Reading List

- Online reference documentation – all 500+ pages of it.
- Frequently Asked Questions List
- "Advanced Perl Programming"
  - by Sriram Srinivasan
- "Programming perl"–
  - by Larry Wall, Randal Schwartz, Tom Christiansen
- The Perl Journal (http://www.tpj.com)
- Internet
  - http://www.perl.com/perl/index.html
  - comp.lang.perl.{misc,modules,tk,announce} – USENET newsgroups
  - Perl porters gateway

Sriram Srinivasan (© 1999)

References: An introduction

- A scalar can hold a reference to any piece of data
  - Hence "$s" can be a integer, double or string–valued scalar
  - Or, it can *refer* to another perl data type (scalar, hash, array, or a function)
  - Equivalent to a C pointer
- Arrays and hashes can hold many scalars
  - ... some or all of which can be references
  - Hence "@array" can contain numbers, strings and/or references (to any data types)

Sriram Srinivasan (© 1999)

Creating references

- **In "C", two ways of creating references**
  - **Referring to an existing object**

```
int *p; int x;
p = &x;
```

  - **Creating anonymous objects and referring to them**

```
p = malloc (sizeof(int) * 10); /* Creating 10 integers */
```

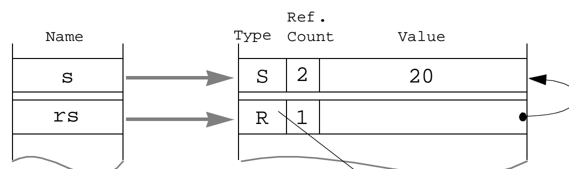
- **In Perl, similar mechanisms available**

References to existing objects

- **Referring to another scalar**

```
$s = 20;
$rs = \$s;  # Making rs point to s
```

- **References always point to values, not to symbols**
  - **The value of a reference variable is internally a pointer to another value.**
- **Values are reference counted to prevent inconsistency**
  - **Each value keeps track of how many other objects are currently referring to it.**
  - **C pointers without the memory management hassles.**



Data Type: Scalar (S), Reference (R), Array(A) ...

Dereferencing

- **Dereferencing: given a reference, getting underlying data**
- **An extra dollar does the trick**

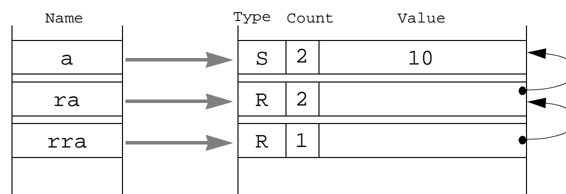
```
$s = 10; $rs = \$s;      # Create the reference
$$rs = 50;               # Dereference and modify the value.
print $s;                # Should  print "50"
```

- **Notes**
  - **"$s" is the value obtained by dereferencing the symbol "s".**
  - **Can replace an identifier <u>name</u> with a scalar variable containing a reference of the correct type**
    - **$$x = $$x + 45; print sin($$x);   # replace "s" by "$x", if $x is a scalar ref.**
  - **Use 'hungarian' notation**

```
$s = 10; $rs = \$s; $rrs = \$rs;
print $$$rrs; # prints 10
```


Visualizing Dereferencing

- **Chase arrows starting from left**
  - **Number of "$" signs == number of arrows chased**

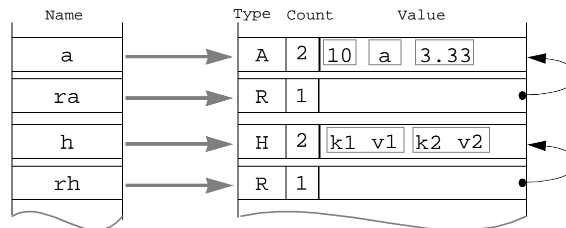


- **Summary**
  - **Taking a reference:** `$ra = \$a;`
  - **Using the reference:** `$b = $$ra + 10;`
  - **The value of a reference is a pointer to another value.**

References to lists and hashes

- **Creating references to arrays and hashes**
  - **No different from creating references to scalars. Put a '\' in front**

```
@a = (10, "a", 3.33);
$ra = \@a;
%h = ("k1", "v1", "k2", "v2");
$rh = \%h;
```



Dereferencing array refs

- **Remember the old rule**
  - **Can replace an identifier <u>name</u> with a scalar variable containing a reference of the correct type**

Operation	With variables	Indirectly through a reference (Given $ra = \@a)
Initialize/copy	@a = (1,2,3);	@$ra = (1,2,3);
Push	push (@a, 1, 2, 3);	push (@$ra, 1, 2, 3);
Print	print @a	print @$ra
Access elements	$a[2]	$$ra[2]
Slices	@a[1,5,6]	@$ra[1,5,6]
Iterate	foreach $e (@a) { .... }	foreach $e (@$ra) { .... }

Dereferencing hash refs

- **Same rule**

Operation	With variables	Indirectly through a reference (Given `$rh = \%h;`)
Initialize/copy	%h = ("a" => "apple", "b" => "boy");	%$rh = ("a" => "apple", "b" => "boy");
Keys	@k = keys %h;	@k = keys %$rh;
Access elements	$h{"a"}	$$rh{"a"}
Slices	@h{"a", "b"}	@$rh{"a", "b"}

Iterator	while (($k,$v) = each %h) {
	....
	}
		while (($k,$v) = each %$rh) {
		....
		}

References to subroutines

- **Put a "\" before the subroutine to obtain a reference.**

```
sub hello{
  print "Hi ", @_, "\n";
}
$r = \&hello; # Note no parentheses
```

- **Dereferencing**

```
&hello("Bob"); # perl-4 style subroutine call. Prints Hi Bob.
&$r("Bob"); # Indirectly through a reference.
```

Reference notes

- **Limited "compile–time" type safety**
  - **Functions whose signatures are known are typechecked.**

```
$ra = \@a;
push ($ra, 1, 2); # Compile-error : Type of arg 1 to push must be
                    #                   array (not scalar deref)
```

- **Use the appropriate prefix while dereferencing**

```
$rs = \$s;
push (@$rs,1, 2);   # Runtime error : "Not an ARRAY reference"
```

- **Perl does not automatically dereference references**

Anonymous objects

- **Why "anonymous" ?**
- **Anonymous scalars**

```
$$rs = 10; # Creates a scalar value(10) and points a reference to it
```

- **Two ways of creating anonymous arrays.**
  - **Use "wrapped" references instead of array variables**

```
@$ra = (10, 20);   # Instead of saying @a = (10, 20);
$$ra[9] = 100;     # Instead of saying $a[9] = 100;
push(@$rComposers, "mozart", "beethoven");
```

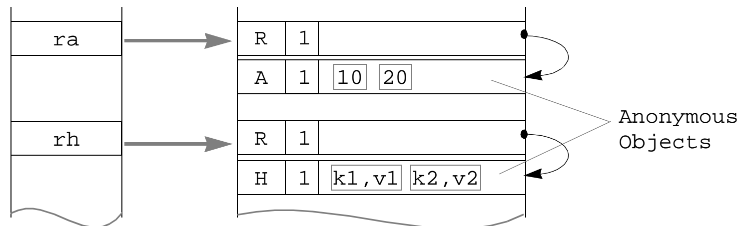
  - **Use the [*list*] construct**

```
$ra = [10,20];   # Creates and returns reference to an anon. array
```

Anonymous objects (contd.)

- **Anonymous hashes are similar**

```
%$rh = ("k1" => "v1", "k2" => "v2");  # => is alias for ","
or,
$rh = {"k1" => "v1", "k2" => "v2"};# Creates and returns
                                    # reference to an anon. hash
```



Anonymous Objects (contd.)

- **Anonymous subroutines**

```
$rs = sub { print "Arg = ", $_[0], "\n"};# Note ";" at end
&$rs(100); # Call the subroutine via the reference.
```

- **Closures**
  - **Anonymous subroutines that remember environment when created**

```
sub getSub {
   my $arg = $_[0];
   my $retval = sub {print "Arg = $arg \n";}; # Note ";" at end
   return $retval;
}
$rs1 = getSub ("hello");
$rs2 = getSub ("world");
&$rs1();  # prints "Arg = hello"
&$rs2();  # prints "Arg = world"
```

Arrow notation

- **For accessing elements of arrays and hashes, use the –> notation optionally:**

  `$ra->[1]` **is the same as** `$$ra[1]`

  `$rh->{k1}` **is the same as** `$$rh[k1]`

- **Calling subroutines using references**

  `$rs->(100)` **is the same as** `&$rs(100)`

Nested data structures

- **Remember that lists and hashes can contain any type of scalar**
- **List of lists**

```
$rl = ["a", 20]; #$rl is a scalar (happens to contain a pointer)
@lol = (1, 3, $rl);  # lol contains three scalars.
```

  - **or more simply**

```
@lol = (1, 3, ["a", 20]);
```

  - **Note, this is very different from**

```
@flatlist = (1, 3, ("a", 20));
```

- `print` **does not print nested structures automatically**

```
print  "@lol" ;# prints "1 3 ARRAY(0xadcc8)"
```

Nested data structures (contd.)

- **Hash of arrays**

```
%tvShows = (  # Note the "(" and "[" usage ...
       "seinfeld"      => ["seinfeld", "kramer", "george", "eileen"],
       "friends"       => ["ross", "chandler", "joey"],
       "home improvement" => ["tim", "jill", "al"],
);
print $tvShows{"seinfeld"}->[1]; # prints "kramer"
print $tvShows{"seinfeld"}[1]; # Eliminating arrow between indices
```

- **Hash of hashes**

```
%hoh = (
        "seinfeld" => {
             "lead"    => "jerry",
             "friend"  => "kramer" },
       "simpsons" => {
            "lead"     => "homer",
            "kid"      => "bart"}
);
```


Exercise (5 minutes)

- **Write a script to print a sorted list of characters for each element in %tvShows.**
- **Given a file,**

```
seinfeld lead jerry pal kramer
simpsons lead homer kid bart
```

write a script to build a hash of hashes.

  - **First word in a line is the name of the show, the rest are key–value pairs.**

- **What happens here ?**

```
@lol = ( [1,2], [3,4]);
@newarr = ("hello", "world");
$lol[1] = @newarr;
```

What does @lol contain now ?

ref()

- **Finding out what a reference variable refers to**
- **Returns FALSE or a string**
  - **FALSE – if it is not a reference variable at all**
  - **"SCALAR", "HASH", "ARRAY" – if it is a reference to a scalar, hash or list**
  - **"REF" – if it points to another reference variable**
  - **"CODE" – if it refers to a subroutine**
  - **"*package*" – depending on the package it belongs to**
- **Example**

```
$a = 10;

$ra = \$a;

print ref($a); $ prints nothing, because $a is not a reference.

print ref($ra); # prints "SCALAR"
```


Example: Pretty–print data structure

- **Usage**

```
@list = (10, {3 => 4, "hello" => [6,7]}, 11.344);
PrettyPrint (@list);
```

- **Output**

```
LIST: [
:  10
:  HASH: {
:  :  3 => 4
:  :  hello => {
:  :  :  LIST: [
:  :  :  :  6
:  :  :  :  7
:  :  :  ]
:  :  }
:  }
:  11.344
]
```

Example: PrettyPrint

```
1: Usage: PrettyPrint (10,{3 => 4, "hello" => [6,7]}, 11.344);
2: $level = -1; # Level of indentation
3:
4: sub PrettyPrint {
5:     PrintList(@_);
6: }
7:
8: sub PrintList {
9:     my ($var);
10:    ++$level; PrintIndented ("LIST: [");
11:    foreach $var (@_) {
12:        if (ref ($var)) {
13:            PrintRef($var);
14:        } else {
15:            PrintScalar($var);
16:        }
17:    }
18:    PrintIndented ("]"); --$level;
19:}
```


Example: PrettyPrint (contd.)

```
20:sub PrintScalar {
21:    ++$level; PrintIndented ($_[0]); --$level;
22:}
23:
24:sub PrintRef {
25:    my $r = shift @_;
26:    $refType = ref($r);
27:    if ($refType eq "ARRAY") {
28:        PrintList(@$r);
29:    } elsif ($refType eq "SCALAR") {
30:        PrintScalar($$r);
31:    } elsif ($refType eq "HASH") {
32:        PrintHash(%$r);
33:    } elsif ($refType eq "REF") {
34:        PrintRef($$r);
35:    } else {
36:        die ("Reference type '$refType' (not supported)");
37:    }
38:}
```

```
39:sub PrintHash {
40:    my($key, $val);
41:    ++$level; PrintIndented ("HASH: {");
42:    while (@_) {
43:        $key = shift; $val = shift; # shift applies to @_ by default
44:        $val = ($val ? $val : '\"\"'); # Use '' for empty values
45:        ++$level;
46:        if (ref ($val)) {
47:            PrintIndented ("$key => {");
48:            PrintRef($val);
49:            PrintIndented ("}");
50:        } else {
51:            PrintIndented ("$key => $val");
52:        }
53:        --$level;
54:    }
55:    PrintIndented ("}");  --$level;
56:}
57:sub PrintIndented {
58:    $spaces = ":  " x $level;
59:    print "${spaces}$_[0]\n";
60:}
```


Symbolic References

- **If dereferencing fails to yield a reference, Perl checks to see if it yields a string. If so, it uses that variable name to dereference it.**

```
$i = 10;
$r = 'i';
print $$r; # $r fails to yield a reference to a scalar, but it
           # does yield a string 'i'. Use that as a variable name,
           # and get its value. This expression prints 10.
```

- **Be wary of symbolic references, because typos tolerated!**
  - `use strict` **disables symbolic referencing.**

Software Engineering with perl

Programming through the ages

- **Procedural Programming**
- **Modular Programming**
  - **Procedural programs inside modules**
- **Data Abstraction**
  - **Data hiding**
  - **User defined types**
  - **Each module built on a user defined type**
- **Object Oriented Programming**
  - **Data abstraction + ...**
  - **Subtyping (inheritance)**
  - **Polymorphism**

Modular Programming

- **"package" construct**
  - **All global names (for variables, functions) belong to a package, by default "main"**
  - **A package declaration ends the previous package and starts a new one.**
  - **Typically, packages are written in separate files, and used with the "require" statement**

File: mod.pl

```
package A;
$str = "A says Hi";
sub PrintStr {
    print $str;
}
package B;
$str = "B says Hi";
sub PrintStr {
    print $str;
}
```

Usage

```
require 'mod.pl';
A::PrintStr();
B::PrintStr();
$A::str = "hello";
$B::str = "bye";
```

- **No privacy for global variables.**
- **Nested names (Math::Calculus::integrate())**

Run-time binding

- **The :: operator checks call at compile–time**
- **Use "–>" for run–time binding**

```
A->PrintStr();
# or even,
$module = <STDIN>;
$module->PrintStr(); # Module name is known only at run-time
```

- **Called subroutine gets the package name as the first argument**
  - **All arguments to the subroutine are shifted one to the right**

```
package Message;
sub say {
  print join (" ", @_), "\n";
}

Message->say("Howdy"); # Prints "Message Howdy"
```

Package Initialization

- **Package level initialization, and destruction**
  - `BEGIN{}` **– All** `BEGIN{}` **blocks executed when module first loaded**
  - `END{}` **– All** `END{}` **blocks executed when interpreter is about to exit.**
- **"use" – convenience function**

  `use mod;` **is equivalent to saying**

  `BEGIN { require 'mod.pm';}` **in your file.**
  - **Hence filenames have to end with a '.pm' suffix to be automatically picked up**
  - **@INC has the include path**

Example: Using standard modules

- **File::Find**

```
1: use File::Find;  # Exports the name "find" into the current namespace
2: find (\&RemoveUnwantedFiles, "C:/sriram");
3: $dirName = ""; # Keeps track of current directory.
4: sub RemoveUnwantedFiles {
5:     if ($dirName ne $File::Find::dir) {
6:         $dirName = $File::Find::dir;
7:         print $dirName, "\n";
8:     }
9:     if (($File::Find::name =~ /\.old$|\.bak$/) &&
10:         (! -d $File::Find::name)) {
11:         unlink $File::Find::name;  # Removes file
12:     }
13:}
```

  - **find2perl – utiltity that converts "find" command to perl code using the above module**

Data Abstraction and Encapsulation

- **Hide data structures**
  - **You never question what's inside an O/S or a database**
  - **Your libraries should be usable in a similarly transparent way**
- **Package subroutines provide gateway to data**
  - **Subroutines represent the package's interface**
- **"Employee" package**

```
use Employee;
$e = Employee::new ("John", 80000);
Employee::give_raise($e, 20000);
print Employee::after_tax_income($e); # prints 70000
```

  - **User doesn't know the data structure used to store an employee record**
  - **Tomorrow, if Employee.pm uses a database, user code is not affected**
  - **Only public functions can be used to update an employee's data.**
  - **Employee data is completely "encapsulated" within the package**

Example: Employee.pm

```
1: package Employee;
2: sub new {
3:     my ($name, $salary) = @_;
4:     my %emp;
5:     $emp {"name"}    = $name;
6:     $emp {"salary"}  = $salary;
7:     return \%emp;  # returns reference to a local variable.
8: }
9: sub give_raise {
10:    my ($rEmp, $raiseAmount) = @_;
11:    $rEmp->{"salary"} += $raiseAmount;
12:}
13:sub after_tax_income {
14:    my ($rEmp) = @_;
15:    return $rEmp->{"salary"} * 0.70 ; # 30% tax bracket.
16:}
```

Extending Employee.pm

- **Say we have hourly and regular employees**
  - **Methods such as give_raise and after_tax_income have to be modified to look at the types of employee.**
  - **Different code, so different packages for the two types.**

```
$emp1 = HourlyEmployee::new ("John", 35) ; # Hourly rate = $35.
$emp2 = RegularEmployee::new("Alice", 80000);  # Annual salary =
80000
```

  - **Problem: Have to keep specifying the exact package for every package**

```
print HourlyEmployee::after_tax_income ($emp1);
print RegularEmployee::after_tax_income($emp2);
```

- **Run–time binding to the rescue**

Example: Employees as objects

- **new() returns an object instead of hashref**

```
1: use HourlyEmployee;
2: use RegularEmployee;
3: $e1 = HourlyEmployee->new("John", 80000);
4: $e2 = RegularEmployee->new("Alice", 100000);
5: $e1->give_raise(5000);
6: print $e2->after_tax_income(); # prints 70000
7: foreach $e (get_all_employees()) {
8:   $e->give_raise($e->salary() * 0.1); # 10% raise to everyone.
9: }
```

- **RegularEmployee::new still needs to be explicit**
- **$e2 "bound" to package RegularEmployee**
  - `$e2->give_raise(10000)` **is now automatically equivalent to** `RegularEmployee::give_raise($e, 10000);`

Example : Using run–time binding

```
1: package RegularEmployee;
2: sub new {
3:     my ($name, $salary) = @_;
4:     my %emp;
5:     $emp {"name"}    = $name;
6:     $emp {"salary"}  = $salary;
7:     return bless(\%emp);  # returns blessed reference to a local var.
8: }
```

- **"bless" tags an ordinary reference with the name of a package**
- **No changes to other subroutines.**
- **Commonly used style:**

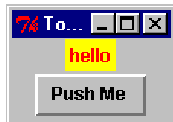
```
1: sub new {
2:     my ($name, $salary) = @_;
3:     # initialize, bless and return anon hashref in one fell swoop.
4:     bless {"name" => $name, "salary"  => $salary};
5: }
```


Object Oriented Programming

- **"Methods " – Fancy name for functions provided by a package**
  - **Constructor – new()**
  - **Static methods – functions at the package (class) level**
    - **find_employee()**
  - **Instance methods – functions that operate on a single object,**
    - **give_raise, after_tax_income**
- **Polymorphism**
  - **Syntax and facility to allow an object to be identified with its class**
- **Inheritance**
  - **Employee**
  - **HourlyEmployee and RegularEmployee as subtypes of Employee**

User Interfaces with Perl/Tk

```
1: use Tk;
2: $mw = MainWindow->new(); # Note: 'MainWindow' automatically imported
3: $label = $mw->Label ("text"=>"hello","fg"=>"red", "bg"=>"yellow");
4: $button = $mw->Button("text"    => "Push Me",
5:                       "command" => \&button_pushed);
6: $label->pack();
7: $button->pack();
8: MainLoop();
9:
10:sub button_pushed {
11:    $label->configure ("text" => "Ouch!!");
12:}
```



Database access

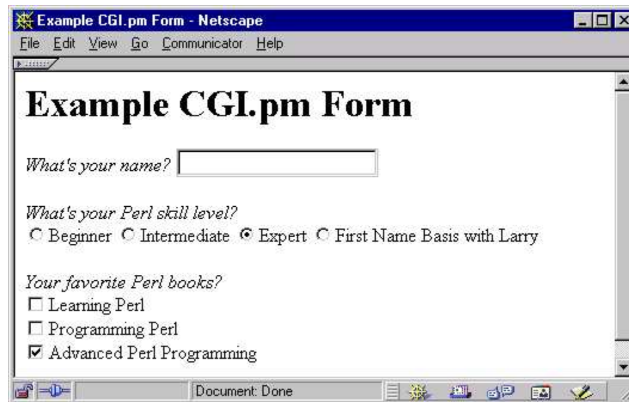
- **DBI – Database independent interface (like ODBC, JDBC)**

```
1: use DBI;
2: $dbname = 'empdb'; $user = 'scott'; $password = 'tiger';
3: $conn = DBI->connect ($dbname, $user, $password,
4:                      'Oracle'); # returns an Oracle 'connection'
5: # Execute sql queries
6: $conn->do ("delete from emptable where status != 'active'");
7: $conn->do ("insert into emptable (name, age) values ('john', 23)");
8: # Or, use prepare() to reuse a template statement over and over again.
9: $stmt = $conn->prepare (
10:       'insert into emptable (name, age) values (? , ?)');
11:$stmt->execute('john',  23);
12:$stmt->execute('alice', 32);
13:# Fetching data
14:$stmt = $conn->do("select name, age from emptable");
15:while (($name,$age) = $stmt->fetch_row()) { ... }
```

- **Always check `$DBI::err` or `$DBI::errstr`**

CGI

- **CGI — Common Gateway Interface**




```
1: use CGI ;
2: $page = CGI->new() ;
3:
4: print $page->start_html("Example CGI.pm Form");
5: print "<H1> Example CGI.pm Form</H1>\n";
```


Example: CGI (contd)

```
6: print
7:      $page->startform(),
8:      $page->em("What's your name?  "), $page->textfield('name'),
9:      "<P><EM>What's your Perl skill level? </EM><BR>",
10:     $page->radio_group(
11:         '-name'   => 'skill',
12:         '-values' => ['Beginner','Intermediate','Expert',
13:                       'First Name Basis with Larry'],
14:         '-default' =>'Expert'),
15:     $page->br(),
16:     "<P><EM>Your favorite Perl books? </EM><BR>",
17:     $page->checkbox_group(
18:         '-name'   => 'Favorite Books', '-linebreak' =>'yes',
19:         '-values' => ['Learning Perl', 'Programming Perl',
20:                       'Advanced Perl Programming'],
21:         '-default' =>'Advanced Perl Programming'),
22:     $page->endform(),$page->hr(),
23:     $page->end_html(); # end of print statement
```

Special methods

- **sub DESTROY()**
  - **Called when object is being finally destroyed.**

```
package Employee;
# ... other Employee methods
sub DESTROY {  # DESTROY is a keyword
    my ($obj) = @_;
    print ($obj->{name}, " has passed away \n");
}

# and in the main program ...
$emp = Employee->new("John", 10000);
undef $emp ; # perl automatically calls $emp->Destroy()
```

- **AUTOLOAD - called if subroutine not found**

```
1: # Calling a non-existing procedure
2: Employee->Foobar();
3: package Employee;
4: sub AUTOLOAD {
5:     # $AUTOLOAD is set to "Employee::Foobar"
6:     print "$AUTOLOAD not found in this package\n";
7: }
```


Inheritance

- **A package (class) can inherit methods from other packages**
  - **Package sets up a (package specific) @ISA list**

```
package HourlyEmployee;
@ISA = ("Employee");
```

  - **If a method is not found in the object's package, perl looks for it in each of the packages mentioned in the @ISA list**
  - **The method look up is depth–first (Employee may have its own @ISA)**
- **There is no attribute inheritance**
  - **Need to follow your own conventions about structuring your data**
  - **Look at the "perlbot"(bag o' tricks) manpage for possible approaches.**
- **Don't use inheritance because it looks cool!**
  - **Use delegation/composition where possible**

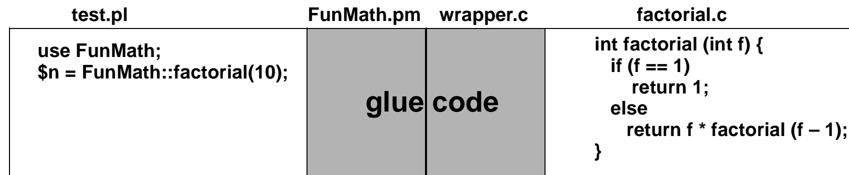
perl and C/C++

Embedding vs. Extending perl

- **Embedding perl**
  - **Your C/C++ program makes calls to perl**
  - **Similar to Emacs and elisp, Autocad and autolisp, Microsoft Word and VBA**
  - **Caller: C program,   Callee : perl script**
- **Extending perl**
  - **perl script calls your "C" functions**
  - **Adding database support, communications support etc.**
  - **Caller: perl , Callee: C code**
- **Doing both**
  - **User Interface toolkits, such as TkPerl**
  - **Needs extensions, so developer can write perl code to draw windows**
  - **Needs embeddability, so UI code can call event–handler code written in perl**

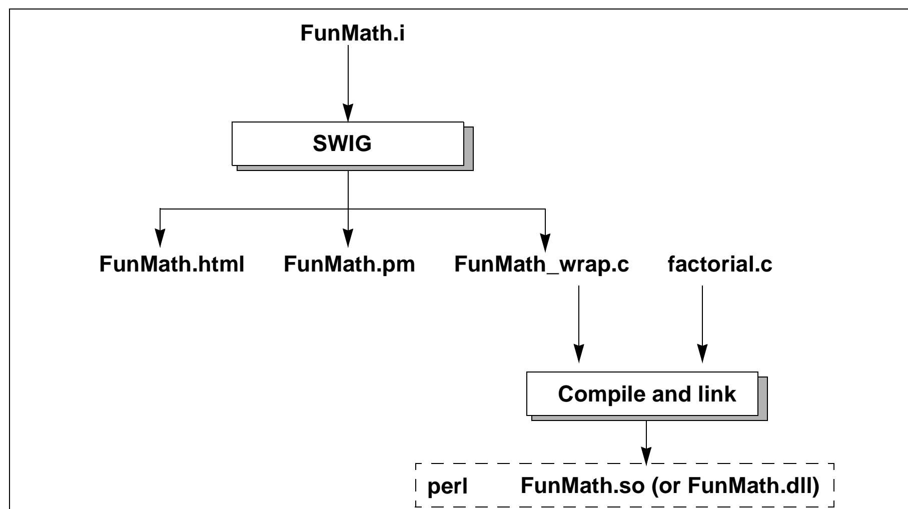
Extending Perl

- **Extension: Supply "glue" code between Perl script and custom C code**



- **Two tools for creating extensions**
  - h2xs, xsubpp pair – comes standard with Perl
  - SWIG (Simplified Wrapper and Interface Generator)
    - http://www.cs.utah.edu/~beazley
- **Input for these tools:**
  - Interface file
  - Typemaps

SWIG Process



- **Write an interface file for your library**

```
%module FunMath
int factorial (int n);
```

SWIG Process (contd)

- **Compile with SWIG**

```
swig -perl5 FunMath.i
```

- **Compile C code and create shared library**

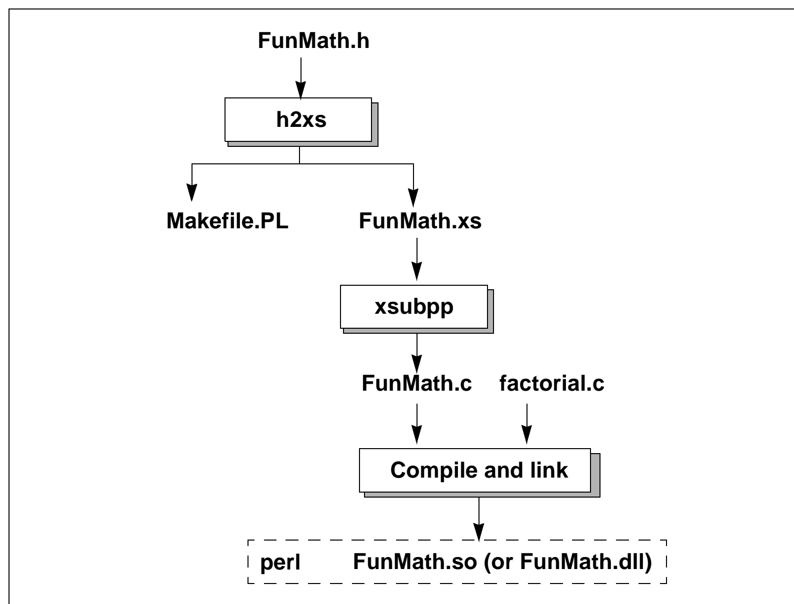
```
cc -c -Kpic FunMath_wrap.c factorial.c # on solaris
cc -G -o FunMath.so FunMath_wrap.o factorial.o
```

- **Use the package**

```
use FunMath; # Looks for FunMath.pm,
             # which dynamically loads FunMath.so

$fact = FunMath::factorial(10);
```


XS Process



XS Process (contd)

- **Pass header file through h2xs**

```
h2xs -x FunMath.h
```

- **Edit Makefile.PL**

```
use ExtUtils::MakeMaker;
WriteMakefile (
   'Name'   => 'FunMath',
   'OBJECT' => 'FunMath.o factorial.o'
);
```

- **This can be used for SWIG also**
- **Compile and Install**

```
perl Makefile.PL
make
make install
```


Comparison of perl with other languages

- **perl**
  - **Strengths**
    - **Lots of built–in functionality**
    - **Easy to port (and has been ported) to multiple platforms**
    - **Text processing**
    - **True interpreter – dynamic evaluation of code possible**
    - **Extensive libraries and tools**
    - **Extensively integrated with commercial libraries (databases, ui toolkits etc.)**
    - **Untyped scalars really useful for text manipulation – numbers and strings are automatically interchangeable.**
  - **Weaknesses**
    - **Construction of complex data structures or objects mistake prone**
    - **Heavy reliance on all kinds of symbols $, %, @, &**
    - **OOness feels grafted on**
- **Java**
  - **Pros**
    - **Good systems programming language. (Down with C++ !)**
    - **Compile–time type checking**

Comparison of perl with other languages

- Java strengths (contd.)
  - Portability
  - Lots of dynamic features – reflection, run–time class loading
  - Security layer
  - Multi–threaded, and true garbage collection
  - Industry weight behind it
- Java Weaknesses
  - Strict data typing gets in the way of prototyping
  - Requires complex environment to work in
  - Not a scripting language

- **Python**
  - Strengths
    - Excellent minimal, lightweight OO interpreted language – my favorite !
    - Good and extensive set of class libraries
    - Easy to understand
    - Easy integration with C/C++
    - Good text processing features

Comparison of perl with other languages

- Python Weaknesses:
  - Much lesser number of people involved

- **Tcl (Tool Command Language)**
  - Strengths
    - Extremely easy to understand and integrate
    - Good "glue" language between applications
    - Tk
  - Weaknesses
    - Can’t handle complex data structures well at all.
    - Much slower than perl for comparable system admin tasks
    - Not good for large scale software engineering – code can get quite horrendous on a larger scale.

Exercise (solution)

- **Printing %tvShows**

```
1: $, = ' ';
2: while (($k, $v) = each %tvShows) {
3:    print "$k :", sort (@$v), "\n";
4: }
```

- **Hash of hashes**

```
1: open (F, 'x.dat') || die "Could not open file\n";
2: while ($l = <F>) {
3:         chomp($l);s
4:         @list = split (/\s+/, $l);
5:         $, = ' ';
6:         $show = shift @list;
7:         $rh = {};  #This step is optional ...
8:         %$rh = @list;   # refs automatically spring into existence
9:         $shows{$show} = $rh;
10:}
```

- **@lol contains ([1,2], 2)**