Cut-to-the-Chase Series - **Perl Programming**

References

Perl Programming – References

Table of Contents

THIS MODULE.. 1

INTRODUCTION TO SUBJECT MATTER.......... 1

GETTING STARTED... 2

REFERENCES TO SCALARS, TERMINOLOGY, AND JUSTIFICATION FOR REFERENCES.............................. 3
Reference 4
Referent... 4
Dereferencing (getting the actual data) 5
And the point would be? 5
Closures... 7
Who keeps referring to me? 8
References are always true. 9
I just want the variable type of my reference........ 9
REFERENCES TO ARRAYS 10
An array of scalar references............................. 12
Reference to a list of arrays 12
Array of array references................................... 16
REFERENCES TO HASHES 17
Reference to a simple hash................................. 18
REFERENCES TO SUBROUTINES 19

ANONYMOUS STUFF 21

ARRAYS 21
Creating... 21
Access 21
Determining Size.. 22
Adding elements... 22
HASH – SINGLE KEY... 22
Dereferencing an anonymous hash 23
Getting a value for an anonymous hash key....... 23
Looping through an anonymous hash 23
HASH – MULTIPLE KEYS 24
Return hash keys 25
Accessing individual elements for a key............. 25
Modifying data... 25
Returning all hash keys and their elements/data 25
A Subtle Point regarding the syntax................... 26
SUBROUTINES 31
Passing values to an anonymous subroutine...... 31
Passing references to an anonymous subroutine 32
Returning an array from an anonymous subroutine 33
Returning a scalar from an anonymous subroutine 33

MISC. STUFF.. 34

ARRAY OF HASH REFERENCES............................... 34

WHAT NEXT?... 35

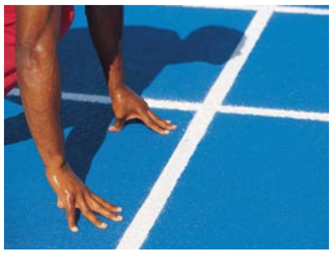
References

It is better to know nothing than to know what ain't so.

ICON KEY
Toy Code
Hands-on
Important Info
Be Careful

This Module

It is assumed that you have already finished all of the foundation topics, or have a very solid understanding of Perl.



Introduction to Subject Matter

Just as new programmers generally get their butt kicked with loops, this tends to be a subject that many programmers find compelling. In fact, one of Java's claims to fame as a language is that it does not have those 'evil' pointers. If you have worked with pointers in languages like C or C++ then you should not have much difficulty here. References are similar enough to pointers for me to make this analogy and these assumptions.

References in Perl can allow you to create a level of abstraction in your programs that would not otherwise be possible. Understanding references is a prelude to object oriented programming in Perl. Additionally, references can be used to create complex data types, algorithms, and a host of other uses.

In my estimation one of the most compelling reasons for using references is that Perl flattens data structures in arrays and hashes. As an example, if we create an array that contains three other arrays as its elements we lose the context of each array within the array (not to mention that what we have is a copy of the data from these three arrays).

It is my goal to stick to writing toy code. In doing so you may at times ask yourself why you wouldn't just want to write code that does not use references. After all, code using references can get pretty abstract (which is precisely the point of references!).

I will try my best to give you enough context for these programs that you can get a sense of application. Having been in this field for the past fifteen years I realize that some of you are going to struggle mightily with the subject matter. The biggest struggle for many of you will be in understanding why you would want to use references. If this is you then the following paragraphs will hopefully bring you some perspective.

Programming and software engineering have evolved to a place where we should strive to not re-invent the wheel, especially if a set of tires already exists. Also, as a programmer, you already know that we live in one of the few professions where borrowing other peoples work is not considered stealing, it is considered the thing to do. What other profession so readily shares its work? If the rest of business were to be modeled after our discipline I fear capitalism would fall.

If our goal is to share code and attempt to reduce size (a form of complexity) through the reuse of logic then we need an elegant means. Perl has a ton (literally) of code to reuse. You should for no other reason learn this subject matter if only to become familiar with the syntax so that you can (other than blindly) begin to explore the enormous code base that is out in the public domain for you to reuse.

Getting Started

A reference is a value that holds the address of another value. Each variable that you define takes up a chunk of memory. As you already know, variables in Perl can be scalars, arrays, hashes, subroutines, and typeglobs. It is possible to create references for each of these types. I will be focusing on the first four in this Module.

In keeping with Perl's theme of "there is more than one way to do it", you will find a number of different ways to both reference and dereference a reference (that sounds a bit strange). I will explore the myriad of ways throughout the module. My goal in doing this is to expose you to the various methods so that you can read other peoples' code. You will find that I will tend to code in a particular style once I have shown you the variations.

HANDY TABLE OF REFERENCE SYNTAX

SYNTAX	EXAMPLES
$<ref_var> = \$<scalar>	$ref = \$myvar
$<ref_var> = \@<array>	$ref = \@myarr
$<ref_var> = \%<hash>	$ref = \%myhsh
$<ref_var> = \&<sub>	$ref = \&mysub

Detect a general theme here?

References to scalars, terminology, and justification for references

It is probably best that we start with scalars.

Here is our first program to dissect.

REFSCALAR1.PL

```
my $a = "some text";
#  reference      referent
my $b        =    \$a;
print \$a , "\n";
print $b , "\n";
print \$b , "\n";
print $$b    , "\n"; #or
print ${$b}  , "\n";
```

Print Results

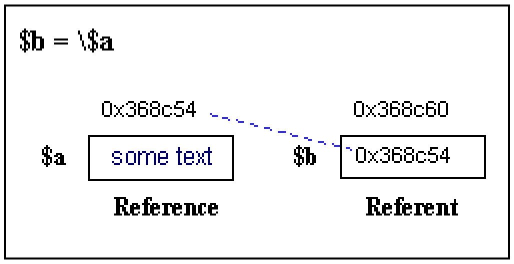
\$a	SCALAR(0x368c54)
$b	SCALAR(0x368c54)
\$b	SCALAR(0x368c60)
$$b	some text

${$b}	some text

I call your attention to the third line of the program. This is where we are creating a reference

```
my $b = \$a;
```

There are two things involved in creating a reference: the reference (our hook to the thing) and the referent (the thing we want to refer to). What is stored in the reference is the address of the referent. Perl is also kind enough to tell us what type of "thing" we are referring to. I think a diagram would be in order now.



Reference

Our reference is going to be a scalar. The scalar will hold the address to the referent and the data type of the referent. As you can see in the diagram above, this is the case.

You see that when we print out the value of $b that we get, **SCALAR(0x368c54)**.

Referent

The referent is the "thing" we are referring to. To refer to another variable (at least for the moment) you use the backslash.

Syntax

```
\<variable symbol><variable name>
```

Examples

```
\$str

\@arr

\%hsh

\&mysub
```

Dereferencing (getting the actual data)

Printing out the value of the reference variable is fine, but how do we use the reference to get at the data?

Here are two ways.

DEREFERENCE SCALAR

SYNTAX	EXAMPLES
$$<scalar>	$$myvar
${$<scalar>}	${$myvar}

```
$$b

${$b}
```

Both of these will get the data that is stored in $a. Both conventions are fine to use. I prefer to use the ${$<scalar>} syntax, though to be 'straight-up' I cannot give you a real good reason why.

And the point would be?

As I said previously, one of the main points of using references is to achieve a greater degree of abstraction in your program. You can also use references to achieve a principle that is referred to as encapsulation in object oriented programming.

Many people approach this subject asking this question for the first time. The attitude is 'I can accomplish what I need to achieve without using references'. And, 'my code is clearer to me and others when I do not use references'. Certainly this subject is one that could take up a long and winding discussion (and remember this is the cut to the chase series). My personal opinion and preference is to write code without references where there appears to be no apparent gain. In instances where you need to manage complexity in terms of data structures whose type is not known at run-time, references make a great deal of sense. These are just two reasons offered, there are many more.

Before the module is finished I will hopefully have demonstrated some good uses of this subject. Keep in mind that for a much of this I will be focused on the mechanics of the subject, not its practical application.

A simple example of abstraction

ⓘ Scope remains as long as there is a reference and Perl keeps track of who has a reference to some 'thing (referent)'…and Perl does garbage collection (memory management).

This piece of information is certainly good news for us if our goal is to write code that gives us some degree of encapsulation (this is a ten dollar word that mean hiding code and data). You can think of encapsulation as the “black box” concept for code. Though with Perl, the concept is more “white box” with the understanding that reasonable people tread lightly in those areas, even though they have access to them.

Take a look at the following example. Please note!!! The whole point of the following example is to demonstrate that a variable is not destroyed as long as there is a reference to it. The goal is to prove to you in code the statement made above. That is…

Scope remains as long as there is a reference

🖫 REFSCOPE.PL

```
my $c;
{
  my $a = "foo";
  {
  $c = \$a;
  }
}
print ${$c};
```

Print Results

${$c}	foo

Without references and based on your understanding of block scope, $a should only exist for the duration of the block to which it is contained. Yet the program, as written, will print “foo” to the console. Why? Within the block we created a reference with our global (package scope) scalar $c to $a. Since $c has package scope it is still around after the block. Since $c is a reference to $a (holds the address of $a), Perl will not garbage collect (destroy) $a, until the reference to it goes away.

It is important to note that while we have access to the value of $a through dereferencing $c, we do not have access to $a directly. In other words (I mean in code),…

```
      print $a; #…outside of block where a lives
```

…will result in a compile error.

Since we are talking about encapsulation here, I am going to take a slight side road (keep in mind this subject is not at all specific to references to scalars. It just seemed an appropriate time to take a side road as we are discussing encapsulation and introduce you to the subject of closures). While closures do not deal explicitly with the concept of references, they do indeed deal with implicit references. A simple example can really serve to demonstrate the principle of encapsulation in Perl.

Closures

🖫 CLOSURE.PL

```
{
  my $str = "Scope of \$str is confined to this block";
  sub print_str
  {
   return $str;
  }
}
print my $strO = print_str();
```

Print Results

$str	Scope of $str is confined to this block

In examining the code you can see that the scalar $str is buried within a block. It's scope therefore is limited to the duration of the block. So how the heck are we able to call the sub from outside of the scope and print the value of string. Certainly some magic seems to be happening here?

While magic does appear to be the soup de jour this is just not the case. What we have here is what is referred to as a closure. **A closure is nothing more than a subroutine that refers to one or more lexically scoped variable(s) that are declared outside its scope.**

In this example you can see how we could use a subroutine within a block to allow us access to a set of variables (remember, including other subroutines!, arrays, etc…). Such a technique gains us the ability to hide data and code exposing it through a subroutine. Also note that while a subroutine is considered a variable, it does not have to follow the same rules of block scope that our data variables are restricted to. In other words the compiler will not complain about scope at compile time. Case in point, our example has our sub as a sub block within a block, yet we can call it without (ourself) having to create a reference to it.

Later in this module I will show you a more complex example that uses a closure and a reference to an anonymous subroutine.

Who keeps referring to me?

Since Perl keeps track of all the references to a referent, this means that it is possible for the referent to figure out whom the reference is. This of course has some interesting implications with respect to writing error handlers or encapsulating objects. The intent here is not to go off onto either tangent, rather to show how to figure out who the referent is by using the Carp module.

USECARP.PL

```
use Carp;
&sort_array;
sub sort_array
{
  (my $package, my $file, my $line) = caller();
  print "\n$package $file $line\n";
  return;
}
package mypack;
&main::sort_array;
```

Print Results

"\n$package $file $line\n";	main usecarp.pl 8
"\n$package $file $line\n";	mypack usecarp.pl 18

In our subroutine we assign a list of three variables that will be returned from caller( ). It just so happens the first three items (there are more items, see the Carp module for more information). Returned (see the longmess( ) subroutine) are the package, the file, and the line number from where the call came.

References are always true.

As you should already know there is truth associated with scalars. It just so happens that a reference will always evaluate true.

🖫 REFTRUTH.PL

```
my $a;
my $b = \$a;

if ($b)
{
 print $b . "\n";
} else {print "\$b is false";}
undef $b;
if ($b)
{
 print $b;
} else {print "\$b is false";}
```

Print Results

$b	SCALAR(0x368c54)
"\$b is false";	$b is false

I just want the variable type of my reference

There is a function that will just return the data type of the reference.

🖫 GETREFTYPE.PL

```
my $myreferent = "Sally";
my $myreference = \$myreferent;
my $reftype = ref $myreference;
print $reftype;
```

Note if you do not specify the reference, "ref" uses $_ .

Print Results

$reftype	SCALAR

We are now going to round out this section by showing how to create a reference to arrays, hashes, and subroutines.

References to Arrays

The mechanics of creating a reference to an array are not that markedly different than creating a reference to a scalar

DEREFERENCE ARRAYS

SYNTAX	EXAMPLES
@$<scalar >	@$myarr
@$<scalar >[elem]	@$myarr[2]
@$<scalar >[elem]	@{$myarr}->[2]
@{$<scalar >}	@{$myarr}
@{$<scalar >} elem]	@{$myarr}[2]
@$<scalar >[elem]	@{$myarr}->[2]

Here is a simple example that demonstrates the essentials

REFARR1A.PL

```
my @arr = qw(a b c d);
my $x = \@arr;
$_=0; #to shut-up the compiler
print @$x[2]     . "\n"  ;   #or
print @{$x}[2]   . "\n"  ;   #or
print @{$x}->[2] . "\n"  ;   #does the same thing
#change a value through the reference
@{$x}->[2] = 3;
```

```
print @{$x}->[2] . "\n"  ;
#loop through the array via the reference
while (@{$x})
{
 print @{$x}->[$_] . "\n";
 shift @{$x};
}
#our array is empty now!
```

Print Results

`@$x[2]`	c
`@{$x}[2]`	c
@{$x}->[2]	c
@{$x}->[2]	3
(loop) @{$x}->[$_]	a
@{$x}->[$_]	b
@{$x}->[$_]	3
@{$x}->[$_]	d

Since we have a few new items of syntax, I will address a few things in the code. First is the peculiar looking **->** . This is the arrow operator (aka the infix operator) and can be used to dereference an array. This operator comes in quite handy when working with anonymous arrays (as ye shall see later). The $_=0 assignment at the beginning is to keep us from getting un-initialized variable warnings. As you can see we use the ever handy and seemingly always available $_ to loop through the array.

Also note that our "while" loop is based on **@{$x}**. Remember earlier I said that a reference is always true. This is indeed true. But our reference is **$x** not **@{$x}** . **@{$x}** is our dereferencer, and it will be true as long as the array has elements. Since we are removing elements as part of the loop, we can execute the loop on this premise. The dereferencer has truth equivalent to @arr.

An array of scalar references

REFARRSCALARS.PL

```
my $str  = "one";
my $str2 = "two";
my $str3 = "three";

my @arr = (\$str, \$str2, \$str3);

print @arr->[0]        . "\n";  #value of @arr[0]
${$arr[0]} = "hanna";  #change $str via reference
print ${$arr[0]};      #print val of $str via ref

print "\n" . $str;  #see indeed we changed $str
```

Print Results

`@arr->[0]`	SCALAR(0x368c54)
`${$arr[0]}`	hanna
`$str`	hanna

Reference to a list of arrays

It is possible to create a reference to a list of arrays

REFARR1B.PL

```
my @arr  = qw(1 2 3);
my @arr2 = qw(a b c);
my @arr3 = qw(A B C);

my $aarr = [\@arr, \@arr2, \@arr3];
```

```
print @{$aarr}->[1]          . "\n";
print \@arr2                 . "\n\n";

print \@arr2->[1]            . "\n";
print \@{$aarr}->[1]->[1]    . "\n\n";

print @{$aarr}->[1]->[1]     . "\n";
print $arr2[1]               . "\n\n";

$arr2[1] = "bee";
print @{$aarr}->[1]->[1]     . "\n\n";

@{$aarr}->[1]->[1] = "b";
print $arr2[1]               . "\n\n";
```

Print Results

`@{$aarr}->[1]`	ARRAY(0x36aeb0)
`\@arr2`	ARRAY(0x36aeb0)
`\@arr2->[1]`	SCALAR(0x368984)
`\@{$aarr}->[1]->[1]`	SCALAR(0x368984)
`@{$aarr}->[1]->[1]`	b
`$arr2[1]`	b
`@{$aarr}->[1]->[1]`	bee
`$arr2[1]`	b

Building such a structure raises some interesting possibilities as well as some interesting syntax. What I have done is assigned a scalar to a list of array references.

```
my $aarr = [\@arr, \@arr2, \@arr3];
```

Such an assignment gives me the ability to dereference any of the array references through my scalar reference. As you can see from the coded fragment below, both statements are one in the same.

```
print @{$aarr}->[1]

print \@arr2
```

`@{$aarr}->[1]`	ARRAY(0x36aeb0)
`\@arr2`	ARRAY(0x36aeb0)

As a quick aside, **@{$aarr}->[1]** could also be written as…

@{$aarr}[1]

My preference is to use the arrow dereferencer as I feel it provides a more consistent means of syntax. I will demonstrate what I mean by this in a little while. We also have the ability to get even more granular with our reference.

```
print \@{$aarr}->[1]->[1] ;
```

which is also the same as saying…

```
print \@arr2->[1]  ;
```

Either of these statements yields the address of the scalar within the array. To get the data itself through the array we could say…

```
print @{$aarr}->[1]->[1] ;
```

…which gets us the second element of the second array (which is the letter 'b').

Now I promised to give you more information on why I choose to use the arrow dereferencer. The following statements yield the same result as the one above.

```
print @{$aarr}[1]->[1]    ;

print @{$aarr}->[1][1]    ;
```

But this one will result in a compiler error.

```
print @{$aarr}[1][1]      ;
```

You can wrestle with the semantics of this and draw your own conclusion. I just felt you needed to be exposed to the open syntax that is allowed.

Finally, you can see that if we change any of the data through the reference that we are actually changing the array element itself (which is the whole point of a reference).

```
@{$aarr}->[1]->[1] = "b";

print $arr2[1]
```

At this point you might be asking yourself what all the fuss is about. Why can't I just create an array of arrays to solve the problem.

While it is certainly possible to create an array of arrays, what ends up happening is that you create a flattened data structure. Also, the array that contains the other arrays contains them by value. Meaning, our data can get out of sync. Consider the following code which serves to demonstrate my point.

ARRARRS.PL

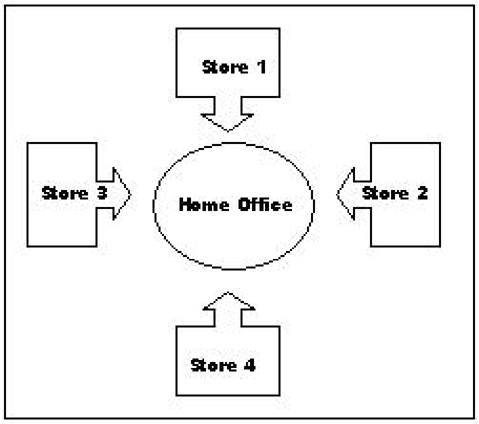
```
my @arr2 = qw(a b c);
my @arr3 = qw(A B C);

# I am only copying the data
my @arr = (@arr2, @arr3);

# see I am flat
print $arr[3]          . "\n";

$arr3[0] = "one";

print $arr[3]          . "\n";  #Still 'B'
print $arr3[0]         . "\n";  #But I have changed
```

Print Results

$arr[3]	A

`$arr[3]`	A
`$arr3[0]`	one

Of course, we could write some code that kept the two in sync. In other words, every time we changed @arr2, or @arr3, we could refresh @arr by saying **@arr = (@arr2, @arr3);** . While this is certainly acceptable, it would put an undue amount of burden on the program, and create yet another place for us to introduce bugs. I hope you are having an "aha" moment right now as to how references can be used to serve a useful role in code. If you have not, consider this high level example which is analogous to the idea we are exploring here.



Consider a scenario where we had four retail outlets and a home office. The retail outlets reported sales back to the home office as part of a nightly dump across a network. We decided that we wanted a means of tracking sales more dynamically. We could implement a system that for every time we had a sale at one of our stores we sent a message back to the home office containing the pertinent information. This would still give us autonomy, but would be a duplication of information. A better solution would be to implement a system where the sales data at the home office and at the stores were one in the same.

If this still does not help, I am not sure what else to say except that sooner or later you will encounter a situation in programming where you will hopefully have that "aha" moment, and hopefully remember this subject as a solution.

Array of array references

This works in a similar fashion to our reference to a list of arrays, though the syntax for accessing elements is a bit simpler. And, like the reference to a list of arrays, we do not end up with a flattened structure.

REFARR1C.PL

```
my @arr  = qw(1 2 3);
my @arr2 = qw(a b c);
my @arr3 = qw(A B C);
my @aarr = (\@arr, \@arr2, \@arr3);
print @aarr            . "\n";
print $aarr[0]         . "\n";
print $aarr[1]->[1]    . "\n";
```

Print Results

`@aarr`	3
`$aarr[0]`	ARRAY(0x368c60)
`$aarr[1]->[1]`	b

Also keep in mind that we can create references to data types other than arrays from within our array.

References to Hashes

Hashes are an enormously efficient and useful way of storing data. First, they allow us to work with data in manner that is more in tune with our own set of linguistics. What I mean is that the key is not an esoteric number, it is in language that can be meaningful to us. Second, and highly related to the first, we have a quick means of indexing and retrieval if we know exactly what we are looking for. This is unlike the array, where we need to loop until we find what we are looking for.

Hashes are a flat structure by themselves. You get a key and a data element. If you are a business programmer you should already understand that quantifying data is more often than not a myriad of complex hierarchical and relational organization. Hashes by themselves do not readily help us to work in these problem spaces. But enter references and the world becomes our oyster.

DEREFERENCE HASHES

SYNTAX	EXAMPLE
%$<scalar >	%$myhsh
%{$<scalar >}	%{$myhsh}

Reference to a simple hash

This is pretty basic stuff if you have just come from the previous two sections. It shows the syntax of creating a hash reference and dereferencing a hash.

REFHSH1A.PL

```
my %phone = (area_cde  => "254",
             exchange  => "455",
           extension => "1223"
          );

my $phone = \%phone;
print $phone                . "\n";
print ${$phone}{extension}  . "\n";

my @keys = keys(%{$phone});
print "@keys"               . "\n";

my @values = values(%{$phone});
print "@values"             . "\n";

my $key="";
my $val="";
while (($key, $val) = each(%{$phone}))
{
  print "$key => $val\n";
}
```

Print Results

`$phone`	HASH(0x368c60)
`${$phone}{extension}`	1223
`$aarr[1]->[1]`	1223

`“@keys”`	exchange area_cde extension
`“@values”`	455 254 1223
`(Loop) "$key => $val\n"`	exchange => 455
`"$key => $val\n"`	area_cde => 254
`"$key => $val\n"`	extension => 1223

References to Subroutines

DEREFERENCE SUBROUTINES

SYNTAX	EXAMPLE
&$<scalar >	&$mysub
&{$<scalar >}	&{$mysub}

By now you should have some awareness that using references gains you these advantages:

- Allows multiple entities to point to the same data
- Preserves the sovereignty of the data structure or ‘thingy’ you are referencing

After this discussion, we should be able to add another bullet to our rationale for using references.

- Allows you the ability to develop code using more object oriented techniques

So as not to be too much of a windbag here I am going to make the presumption that you are writing subroutines that accept arguments and return something. This assumption will keep us to a single code example (cut-to-the-chase).

SUBREF1.PL

```
my @arr3 = qw(1 4 5 8 3);
my $sort_arr = \&sort_array;
print my @arr2 = $sort_arr->(@arr3);
```

```
sub sort_array
{
  my @srt = @_;
  @srt = sort @srt ;
  return @srt;
}
```

Note: this is only a partial listing of the actual code example.

As you can see in the second line, we are able to create a reference to a named subroutine in the same manner as we did for referencing arrays, scalars, and hashes.

```
my $sort_arr = \&sort_array;
```

If we were to print the contents of our scalar $sort_arr we would see something like:

```
CODE(0x1a9ef18)
```

We can now make calls to the subroutine via our reference. There are a number of syntaxes you can use to do this. The first one presented is the most common form you will encounter.

```
print my @arr2 = $sort_arr->(@arr3);
```

These two forms are also acceptable.

```
&$sort_arr(@arr3)

#or

&{$sort_arr}(@arr3)
```

One may question the rationale for creating a reference to a subroutine instead of just accessing the subroutine itself. One motivation is that a reference can supply a degree of encapsulation to the subroutine. Of course, in this case, where our subroutine has a name, we could still access the subroutine directly. This makes a nice segue into the next subject area.

Anonymous Stuff

If references were not abstract enough, it is possible to further abstract code through the creation of anonymous references. Up until now we have created references to named data types (please note I am including a subroutine as a data type).

I do not care to go off on a long and winding dissertation on the subject of reuse or object oriented programming; but a key benefit of using references is it allows us to achieve a greater degree of abstraction. This will allow us to better achieve reuse or the principles of OOP (object oriented programming) should we decide to write object oriented Perl programs.

For now, I am merely concerned that you get the mechanics of the subject matter. Later we will delve into some (hopefully) practical examples. It is also reasonable to expect that you may have an "aha" moment and see application just from the examples provided here.

I have seen these techniques used when good old conventional code would suffice. My personal bias is to only use references and anonymous references when I cannot readily get the job done any other way. But, in the spirit of Perl and programming in general it is my belief that you should write code in the manner that works best for you, your endeavors and your organization.

Arrays

🖫 ANONREFARR1a.PL

The following example demonstrates how to create an anonymous array

Creating

```
my $anonarr = [1, 2, 3];
```

If we print $anonarr we see that $anonarr is a reference to our anonymous array.

```
ARRAY(0x1a7f164)
```

If we want to get at an individual element in the array you can code…

Access

```
$anonarr->[2]
```

This will return the third element. If you want to access the whole array.

```
print @{$anonarr};
```

Remember, we are dereferencing an array not a scalar. Our scalar is a reference to an anonymous array. Yes, this can be quite abstract.

We can perform all of the operations on an anonymous array that we can on a regular array or a reference to an array.

Determining Size

```
my $sz = @{$anonarr};
```

Adding elements

```
$anonarr->[3] = "Four";
```

```
my $max = 10;

while ($sz <= $max)

{

  $anonarr->[$sz++] = "new element$sz";

}

print @{$anonarr};
```

Hash – Single key

The following is a simple example of an anonymous reference to a hash.

🖫 ANONREFHSH.PL

```
my $phone = {
             area_cde  => "425",
             exchange  => "277",
             extension => "1969"
           };

print $phone . "\n";
print $phone->{extension} , "\n";
print %$phone,"\n";  #key and values
# or, a with a little more control
while ( (my $key, my $val) = each (%$phone) )
```

```
{
  print "$key -> $val\n";
}
```

Print Results

`$phone`	HASH(0x1a7f120)
`$phone->{extension}`	1969
`%$phone`	area_cde425exchange277extension1969
`"$key -> $val\n"`	area_cde -> 425 exchange -> 277 extension -> 1969

In respect to general semantics we know that a phone number by itself is not in its most atomic form with respect to its data columns. In other words, a phone number is an ordered aggregation of three discrete fields; area code, exchange, and extension. We certainly could build this data structure by creating a hash named phone. We can also accomplish this feat using an anonymous hash named phone.

Dereferencing an anonymous hash

```
%$phone
```

area_cde425exchange277extension1969

Getting a value for an anonymous hash key

```
$phone->{extension}
```

```
1969
```

Looping through an anonymous hash

```
while ( (my $key, my $val) = each (%$phone) )

{
  print "$key -> $val\n";

}
```

```
area_cde -> 425

exchange -> 277

extension -> 1969
```

Hash – Multiple keys

It is possible to create even more complex data structures under the guise of an anonymous hash. Staying in continuity with our previous example, let's extend it to include both work and home phone numbers.

🖫 ANONREFHSH1a.PL / ANONREFHSH1b.PL

Lets first look at our data structure. It is the same in both programs.

```
my $phn = {
         'home' => {
                 area_cde  => "254",
                 exchange  => "455",
                 extension => "1223"
             },
         'work'  => {
                 area_cde  => "312",
                 exchange  => "756",
                 extension => "7643"
             }
          };
```

As you already know $phn is a reference in and of itself.

```
$phn
```

HASH(0x1a755b0)

Lets dereference $phn and see what we get.

```
%$phn
```

homeHASH(0x1a7f120)workHASH(0x1a75538)

Hmmmmmm?

What you may not realize at this point is that both 'work' and 'home' are also references. So we have a reference to other references.

Here are some basic techniques for dealing with these.

ANONREFHSH1b.PL

Return hash keys

```
my @keys = keys(%$phn)
```

```
home work
```

Accessing individual elements for a key

```
$phn->{'work'}->{'area_cde'}
```

```
312
```

Modifying data

```
$phn->{'work'}->{'extension'} = "4432"
```

Returning all hash keys and their elements/data

ANONREFHSH1a.PL

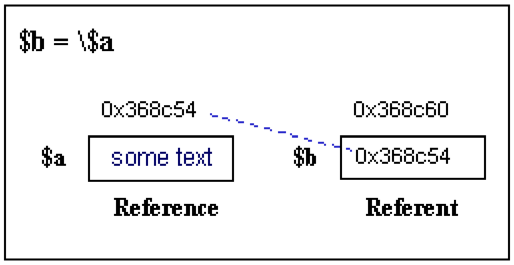
Since we are dealing with a hierarchical structure, we have a nested loop.

```
foreach $key (sort keys (%$phn))

{

   my $ref = $phn->{$key};

   print "$key => \n";
   foreach $key2 (keys (%$ref))

   {

      my $value = $ref->{$key2};

      print "$key2 => $value \n";

   }

}
```

```
home =>

area_cde => 254

exchange => 455

extension => 1223

work =>

area_cde => 312

exchange => 756

extension => 7643
```

A Subtle Point regarding the syntax

With all this cryptic syntax flying around, I want to belabor a point that can be quite troubling when working with references. By now it should be ingrained what a reference and a dereference are. We should take a look at a previous picture to quickly review this concept.



In this example, our reference is to a scalar. If we choose to obtain the value of $a through the referent $b we would code…

```
$$b
```

…thus allowing us to resolve our reference and see what is actually stored in this location.

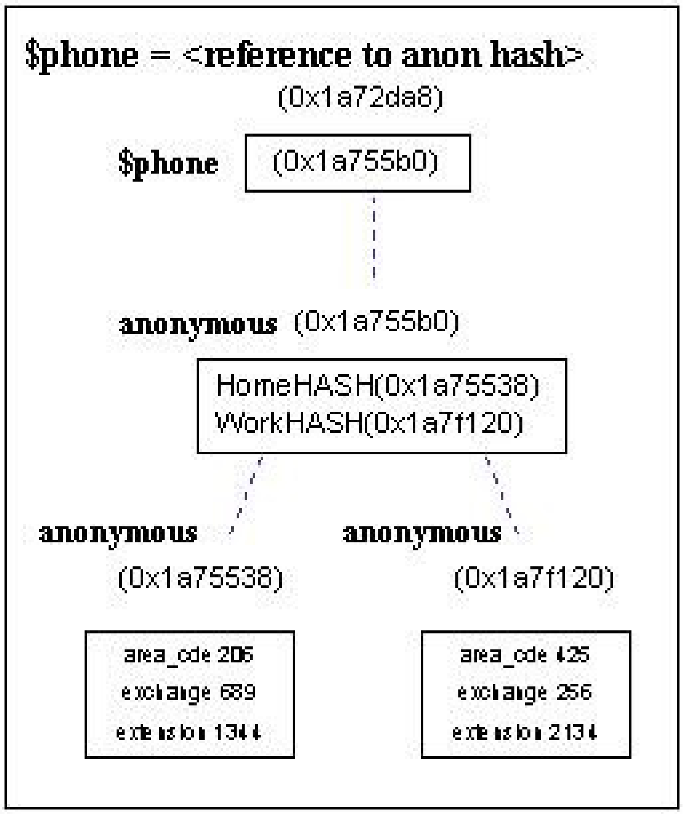
With dereferencing we need to remain cognizant of the type of structure we are dereferencing. This may seem obvious but it is easy to get into trouble when dealing with complex structures. For example…

🖫 ANONREFHSH1c.PL

```
my $phone = {
           Work =>
                    { area_cde  => "425",
                      exchange  => "256",
                      extension => "2134"
                    } ,
           Home =>
                    { area_cde  => "206",
                      exchange  => "689",
                      extension => "1344"
                    }
          };

print my $a = \$phone . "\n";
print $phone . "\n";
print %$phone ;
print "\n";
print %$phone->{Work};
print "\n";
print %$phone->{Home};
print "\n";
print $$phone{Home};
print "\n";
print $$phone->{Home};
```

Please make note that our structure is the same as the ones in the previous examples.

We have so much indirection going on (even though this is a fairly simplistic example) I think it would be appropriate to examine graphically the data structure represented in our code.



First, our reference $phone holds a reference to an anonymous hash. Perl is quite nice (sophisticated) in that it is able to tell us what type of reference we have. Not only does it contain the address, but the type of storage we have at that address. I think it is quite ironic that a language that does not offer typed data as part of the primitive language, would allow us to easily differentiate the actual data type stored in memory.

Looking at what data is actually stored in $phone…

```
$phone
```

```
HASH(0x1a755b0)
```

…we see that we have a reference to a hash. When we dereference $phone…

```
%$phone
```

```
HomeHASH(0x1a75538)WorkHASH(0x1a7f120)
```

…we get the data that is stored at location 0x1a755b0. Since, we already know our reference is to a hash we should expect to see at least one key/value pair. In this instance we return two **key**/**value** pairs.

```
Home  HASH(0x1a75538)

Work  HASH(0x1a7f120)
```

What starts to make this so abstract is that our key value pair is yet another reference. We see that we have a key named 'Home' that is a hash reference and a key named 'Work' that is also a hash reference.

We could access each value with the following syntax.:

```
%$phone->{Work}

%$phone->{Home}
```

```
HASH(0x1a7f120)

HASH(0x1a75538
```

Please make note that we are using the infix operator (->) with the hash dereferencer. If we were to attempt to dereference $phone in the wrong context (in this case using the scalar dereferencer …

```
$$phone->{Work}
```

…we would receive the following runtime error).

```
Not a SCALAR reference at anonrefhsh1c.pl line 26.
```

The key point to remember here is that when you are using the infix operator, you must make sure you are applying the correct context.

The following code snippet uses the infix operator appropriately and is the equivalent of `%$phone->{Work}`

```
$phone->{Work}
```

…returns…

```
HASH(0x1a75538)
```

Hopefully this has served to demystify all the indirection that is a natural byproduct of anonymous hashes. Some of you reading this may feel like the room is spinning. My goal is not to confuse or bewilder you (honest). But, if you ever hope to explore and utilize the myriad of modules out there that are free for the taking then understanding all of this stuff will carry you along way. This is certainly a very compelling and difficult subject for most programmers.

So we do not leave any loose ends regarding this topic I am going to finish it by showing code used to access individual elements in our hashes. (This is review of what has already been covered)

🖫 ANONREFHSH1c2.PL

```
print $phone->{Work}->{extension} . "\n";
print $phone->{Home}->{extension} . "\n";


my $getphone = \$phone  ;
print ${$getphone}->{Work}->{extension};
```

The syntax for the first two print statements has been previously discussed. Notice that we can create a reference to our anonymous hash.

```
my $getphone = \$phone  ;
```

And, we can dereference this scalar to get at the data.

```
print ${$getphone}->{Work}->{extension};
```

Finally, I want to end this topic by trying to put into perspective the practical merits of all of this discussion. The primary and principle goal of references is to allow us to abstract the programs we write. The primary reason we want to abstract programs is to accomplish a degree of reuse. The noble objective of reuse is that it minimizes the volume of code we need to write to accomplish a given task. Also reuse serves to help us save time and rework in the programs we write and deliver.

Each and every one of these examples could have easily been written to accomplish the same end output without using references. Granted, but the point of this discussion is to present in its most basic and stripped down format the mechanics of references so that you understand the syntax and techniques that are deployed in writing reusable code in Perl. Just venture out and take a look at the module code that is available for Perl. You will find a great deal of it is written using references.

In perspective to the phone examples that have been presented, our goal in creating these may very well be to provide an agreed upon data structure to use for dealing with phone numbers. If this is the case then what the programmer reusing the structure needs to be concerned with is how to access the individual elements and how to refer to the structure as a whole. The technical details of how we design the structure really only need to be understood by the architect.

I am writing this subject to address two distinct audiences; programmers that will be *creating* their own modules in Perl, and programmers that will be *using* existing modules. I may certainly be a bit ambitious in trying to address both audiences in a singular document.

If you are planning on writing your own modules, and have done similar types of work in other languages then I do not need to tell you the importance of understanding all of this. If you are in the other camp you may well be questioning why you need to understand all of this. This certainly is a legitimate question. In a perfect world a programmer using a library or module or class (or whatever you want to call it) should not have to be concerned with the technical details of what they are using. But, that is only in a perfect world. More often than not you will be faced with less than stellar documentation and it helps immeasurably if you understand the syntax and mechanics of this subject.

Subroutines

An advantage of an anonymous subroutine is that I force the consumer to access the subroutine via the reference. This creates a standard entry point and gives me a degree of encapsulation.

At this point you should have a good grasp on anonymous references. No new syntax will be presented and I am assuming that you already have a grip on how to pass values in and out of subroutines. Four examples will be presented.

Passing values to an anonymous subroutine

🖫 ANONSUB1A.PL

This example serves to demonstrate how to call an anonymous subroutine. The syntax is especially important to understand if you are planning on using the plethora of modules out in the public domain, as this is usually how you will have to access their code.

```
my $v = sub {
            while (@_)
            {
             print pop(@_) . "\n";     #stack
            }
           };


$v->("how to call an anon sub and pass vals\n");
```

```
my $str = "passing a scalar\n";
$v->($str);

my @arr = qw(1 2 3 4 5);
$v->($str,@arr);

my %hsh = (hanna=>'one', dool=>'two', set=>'three');
$v->(%hsh);
```

This code example demonstrates how to pass both singular and multiple arguments into an anonymous subroutine. The general syntax is…

$<scalar_reference_to_subroutine>**->(**<data_to_pass>**)**

Notice from the code that it is possible to pass multiple arguments into the subroutine.

```
$v->($str,@arr);
```

Passing references to an anonymous subroutine

ANONSUB1B.PL

This example is a slight modification of the previous one (producing the same output) that demonstrates that we can pass references into a subroutine.

```
my $str = "passing a scalar\n";
my $Rstr = \$str;
$v->($$Rstr);

my @arr = qw(1 2 3 4 5);
$Rstr = \@arr;
$v->(@$Rstr);

my %hsh = (hanna=>'one', dool=>'two', set=>'three');
```

```
$Rstr = \%hsh;
$v->(%$Rstr);
```

Returning an array from an anonymous subroutine

🖫 ANONSUB2.PL

This example demonstrates how to return an array from an anonymous subroutine. You can see that this is done in the same manner as a regular old subroutine.

```
my @a = qw(1 2 3 4 5);
my $r = arr_back(@a);

while ($r->(@a))
{
  print @a . "\n";
  shift (@a);
}

sub arr_back
{
  my $asub = sub
           {
            my @arr = @_;
            @arr = reverse(@arr);
            return @arr;
           };
  return $asub;
}
```

Returning a scalar from an anonymous subroutine

🖫 ANONSUB3.PL

```
my $b = $ARGV[0];
chomp $b;
```

```
print "$b\n";

my $a = arr_back($b); #call the sub
print $a->(my $c);    #deref

sub arr_back
{
  my $rev = $_[0];
  my $asub = sub
           {
            my $rev2 = $rev;
            $rev = reverse($rev);
            if ($rev =~ /$rev2/i)
            {
              $rev = $rev . " is a palindrome"
            }
            else
            {
              $rev = "that word is not a palindrome"
            }
           };
  return $asub;
}
```

Misc. Stuff

Here is one more subject I did not know how to classify but still felt it deserved coverage.

Array of Hash References

We have already discussed how to create an array references to other arrays. It is also possible to create an array of hash references. For that matter, you can also have an array with references to any data type (subroutines, scalars, etc…).

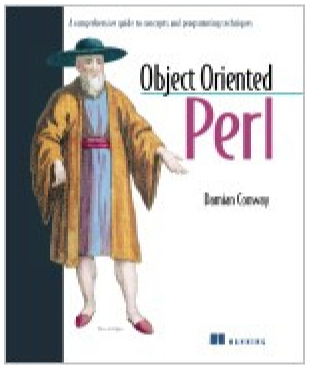
🖫 ARRHSHREFS.PL

```
my %name = (firstname    => "Fred",
          middle_init  => "L",
          lastname     => "Flintstone"
          );

my %address = (street  => "123 Rubble Way",
             city    => "Seattle",
             state   => "WA",
             zip     => "98058"
            );

my %phone = (area_cde  => "254",
           exchange  => "455",
           extension => "1223"
          );

my @employee_record = (\%name, \%address, \%phone);

print %{$employee_record[1]} ; print "\n";
print ${$employee_record[0]}{lastname} ;
```

What next?

Depending on how this is being covered in the course you are taking there are a number of directions you can now travel. You certainly have been given enough introduction to the subject to venture into the world of object oriented programming in Perl. Also, as references are a grand form of abstraction, and since abstraction is the lingua franca of reuse, you could head off into the world of modules. This could be either creating your own modules or using the myriad of modules that are out there in the public domain free for your (re)use.

If you are interested in studying object oriented programming in Perl, I would recommend the book by Damian Conway titled, "Object Oriented Perl".


A comprehensive guide to concepts and programming techniques
Object Oriented
Perl
Damian Conway
MANNING