the code block is evaluated in list context.
  - Note that map takes and returns an array, so map statements can be chained together (and can be chained with **sort** and **grep**).



- A simple example uses **map** to return an array in which each element is twice the size of the original element:

```
1 my @numbers = (-3, 5, 15, 3.1416, 0, 18_000_000, -54.23, -9);
2 print "(@numbers)\n";
3
4 my @doubled = map { $_ * 2 } @numbers;
5 print "(@doubled)\n";
```

This code outputs

```
(-3 5 15 3.1416 0 18000000 -54.23 -9)
(-6 10 30 6.2832 0 36000000 -108.46 -18)
```

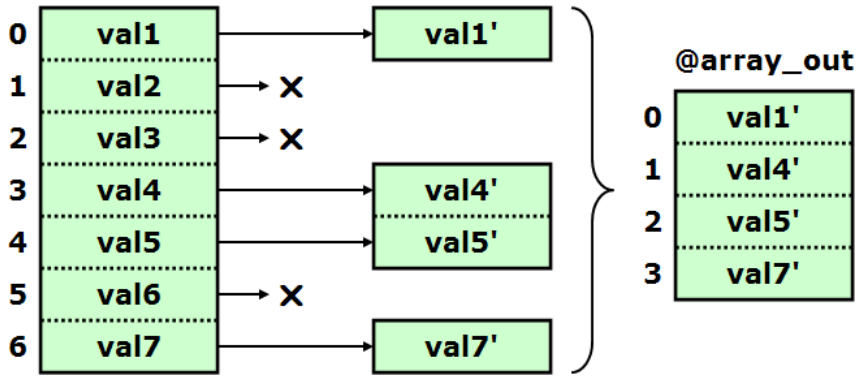
- As stated, **map** can be chained:

```
1 my @numbers = (-3, 5, 15, 3.1416, 0, 18_000_000, -54.23, -9);
2 print "(@numbers)\n";
3
4 # triple, format with 2 decimals, and append units
5 my @stuff =  map { $_ . " in." } map { sprintf "%2.2f", $_ } map { $_ * 3 } @numbers;
6 print "(@stuff)\n";
```

This code outputs

```
(-3 5 15 3.1416 0 18000000 -54.23 -9)
(-9.00 in. 15.00 in. 45.00 in. 9.42 in. 0.00 in. 54000000.00 in. -162.69 in. -27.00 in.)
```

- **map** does not affect the **order** of array elements, but it can affect the **number** of elements in the result.
  - By returning an empty list (that will be flattened away) for some elements, it can clip out elements like **grep** does:



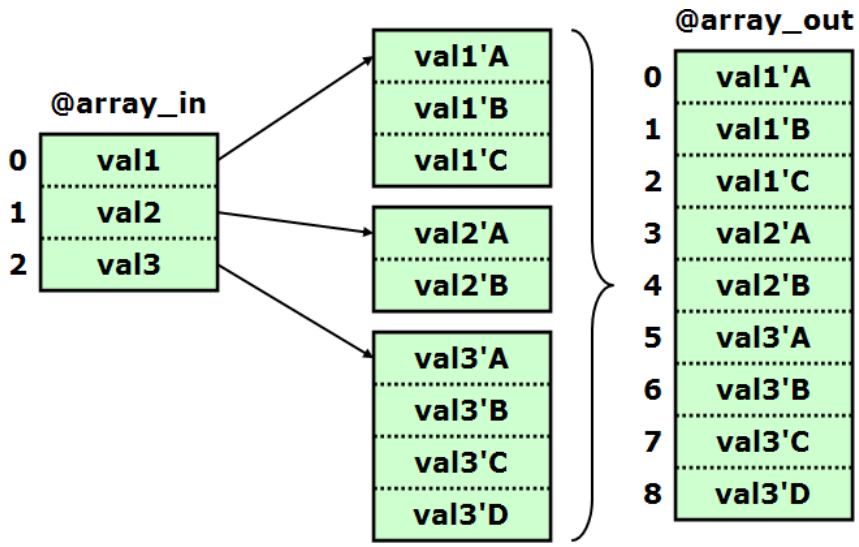
For example, the code

```
my @numbers = ( 1 .. 6 );
print "before: (@numbers)\n";
@numbers = map { $_ % 2 ? ($_ * 2) : () } @numbers;
print " after: (@numbers)\n";
```

outputs

```
before: (1 2 3 4 5 6)
 after: (2 6 10)
```

    - So **map** can emulate **grep**, but can transform the values at the same time.
  - **map** can also return more than one element for one or more iterations, which means the result may be **longer** than the input.



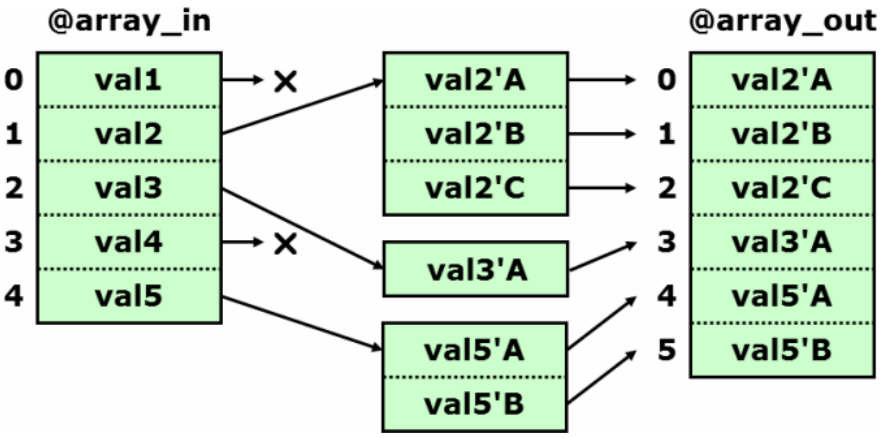
For example, the code

```
my @numbers = ( 1 .. 6 );
print "before: (@numbers)\n";
@numbers = map { (1 .. $_) } @numbers;
print " after: (@numbers)\n";
```

outputs

```
before: (1 2 3 4 5 6)
 after: (1 1 2 1 2 3 1 2 3 4 1 2 3 4 5 1 2 3 4 5 6)
```

- And, of course, if some elements return no elements, and others return one or more, the end result can get arbitrarily complicated:



- You can chain `map` statements together to affect the output in more complex ways:

```
1 my @numbers = ( 1 .. 6 );
2 print "before: (@numbers)\n";
3 @numbers = map { (1 .. $_) }
4            map { $_ % 2 ? ($_ * 2) : () }
5                @numbers;
6 print " after: (@numbers)\n";
```

outputs

```
before: (1 2 3 4 5 6)
 after: (1 2 1 2 3 4 5 6 1 2 3 4 5 6 7 8 9 10)
```

- Chains are **NOT** (necessarily) **COMMUTATIVE!**

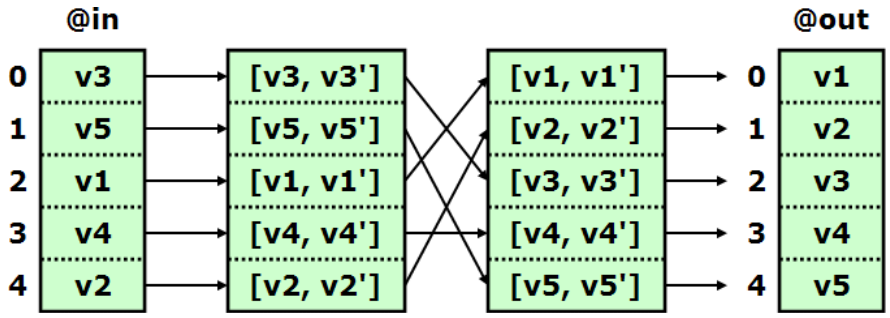
```
1 my @numbers = ( 1 .. 6 );
2 print "before: (@numbers)\n";
3 @numbers = map { $_ % 2 ? ($_ * 2) : () }
4            map { (1 .. $_) }
5                @numbers;
6 print " after: (@numbers)\n";
```

outputs

```
before: (1 2 3 4 5 6)
 after: (2 2 2 6 2 6 2 6 10 2 6 10)
```

- If you pass `sort` a comparator that performs a complex transformation of the original data, the comparator will perform this transformation every time it is called, and this will be very slow.
  - The solution is something called a *Schwartzian Transform*:
  - Use `map` to create an array of arrayrefs where each arrayref stores the transformed values and the original value.
    - You have to use arrays of arrayrefs because `map` and `sort` can only deal with plain arrays.
  - Use `sort` to sort the resulting map, using a comparator that simply examines the second element of each arrayref.
  - Use `map` to strip out the transformed value and return the original.
  - Remember that the steps are executed in the reverse order as they appear in the code.

- This is what it looks like:



- This is code to implement it:

```
# Fields are salary, contractor?, last name, first name, ID
my @data = (
  "65435,0,johnson,bill,FR-7146QW",
  "73452,0,Smith,Mr.,FW-7592RW",
  "73521,1,White,Don,RW-1346EQ",
  "88573,0,Lee,Mary,DF-5431FE",
  "57653,0,BROWN,FRANK,IT-6356VR",
);

print Dumper \@data;

# sort by last name (field #2), case-insensitively
my @sorted = map  { $_->[0] }
             sort { $a->[1] cmp $b->[1] }
             map  { [ $_, uc ((split ",", $_)[2]) ] }
             @data;

print Dumper \@sorted;
```

And it outputs:

```
$VAR1 = [
          '65435,0,johnson,bill,FR-7146QW',
          '73452,0,Smith,Mr.,FW-7592RW',
          '73521,1,White,Don,RW-1346EQ',
          '88573,0,Lee,Mary,DF-5431FE',
          '57653,0,BROWN,FRANK,IT-6356VR'
        ];
$VAR1 = [
          '57653,0,BROWN,FRANK,IT-6356VR',
          '65435,0,johnson,bill,FR-7146QW',
          '88573,0,Lee,Mary,DF-5431FE',
          '73452,0,Smith,Mr.,FW-7592RW',
          '73521,1,White,Don,RW-1346EQ'
        ];
```

- The intermediate data structure looks like this:

```
$VAR1 = [
          [
            '65435,0,johnson,bill,FR-7146QW',
            'JOHNSON'
          ],
          [
            '73452,0,Smith,Mr.,FW-7592RW',
            'SMITH'
          ],
          [
            '73521,1,White,Don,RW-1346EQ',
            'WHITE'
          ],
          [
            '88573,0,Lee,Mary,DF-5431FE',
            'LEE'
          ],
          [
            '57653,0,BROWN,FRANK,IT-6356VR',
            'BROWN'
          ]
        ];
```

- **`map` is destructive**. This is sometimes used to perform a *transform*, in the event the original values do not need to be preserved.

```
1 my @data = ("John", "Bill", "Frank", "Mary");
2 print "(@data)\n";
3
4 map { $_ = uc $_ } @data;
5 print "(@data)\n";
```

This code outputs:

```
(John Bill Frank Mary)
(JOHN BILL FRANK MARY)
```

  - This is also referred to as transforming or changing the array *in-place*, since no extra storage is required as would be for code like `@array = transform(@array);`
    - Discussion question: the previous line of code only includes one variable; why is extra storage required?
  - The call to `map` in the previous code is equivalent to

```
foreach (@data) {
  $_ = uc $_;
}
```

  - To preserve the original values, use a normal `map` call that returns a modified (copy of) `$_` but does not actually modify `$_`, and assign the result to a second array.
- One of `map`'s many uses is to easily convert an array into a membership hash using a single simple line of code: `%mem_hash = map { $_ => 1 } @array;`
  - Also, the `keys` of this hash are the same as the members of the original array, with duplicates removed (although order is not preserved).

A Final Thought

"Think Before You Type"

- Try to visualize what your program will be doing before you plunge in.
- This is good advice generally, but is especially important with data structures, which are often the most complex part of a program.
- Think about the natural relationships between data items, which will usually help you develop understandable data structures.
- But even more importantly, be sure to consider **HOW** the data structure will be used.
  - For example, if you'll be doing lookups by ID, the ID should be the hash key.
  - If you won't be looking up by ID, perhaps a different value would be a better key.
  - If you need random access to an element, a hash probably is better.
  - If you need to iterate over all the elements, an array might make more sense.

    - Use arrayrefs when the order of the data is important (remember hashes do not preserve order), including when you want to treat your data like a more restricted list (queue or stack).
- Remember that data structures can be optimized for storage space or speed, or for understandability. Unless you are trying to write a highly optimized program (in which case, why are you using Perl?), try to optimize data structures for understandability and ease of use.
    - Use hashrefs whenever you have a list of "identifiable" things (i.e. they have names), each of which has a certain property—including membership in a set.
    - Use arrayrefs of hashrefs to emulate arrays of records.
    - Don't be afraid to include arrayrefs as a value in a hashref if one of a "thing's" properties is a list of values.
        - Even if those values are in turn records!
    - In a hash, keep the "type" of the keys the same. For example, if you have a hash of students where ID points to a student record, don't include keys with the name of students as well.
        - But if you will also be looking up students by name, consider having a second data structure that maps names to IDs, or maps names directly to the student records.
    - In an array or hash, the elements should be the same type of thing, subject to interpretation.
        - For instance, don't store school records and student records in the same array.
        - But storing a person's name, age, SSN, and eye color in a hash is OK, because they are all the same "thing", in this case "attributes of a person".
            - Note the keys "**NAME**", "**AGE**", "**SSN**", and "**EYE_COLOR**" are all the same type, that type being "names of attributes of a person."
- Remember to use the substitution principal to help readability (and to avoid bugs if you change the layers of the data structure).
- Consider keeping auxiliary data structures. Suppose you have

```
$students = {
              'Tufts' => {
                           '178-28-1793' => {
                                              NAME => 'John Smith',
                                              DOB => '10-23-1980',
                                              SCHOLARSHIP => 1,
                                            }
                           '267-10-2793' => {
                                              NAME => 'Jane Smith',
                                              DOB => '03-03-1979',
                                              SCHOLARSHIP => 1,
                                            }
                             ...
                           '519-93-1004' => {
                                              NAME => 'Jack Smith',
                                              DOB => '03-17-1981',
                                              SCHOLARSHIP => 0,
                                            }
                         },
              'UNH' => {
                           ...
                        },
                };
```

and you want to determine a list of students who have scholarships. You need code something like this:

```
foreach $school (keys %{$students}) {
  foreach $student (keys %{$students->{$school}}) {
    if ($students->{$school}->{$student}->{SCHOLARSHIP}) {
      # student has a scholarship
    }
  }
}
```

- The multi-level hash shown might be the best data structure for your program, depending on what other operations will be performed on the data.
- But this specific task of doing something with all students with scholarship is a little complicated.

- It might be easier to generate a membership hash or simple list of all students with scholarships. This can be done once (as a data structure transformation) before you need it, or even better, it can be constructed as an auxiliary data structure as you construct `$students`.
- If you choose to keep data (or references to it) in multiple locations, be sure to keep the locations synchronized when you modify data, particularly when you delete records.

- Consider storing a sub-structure's key as part of that structure:

```
$students = {
              'Tufts' => {
                           '178-28-1793' => {
                                              NAME => 'John Smith',
                                              DOB => '10-23-1980',
                                              SCHOLARSHIP => 1,
                                              SCHOOL => 'Tufts',
                                              SSN => '178-28-1793',
                                            }
                           '267-10-2793' => {
                                              NAME => 'Jane Smith',
                                              DOB => '03-03-1979',
                                              SCHOLARSHIP => 1,
                                              SCHOOL => 'Tufts',
                                              SSN => '267-10-2793',
                                            }
                             ...
                           '519-93-1004' => {
                                              NAME => 'Jack Smith',
                                              DOB => '03-17-1981',
                                              SCHOLARSHIP => 0,
                                              SCHOOL => 'Tufts',
                                              SSN => '519-93-1004',
                                            }
                         },
               };
```

  - This is so that when a lower-level record (in this case, representing a student) is considered on its own, the important information encoded by the keys is also available. This is particularly important when passing a record to a subroutine:

    ```
    register_student($students->{$school}{$ssn});
    ```

    If the keys are not included in the record, the subroutine would need those values passed as extra arguments. The subroutine itself would be more complicated, and the call to it would be much less elegant:

    ```
    register_student($school, $ssn, $students->{$school}{$ssn});
    ```

- A specific form of keeping some data seperately is *indirection*. This is used when you need to look up data based on two (or more) different keys, or the transform of a set of keys that is available.
  - For example, suppose you had a data structure `$sales_data` that was indexed by names of months, so that a lookup into it resembled

    ```
    $sales_records = $sales_data->{January}
    ```

  - But the month is often stored as an integer. The solution is to keep a map of integer months to month names, in this case in an array:

    ```
    @month_name = ('January', 'February', 'March', 'April',
                   'May', 'June', 'July', 'August', 'September',
                   'October', 'November', 'December');
    ```

  - Now you can look up sales records like this:

    ```
    $sales_records = $sales_data->{$month_name[0]}
    ```

  - You might also use indirection if you had two sets of related keys, such as SSN and Employee ID. Your records should only have one type of keys, so choose on (in this case, say ID) and then create an auxiliary table that maps Employee ID to SSN. Now you can look up records in one of two ways, depending on which key you have available:
    - `$employees->{$employee_id}` using the employee ID number
    - `$employees->{$employee_id_by_SSN{$SSN}}` using the SSN, indirectly

Answers to discussion questions:

- What does each of the following lines of code do?
  - **`$foo = [1,2,3];`** creates a reference to an anonymous array containing (1, 2, 3) and assigns it to **`$foo`**
  - **`@foo = (1,2,3);`** creates an array named **`@foo`** containing (1, 2, 3)
  - **`$foo = (1,2,3);`** assigns 3 to **`$foo`** (via the comma operator)
  - **`@foo = [1,2,3];`** creates an array named **`@foo`** with one element, a reference to an anonymous array containing (1, 2, 3).
    - This is probably not what you intended.
- To check orthogonality, determine the size of the first sub-array and then loop through all the sub-arrays, making sure each one is also that size.
- The code **`%hash = ( $hashref1, $hashref2 )`** will produce a one-element hash where the key is a string like "**`HASH(0x139e87)`**" and the value is (a copy of) the reference stored in **`$hashref2`**. This is probably not what you meant!
- Code such as
  `@array = transform(@array)`
  requires extra storage because the return values from **`transform()`** are all stored as an anonymous array before being assigned to **`@array`**.
