Overview

- Last Class!

- Wrapping up
  - Some more perl
  - Useful general tools
  - Useful tricks
  - Tips and advice
Announcements

- Please fill out the evaluations on courseworks.....

- Please ask if you are stuck...make sure you do all the assignments....
Setup perl server...


- Also lets take a quick look at cpan
Perl list stuff

- Mentioned:
  - reverse
  - sort

- pop / push
- shift / unshift

- Allow you to quickly manipulate a list of values from either side
Perl continued....

- Perl has a ton of built in stuff

- `@somelist = qw/item1 item2 item3 item4/;`

- This is a quote function which takes a bunch of items and returns the list of them

- `@list = ("item1","item2","item3","item4");`
  - Easier to spot problems...
Perl Taint mode

- `-T`
  - Taints all data references (incoming)

- `#!/usr/bin/perl -wT`

- Flags data to make sure perl doesn’t do anything insecure
Tainted?

- STDIN
- CGI

If variables/values are tainted, limits on what you can do with it...

Tainted follows it around with assignments, here is trick to check if variable is tainted...

```perl
sub is_tainted {
    my $var = shift;
    my $blank = substr($var, 0, 0);
    return not eval { eval "1 || $blank" || 1;
}
```
Why

- Why would you want to keep track of tainted data?
Getting out of taint

- Match related patterns ($1,$2 ..)
- Idea: would check for security problems and then allow it

- Reminder: only in taint mode if set
Perl modules

- Modules are reusable code which you put into a .pm file

- In your code
  
  use something

  makes perl look for something.pm

  use some::else

  makes perl look for some/else.pm
Code examples

- Unix organizes the /etc/passwd file as follows:

  pcap:x:77:77:ARPWATCH User:/var/arpwatch:/sbin/nologin
  ident:x:98:98:pident user:/:/bin/false
  nobody:x:99:99:Nobody:/:/sbin/nologin
  xfs:x:405:405:X Font Server:/etc/X11/fs:/bin/false
  mysql:x:6730:1101:mysql server:/var/lib/mysql:/bin/bash
So how to parse it for finding things…

Example:
- Write a program to grab every file from 10,000 machines and look for specific user/type of account..
- Looking for breakins….
- Maybe terminating an employee
- Auditing the system (see who has what permission)
sub read_passwd {
    my %users;
    my @fields = qw/name pword uid gid fullname home shell/;

    while(<STDIN>) {
        chomp;
        my %rec;
        @rec ={@fields} = split(/:/);
        $users{$rec{name}} = \%rec;
    }
    return \%users;
}
my $users = read_passwd();

my @names;

foreach (keys %{$users}) {
    next unless $users->{$_}{fullname};

    my ($fname, $lname) = split (/\s+/, $users->{$_}{fullname}, 2);

    push @names, "$fname $lname";
}

print map { "$_\n" } sort @names;
Random helpful stuff

- $| = 1$
  will turn off output buffering great when working with cgi (later today)

- In perl, can call external commands i.e. we can execute command line arguments
  1. Backticks (``)
  2. System
  3. exec
Cgi perl

- Lets do a few simple examples in perl

- Much easier to do parsing....
Perl + cgi

- Remember:
  - Perl is only a tool here
- STDIN
  - Contents passed to perl script
- STDOUT
  - Will need HTTP headers before printing
- STDERR
  - Depends on server, sometimes just error logs, sometimes error reports on client
- Don’t forget to setup the webserver…..abyss in this case…
%ENV

- This is your best friend in PERL CGI
- Way of getting information from the client
- Create content is way to pass back information to the client
Remember

- Need to set permissions:
  - `chmod 0755 ??? .cgi`
  - `-rwxr-xr-x`
- Need to place script in correct place
  - Usually `cgi-bin/` directory
- Naming
  - Usually need to end in `.pl` or `.cgi` (depending on server)
Problem

How can we print out all the environment variables?
Example

#!/usr/local/bin/perl

use strict;

my $vars
print "Content-type: text/html\n\n";

foreach $vars (sort keys %ENV){
    print "<P><B>$vars</B><BR>";
    print $ENV{$vars};
}
Practical shell programming...
Unix Command Shell

- What is UNIX exactly?
- What are Unix flavors?
- What in the world is a command shell?
Brief History

- Early on, OS were specialized to hardware
  - Upgrade = new OS
- 1965, Bell Labs and GE
  - Multics
    - System to support many users at the same time
    - Mainframe timesharing system
- 1969 – Bell withdrew, but some researchers persisted on the idea of small operating system
More history

- So first ideas coded in Assembler and B
- Rewritten in C – wanted high level code
  - First concept of software pipes
  - Released in 1972
  - Released source through licensing agreements
  - Addition of TCP and specialization versions to different groups
  - Taught in university courses where it caught on
  - Brought to business by new graduates 😊 (early 80’s)
  - System V (1983)
Command shell

- Allows you to interact with the operating system
- Usually refer to non graphical one

- Windows NT/XP:
  - Start -> run -> cmd

- Windows 98
  - Start -> run -> command

- Unix
  - Log in (most of the time)

- Mac
  - terminal
Technical Details

- Shell is simply a program which takes your commands and interprets them
- Usually write your own in OS course
- Many different kinds of shells
  - Mainly to confuse you 😊
- Main advantage
  - Can use build in language to write simple but powerful scripts
Main shells (unix)

- Bourne Shell
  - sh
  - ksh
  - zsh
- C shell
  - csh
  - tcsh
shell

- sh is the “Bourne shell”, the first scripting language
- it is a program that interprets your command lines and runs other programs
- it can invoke Unix commands and also has its own set of commands

while ( 1 ) {
print prompt and wait for user to enter input;
read input from terminal;
parse into words;
substitute variables;
execute commands (execv or builtin);
}

shell commands can be read:
- from a terminal == interactive
- from a file == shell script

search path
- the place where the shell looks for the commands it runs
- should include standard directories:
  - /bin
  - /usr/bin
  - it should also include your current working directory (.)
are you running the Bourne shell?

type:

$SHELL

if the answer is /bin/sh, then you are

if the answer is /bin/bash, then that’s close enough

otherwise, you can start the Bourne shell by typing sh at the UNIX prompt

enter Ctrl-D or exit to exit the Bourne shell and go back to whatever shell you were running before...
Power of Shells

- capable of both synchronous and asynchronous execution
  - synchronous: wait for completion
  - asynchronous: in parallel with shell (runs in the background)

- allows control of stdin, stdout, stderr

- enables environment setting for processes (using inheritance between processes)

- sets default directory
Useful tools & commands

- **wc** – counts characters, words and lines in input
- **grep** – matches regular expression patterns in input
- **cut** – extracts portions of each line from input
- **cat** – print files
- **sort** – sorts lines of input
- **sed** – stream edits input
- **ps** – displays process list of running processes
- **who** – displays anyone logged in on the system
WC

- unix command: counts the number of characters/words/lines in its input
- input can be a file or a piped command (see below)

Example:
- filename = “hello.dat”
  hello
  world

Usage:
unix-prompt$ wc hello.dat
2 2 12 hello.dat
unix-prompt$ wc -l hello.dat
2 hello.dat
unix-prompt$ wc -c hello.dat
12 hello.dat
unix-prompt$ wc -w hello.dat
2 hello.dat
Global Regular Expression Parser
GREP

- one of the most useful tools in unix

- three standard versions:
  - plain old grep
  - extended grep: egrep
  - fast grep: fgrep

- used to search through files for ... regular expressions!

- prints only lines that match given pattern

- a kind of filter

- BUT it’s line oriented
input can be one or more files or can be piped into grep

examples:
grep "^[aeiou]" myfile
ls -l | grep t

useful options:
- -i ignore case
- -w match pattern as a word
- -l return only the filename if there’s a match
- -v reverse the normal action (i.e., return what doesn’t match)
examples:
grep -i "^[aeiou]" myfile
grep -v "^[aeiou]" myfile
grep -iv "^[aeiou]" myfile

Example:
- how do you list all lines containing any digit?
- how do you list all lines containing a 5?
cut

- unix command: extracts portions of each line from input

- input can be a file or a piped command
- Can cut file according to delimiters (fields) and characters

- syntax: cut <-c|f> <-d>
- note that c and +f+ start with 1; default delimiter is TAB
cat

- Concatenate files and print to standard out

- Easy way to pipe the contents of a file to another command
sort

- unix command: sorts lines of input
- input can be a file or a piped command (see below)
- three modes: sort, check (sort -c), merge (sort -m)
- syntax: sort <-t> <-n> <-r> <-o> POS1 -POS2+
- note that POS starts with 0; default delimiter is whitespace
sed

- stream editor
- does not change the file it “edits”
- commands are implicitly global
- input can be a file or can be piped into sed

example: substitute all A for B:
- sed ’s/A/B/’ myfile
- cat myfile | sed ’s/A/B/’

use the -e option to specify more than one command at a time:
- sed -e ’s/A/B/’ -e ’s/C/D/’ myfile

pipe output to a file in order to save it:
- sed -e ’s/A/B/’ -e ’s/C/D/’ myfile >mynewfile
sed

- sed can specify an address of the line(s) to affect
- if no address is specified, then all lines are affected
- if there is one address, then any line matching the address is affected
- if there are two (comma separated) addresses, then all lines between the two addresses
- are affected
- if an exclamation mark (!) follows the address, then all lines that DON’T match the
- address are affected
- addresses are used in conjunction with commands

- examples (using the delete (d) command):
  sed '$d' myfile
  sed '/^$/d' myfile
  sed '1,/under/d' myfile
  sed '/over/,/under/d' myfile
Regular expression like grep
Except forward slash
delimiter is slash (/)
backslash (escape) it if it appears in the command, e.g.:
```
$ sed 's/\usr/\bin/\usr/etc/\myfile'
```
- meta-character ampersand (&) represents the extent of the pattern matched
- example:
sed 's/[0-9]/#&/ˈmyfile
- what does this do?

- you can also save portions of the matched pattern:
sed 's/\([0-9]\)/#/1/ˈmyfile
sed 's/\([0-9]\)\([0-9]\)/#/1-\2/ˈmyfile
Shell programming

creating your own shell scripts

- naming:
  - DON’T ever name your script (or any executable file) “test”
  - since that’s a sh command

- executing
  - the notation #! inside your file tells UNIX which shell should execute the commands in your file

- example— create a file called “myscript.sh”
  
  #!/bin/sh
  
  echo hello world

- make the script executable: unix-prompt# chmod +x myscript.sh
- execute the script:
  
  ./myscript.sh

myscript.sh
- **quote (')**
  
  ’something’: preserve everything literally and don’t evaluate anything that is inside the quotes

- **double quote (")**

  "something2": preserve most things literally, but also allow $ variable expansion (but not ’ evaluation)

- **backquote (‘)**

  ‘something3‘: try to execute something as a command
Filename is t.sh

```
#!/bin/sh

hello="hi"

echo 0=$hello

echo 1='\$hello'

echo 2="\$hello"

echo 3='\$hello'

echo 4="'\$hello'"

echo 5="'\$hello'"
```

```
filename=hi

#!/bin/sh

echo "how did you get in here?"
```

output=

unix$ t.sh

0=hi

1="$hello"

2=hi

3=how did you get in here?

4=how did you get in here?

5='hi'
comments

- single line comments only (no multi-line comments)

- line begins with # character
Simple commands

- sequence of words
- first word defines command
- can be combined with &&, ||, ;
  - to execute commands sequentially: `cmd1; cmd2;`
  - to execute a command in the background: `cmd1&`
  - to execute two commands asynchronously: `cmd1& cmd2&`
  - to execute cmd2 if cmd1 has zero exit status: `cmd1 && cmd2`
  - to execute cmd2 only if cmd1 has non-zero exit status: `cmd1 || cmd2`
- set exit status using exit command (e.g., exit 0 or exit 1)
pipes

- sequence of commands
- connected with |

- each command reads previous command’s output and takes it as input

- example:
echo "hello world" | wc -w
2
variables

- variables are placeholders for values
- shell does variable substitution
- $\text{var}$ or ${\text{var}}$ is the value of the variable
- assignment:
  - var=value (with no spaces before or after!)
  - let "var = value"
  - export var=value
- BUT values go away when shell is done executing
- uninitialized variables have no value
- variables are untyped, interpreted based on context
- standard shell variables:
  - ${N}$ = shell Nth parameter
  - $$ = process ID
  - $? = exit status
filename=\u0027u.sh\u0027

#!/bin/sh
echo 0=$0
echo 1=$1
echo 2=$2
echo 3=$$
echo 4=?

- output

unix$ u.sh
0=./u.sh
1=
2=
3=21093
4=0

unix$ u.sh abc 23
0=./u.sh
1=abc
2=23
3=21094
4=0
shell variables are generally not visible to programs
environment variables are a list of name/value pairs passed to sub-processes
all environment variables are also shell variables, but not vice versa

show with env or echo $var

standard environment variables include:
- HOME = home directory
- PATH = list of directories to search
- TERM = type of terminal (vt100, ...)
- TZ = timezone (e.g., US/Eastern)
Loops

- similar to C/Java constructs, but with commands
- until test-commands; do consequent-commands; done
- while test-commands; do consequent-commands; done
- for name [in words ...]; do commands; done

- also on separate lines
- break and continue control loop
- **while**
  
i=0
  
while [ $i -lt 10 ]; do
  echo "i=$i"
  ((i=$i+1)) # same as let "i=$i+1"
  done

- **for**
  
for counter in `ls *.c`; do
  echo $counter
  done
if

if test-commands; then
    consequent-commands;
[elif more-test-commands; then
    more-consequents;]
[else alternate-consequents;]
fi

- colon (:) is a null command

- example

```bash
#!/bin/sh
if expr $TERM = "xterm"; then
    echo "hello xterm";
else
    echo "something else";
fi
```
case test-var in
  value1) consequent-commands;;
  value2) consequent-commands;;
  *) default-commands;
esac

- pattern matching:
  - ?) matches a string with exactly one character
  - ?*) matches a string with one or more characters
  - [yY][yY][eE][sS]) matches y, Y, yes, YES, yES...
  - /*[0-9]) matches filename with wildcards like /xxx/yyy/zzz3
  - notice two semi-colons at the end of each clause
  - stops after first match with a value
  - you don’t need double quotes to match string values!
example

#!/bin/sh
case "TERM" in
  xterm) echo "hello xterm";;
  vt100) echo "hello vt100";;
  *) echo "something else";;
esac
biggest difference from traditional programming languages

shell substitutes and executes

order:
- brace expansion
- tilde expansion
- parameter and variable expansion
- command substitution
- arithmetic expansion
- word splitting
- filename expansion
Command subing

- replace $(command) or 'command' by stdout of executing command
- can be used to execute content of variables:

```bash
unix$ x=ls
unix$ $x
myfile.c
a.out
unix$ echo $x
ls
unix$ echo 'ls'
myfile.c
a.out
unix$ echo 'x'
sh: x: command not found
unix$ echo '$x'
myfile.c
a.out
unix$ echo $(ls)
myfile.c
unix$ echo $(x)
sh: x: command not found
unix$ echo $(x)
sh: x: command not found
unix$ echo $(x)
myfile.c
a.out
```
Filename expansion

- any word containing *?([ is considered a pattern
- * matches any string
- ? matches any single character
- [...] matches any of the enclosed characters

```
unix$ ls
myfile.c
a.out
a.b
unix$ ls a*
a.out
a.b
unix$ ls a?
a.out
a.b
unix$ ls a.*
a.out
a.b
unix$ ls a.? a.
a.b
unix$ ls a.???
a.out
unix$ ls [am].b
a.b
```
redirection

- stdin, stdout and stderr may be redirected
- < redirects stdin (0) to come from a file
- > redirects stdout (1) to go to file
- >> appends stdout to the end of a file
- &> redirects stderr (2)
- &> redirects stdout and stderr, e.g.: 2>&1 sends stderr to the same place that stdout is going
- << gets input from a here document, i.e., the input is what you type, rather than reading from a file
Built in commands

- alias, unalias — create or remove a pseudonym or shorthand for a command or series of commands
- jobs, fg, bg, stop, notify — control process execution
- command — execute a simple command
- cd, chdir, pushd, popd, dirs — change working directory
- echo — display a line of text
- history, fc — process command history list
- set, unset, setenv, unsetenv, export — shell built-in functions to determine the characteristics for environmental variables of the current shell and its descendents

- getopts — parse utility options
- hash, rehash, unhash, hashstat — evaluate the internal hash table of the contents of directories
- kill — send a signal to a process
- `pwd` — print name of current/working directory
- `shift` — shell built-in function to traverse either a shell’s argument list or a list of field-separated words
- `readonly` — shell built-in function to protect the value of the given variable from reassignment
- `source` — execute a file as a shell script
- `suspend` — shell built-in function to halt the current shell
- `test` — check file types and compare values
- `times` — shell built-in function to report time usages of the current shell
- `trap`, `onintr` — shell built-in functions to respond to (hardware) signals
- `type` — write a description of command type
- `typeset`, `whence` — shell built-in functions to set/get attributes and values for shell variables and functions
More programs you might like

- **cal**
  - Prints a calendar

bash-2.05$ cal 2 2004

February 2004
Su Mo Tu We Th Fr Sa
1  2  3  4  5  6  7
 8  9 10 11 12 13 14
15 16 17 18 19 20 21
22 23 24 25 26 27 28
29
Usage stuff

■ df
bash-2.05$ df -h

<table>
<thead>
<tr>
<th>Filesystem</th>
<th>Size</th>
<th>Used</th>
<th>Avail</th>
<th>Use%</th>
<th>Mounted on</th>
</tr>
</thead>
<tbody>
<tr>
<td>/dev/hda3</td>
<td>197M</td>
<td>157M</td>
<td>31M</td>
<td>84%</td>
<td>/</td>
</tr>
<tr>
<td>/dev/hda7</td>
<td>296M</td>
<td>65k</td>
<td>280M</td>
<td>1%</td>
<td>/tmp</td>
</tr>
<tr>
<td>/dev/hda5</td>
<td>2.4G</td>
<td>2.0G</td>
<td>385M</td>
<td>84%</td>
<td>/usr</td>
</tr>
</tbody>
</table>

■ du
bash-2.05$ du -ch code2

48k   code2/ai1
56k   code2
56k   total

■ quota
How to do research?

- **Practical research**
  - Know a programming language
  - Have an inquisitive mind
  - Keep an open mind to new ideas
  - Try to solve an open research problem 😊

- **Theory research**
  - Learn some math
  - Learn some theory
  - Relearn the math
  - Solve something 😊
Where to start?

1. Need an idea
2. See if anyone’s done it or tried it in your way
   1. Citeseer (citeseer.ist.psu.edu)
   2. Google
   3. Appropriate Faculty/Researcher
   4. Google groups
Sketch out the idea on small scale

- Design a small experiment which can validate your idea
- Data, data, data, and Data
  - Make or break you
  - Will help your research
    - Make sure it isn’t a circular relationship
- Evaluate results
  - Don’t fake them
  - Even bad results are results
  - Can learn of what not to do
- Write up results
Write up options

- Word vs Latex
- gnuplot
- cvs
- Element of Style
In the real world

1. Keep it simple
   1. Don’t re-invent the wheel
   2. Design first
   3. Even with fancy blinking lights, a bad idea is still a bad idea (but with bad taste)

2. Incremental testing
   1. Recognize when the bug is your fault
   2. See if others have faced it too
   3. Make sure version 1 works on most popular browsers
Bottom line

- We’ve covered a lot this semester
  - Some of it was fun
  - Some of it was hard work (ok most)
  - Some of it was frustrating.

- BUT
  - You have lots of tools
  - Have an idea of where to start when dealing with programming projects
Important lessons for learning new languages

- CS is not meant to be a trade school
- Language isn't important…things change
- Ideas and design are more important

Lessons:
- Choose correct environment
- Choose correct tools
- Make sure to test out ideas…might be someone else’s fault (program think)
- Enjoy what you are doing
Important

- To get the most out of a language find comfortable programming environment
  - Emacs – color files
  - Eclipse
  - Others, see
    - [www.freebyte.com/programming/cpp/](http://www.freebyte.com/programming/cpp/)
Review time

- For the exam, make sure you’ve completed the labs (at least thought about them)
- Mainly on C/CPP
- Understand the slides we did in class
- Should understand the programming concepts we have covered
- Basics idea Perl
Word list

- Compiling
- Linking
- Reference parameter
- Variable scope
- Stdio.h
- Stdlib.h
- cout
- cast
- Inline
- makefiles
- Preprocessor
- Typedef
- Struct
- Pointer
- . Vs ->
- Function pointer
- Reference
- const
- malloc
Word list II

- getopt
- constructor
- destructor
- iostream
- overloading
- extern
- private
- Public
- GDB
- Cgi
- GET/POST
- overload
- overriding
- Template
- This
- Friend class
- New/delete
- virtual
C

- Basic constructs
- Basic type
- Advanced types
- (review labs and class examples)
- Memory stuff – understand what is happening
- Arrays
- Functions
- Pointers
- Debuggers
C

- Working with CGI
- Working on different platforms
- Makefiles
- How to built libraries and including code
C++

- Basic language
- Difference to c
- Classes
- Permissions
- new/free memory allocations
- Inheritance and polymorphism
- Keywords
- Working with files....
Closing Remarks

- If you like this.....just the beginning
- If you didn’t ..... You now know how complicated it is....never trust a program 😊

- Hope you had a fun semester..
- Final will be posted on courseworks....
Time permitting

- Anyone want to show off their project 1?
■ Bonus section 😊
Php.net

- developed 1994-1995 as “Personal Home Page” tools, by Rasmus Lerdorf
  - first as collection of perl scripts and then own interpreter

- V1.0 was a quick tool for embedding sql queries in a web page
- V2.0 structured code was added but with a buggy language parser
- V3.0 fixed parser bugs - June 1998 introduced object oriented ideas
- V4.0 more object, and passing variables in the system modified
- V5.0 new engine, many fixes etc

- Early as Jan 1999, 100,000 web pages were using php!!! Much higher now!
Some say..

- **php is better than cgi because:**
  - it runs as part of the web server process and doesn’t require forking (unlike cgi)
  - it runs faster than cgi
  - it’s faster to write...
  - Tons of libraries supported

- **php was designed to run with apache web server on unix**
  - but also runs on windows and mac

- **Did I mention…it’s free!**
  - One important way of getting something adopted….don’t underestimate the power of a ‘free lunch’
LAMP

- Linux
- Apache
- Mysql
- Perl/PHP/Python
- **php** is coded in C
  - has a well-defined API
  - extensible

- The way it runs:
  - A php engine is installed as part of a web server
  - The engine runs the php script and produces html, which gets passed back to the browser
  - So user never sees the php code (if done right)
3 different ways!

- hello.php (plain php)
- hello2.php (php embedded in html)
- hello3.php (uses <?php start tag)
Hello.php

<?
print "hello world!";
?>
Hello2.php

<html>
<body bgcolor=#000000 text=#ffffffff>
<?
print "hello world!";
?>
</body>
</html>
Hello3.php

<html>
<body bgcolor=#000000 text=#ffffff>
<?php
print "hello world!";
?>
</body>
</html>
basics

- php start and end tags: <? ... ?>
- also: <?php ... ?>
- semi-colon ends a statement (like C)
- string constants surrounded by quotes ("), or (')
- you can embed multiple php blocks in a single html file
- variable names are preceded by dollar sign ($)
- user input is through html forms
- the language is case-sensitive, but calls to built-in functions are not
  - Any ideas why ???????

- identifiers are made of letters, numbers and underscore (_); and cannot begin with a number
- expressions are similar to C
Data types

- integers
- floating-point numbers
- strings
- loosely typed (you don’t have to declare a variable before you use it)
- conversion functions: intval, doubleval, strval, settype
- settype( <value>, <newtype> ) where newtype="integer", "double" or "string"
- typecasting: (integer), (string), (double), (array), (object)
operators

- mathematical: +, -, *, /, %, ++, --
- relational: <, >, <=, >=, ==, !=
- logical: AND, &&, OR, ||, XOR, !
- bitwise: &, |, ^ (xor), ~ (ones complement), >>, <<
- assignment: =, =, -=, *=, /=,
- other:
  - .  concatenate
  - -> references a class method or property
  - => initialize array element index
Conditionals

- if/elseif/else:
  if ( <expression1> ) {
  <statement(s)>
  }
elseif ( <expression2> ) {
  <statement(s)>
  }
else {
  <statement(s)>
  }
Conditional II

- tertiary operator:
  `<conditional-expression> ? <true-expression> : <false-expression>`;

- switch:
  ```
  switch( <root-expression> ) {
  case <case-expression>:<statement(s)>;
  break;
  default:
  <statement(s)>;
  break;
  }
  ```
loops

- while
  while ( <expression> ) {
  <statement(s)>;
  }

- do-while
  do {
  <statement(s)>;
  } while ( <expression> );

- for
  for ( <initialize> ; <continue> ; <increment> ) {
  <statement(s)>;
  }

- break:
  - execution jumps outside innermost loop or switch
other

- exit() function
  - halts execution, meaning that no more code (php or html) is sent to the browser

- built-in constants
  - PHP_VERSION
  - __FILE__, __LINE__
  - TRUE = 1, FALSE = 0
  - M_PI = pi (3.1415927....)
Writing your own functions

- declared just like C:
  ```
  function <name> ( args ) {
  <body>
  [return <value>]
  }
  
  called just like C
  
  arguments (and local variables) are local, and don’t exist when you exit the function; but you can use “static” to declare a variable so that when you call a function again, the value is retained
  
  use the “global” statement to declare global variables that you want to be able to access from within a function, or the GLOBALS array (which is like a perl hash)
  e.g., GLOBALS[’username’]
  
  recursion is okay, but be careful!
Simple 1

<?
$today = date("l F d, Y");
$yourname = $_POST['yourname'];
$cost = doubleval( $_POST['cost'] );
<numdays = intval( $_POST['numdays'] );
?>

<html>
<body>
today is:

<?
PRINT( "$today<br>"");
PRINT( "$yourname, you will be out \$" );
print( doubleval( $cost * $numdays ));
print(" for buying lunch this week!"");
?>
</body>
</html>
arrays

- indexed using [...]
- indeces can be integers or strings (like a perl hash)
- when strings are indeces, it’s called an “associative array”
- array() function can be used to initialize an array
- e.g., $var = array( value0, value1, value2, ... );
- use the => operator to define the index:
  $var = array( 1=>value1, value2, ... );
  $var = array( "a"=>value1, "b"=>value2, ... );

- multidimensional arrays are okay (like C)
<html>
<body bgcolor=#ffffff>
<?$states = array( "CA","NY" );
print "here are the states:<br>
for ( $i=0; $i<count( $states ); $i++ ) {
    print "-- $states[$i]<br>
}
print "<p>",
$cities = array( "CA"=>array( "san francisco","los angeles" ),
                 "NY"=>array( "new york","albany","buffalo" ));
print "here are the CA cities:<br>
for ( $i=0; $i<count( $cities["CA"] ); $i++ ) {
    print( "-- "$cities["CA"][$i]."<br>
}
print "here are the NY cities:<br>
for ( $i=0; $i<count( $cities["NY"] ); $i++ ) {
    print( "-- "$cities["NY"][$i]."<br>
}
Code II

print "<p>";
$states[] = "MA";
print "now here are the states:<br>";
for ( $i=0; $i<count( $states ); $i++ ) {
    print "-- $states[$i]<br>";
}
$cities[] = "MA";
$cities["MA"][] = "boston";
print "here are the MA cities:<br>";
for ( $i=0; $i<count( $cities["MA"] ); $i++ ) {
    print( "-- ".$cities["MA"][$i]."<br>");
}

?>
</body>
</html>
classes

- defining a class:

```php
class <class-name> {
    // declare properties
    // declare methods
}
```

- use just like java and c++
- example: myclass.php and userclass.php
- note use of include statement
userclass.php

```php
<?

class user {

    // properties
    var $name;
    var $password;
    var $last_login;

    // methods
    function init( $inputname, $inputpassword ) {
        $this->name = $inputname;
        $this->password = $inputpassword;
        $this->last_login = time();
    }

    function getLastLogin() {
        return( date( "M d Y", $this->last_login ));
    }

}
```
myclass.php

<html>
<body>
<?
include "userclass.php";

$currentuser = new user;
$currentuser->init( "yaddi","cat" );

print( "name = ".$currentuser->name."<br>" );
print( "last login = ".$currentuser->getLastLogin() );

?>
</body>
</html>
<?php

class Car {
    public $miles; // variable that can be accessed outside the class
    private $mpg;  // variable that can only be accessed within the class
    protected $mph; // variable that can only be accessed from within the class, and
                    // from any inherited child classes

    public function __construct($param) { // constructor is called when object "Car" is
        $this->miles = $param;
    }

    public function start() {
        // starts the car...
    }

    public function stop() {
        // stops the car...
    }

    public function getMpg() {
        return $this->mpg;
    }
}

$car = new Car($param);
echo $car->miles; // echos the value of the property "miles" of the class "Car"
?>
I/O

- get input from html forms using
  $_POST['<name>']
  $_GET['<name>']
  $_REQUEST['<name>']

- file I/O
  ```php
  $fp = fopen( "filename","w" );
  fwrite( $fp,"stuff" );
  fclose( $fp );
  
  note that fopen second argument mode is like C)
  ```
More than just reacting

- We have been working with perl/c/cpp in a static context
- Some information is presented to the user
- React to user input

- Is this how Google Maps works?
Ajax

- Asynchronous JavaScript And XML

- technique for developing interactive applications over the web

- Style
- Platform
- Format
- XMLHttpRequest
- Objects
Basic HTML

- Specific set of tags (depends on version)
- Up to user to set things correctly
- Most browsers will attempt to figure out what you want
  - Example not putting end body end html, will still work
Advanced HTML

- CSS
  - Cascading style sheets
    - Define format of the WebPages
    - Central location of style
    - With a few clicks can completely change thousands of WebPages.

- DOM
  - Document object model
    - Formal data structure to represent web based documents/information

- Client side scripting
DOM problems

- Different browsers supported things differently.

```javascript
if (document.getElementById &&
    document.getElementsByTagName) {
    // as the key methods getElementById and getElementsByTagName
    // are available it is relatively safe to assume W3CDOM support.

    obj = document.getElementById("navigation")
    // other code which uses the W3CDOM.
    // .....  
} 
```
Examples

javascript

- Client side
  - PHP & CGI were both server side
- Developed under Netscape as LiveScript
  - Currently 1.5
- Developed under IE as Jscript
- Object oriented approach
- Syntax like c
  - No input/output support native
  - Keywords
  - DOM for interfacing between server and client
- Can evaluate reg expressions (eval)
Javascript

- Heavy use of defined functions
  - Example: MouseOver
- Need to adopt to specific browser if doing anything fancy
- Adobe
  - Support javascript in pdf
- MAC
  - Dashboard widgets
Programming

- You need to learn on your own
- Many good books/websites
- Most of the time .js file if not in html
- Powerful example:
  - Thunderbird/firefox
- Get good debugger