Lecture 10

Perl (cont.)
UNIX Development Tools

Today

- Finish introduction to Perl
- CGI scripting with Perl
- Talk about next assignment
- UNIX software development tools

RE (cont.)

- **split** string using RE (whitespace by default)
```
@fields = split /:/, "::ab:cde:f";
# gets ("","","ab","cde","f")
```
- **join** strings into one
```
$str = join "-", @fields;  # gets "--ab-cde-f"
```
- **grep** something from a list
  - Similar to UNIX grep, but not limited to using regular expressions

```
@selected = grep(!/^#/, @code);
```

Awk Influence

- **$,** field separator OFS
- **$/** input record separator (line delimiter) RS

```
undef $/;
$str = <MYFILE>; # str gets whole file
```

- **$** output record separator (for **print**) ORS

Subroutines

- Defined with **sub:** `sub myfunc { ... }`
- Subroutines calls are prefaced with **&**. e.g. **&foo**
- Any of the three principal data types may be passed as parameters or used as a return value
- **$_** passed by default
- Parameters are received by the subroutines in the special array `@_: $_[0], $_[1], ...`
- The scalars in **@_** are implicit aliases for the ones passed
- By default, value of last expression evaluated is returned. Override with **return** statement

Lexical Variables

- Local (lexical) variables can be declared with **my**

```
my $val = 10;
my($param1, $param2) = @_;
```

- Can also be used in any block
- Use the **use strict** pragma to enforce good programming practice

```
#!/usr/bin/perl -w
use strict;
$foo = "bar";      # not localized by my
print "$foo\n";
```

Subroutine Examples

```
&foo;               # call subroutine foo with $_
&foo(@list);        # call foo passing an array
$x = &foo('red', 3, @arr);
@list = &foo;       # foo returns a list

sub simple {
   my $sum;
   foreach $_ (@_) {
       $sum += $_;
   }
   return $sum;
}

$foo = 'myroutine'
&$foo(@list);       # call subroutine indirectly
```

Another Subroutine Example

```
@nums = (1, 2, 3);
$num = 4;
@res = dec_by_one(@nums, $num); # @res=(0, 1, 2, 3)
                                # (@nums,$num)=(1, 2, 3, 4)
dec_by_1(@nums, $num);          # (@nums,$num)=(0, 1, 2, 3)

sub dec_by_one {
    my @ret = @_;               # make a copy
    for my $n (@ret) { $n-- }
    return @ret;
}

sub dec_by_1 {
    for (@_) { $_-- }
}
```

Pass-by-reference

```
@x = (1, 2);
@y = (3, 4);
@z = &vector_add( \@x, \@y );

sub vector_add {
   my ($u, $v) = @_;
   # @$u and @$v refer to @x and @y
   my @sum;
   $sum[0] = $$u[0] + $$v[0];
   $sum[1] = $$u[1] + $$v[1];
   return @sum;
}
```

String Functions

- Several C string manipulation functions:
  - crypt, index, rindex, length, substr, sprintf
- Adds others:
  - **chop** removes the last character from a string
  - **chomp** removes trailing string corresponding to **$/**, normally newline
  - work with scalars or arrays

```
@stuff = ("hello\n", "hi\n", "ola\n");
chomp(@stuff);
```

Other Built-in Functions

- Numeric functions
  - abs, sin, cos, log, sqrt, rand
- Functions for processes and process groups
  - kill, sleep, system, waitpid
- Time functions
  - localtime, time

```
$time_str = localtime;
print $time_str;  # Wed Nov 10 19:24:30 2004
```

Good Way to Learn Perl

- a2p
  - Translates an **awk** program to Perl
- s2p
  - Translates a **sed** script to Perl

Modules

- Modules are reusable code with specific functionality
- Standard modules are distributed with Perl, others can be obtained from CPAN
- Include modules in your program with **`use`**, e.g. `use CGI;`

Example — Mail::Mailer

```
use Mail::Mailer;

$mailer = Mail::Mailer->new("sendmail");
$mailer->open({ From    => "elee\@cims.nyu.edu",
                To      => "kornj\@cs.nyu.edu",
                Subject => "Lecture 10"
              })
    or die "Can't open mailer: $!\n";
print $mailer "Let's cancel today's lecture\n";
$mailer->close( );
```

CGI Review

- Allow web server to communicate with other programs
- Web pages created dynamically according to user input
- 2 methods for sending form data:
  - **GET** - variable=value pairs in the environment variable **`QUERY_STRING`** (from **`%ENV`** in Perl)
  - **POST** - variable=value pairs in STDIN

CGI Review (cont.)

GET:

```
GET /cgi-bin/myscript.pl?name=Bill%20Gates&
title=Chairman HTTP/1.1
```

POST:

```
POST /cgi-bin/myscript.pl HTTP/1.1
```

…more headers…

`name=Bill%20Gates&title=Chairman`

A (rather ugly) CGI Script

```
#!/usr/local/bin/perl

$size_of_form_info = $ENV{'CONTENT_LENGTH'};
read (STDIN, $form_info, $size_of_form_info);

# Split up each pair of key/value pairs
foreach $pair (split (/&/, $form_info)) {
   # For each pair, split into $key and $value variables
   ($key, $value) = split (/=/, $pair);
   # Get rid of the pesky %xx encodings
   $key =~ s/%([\dA-Fa-f][\dA-Fa-f])/pack ("C", hex ($1))/eg;
   $value =~ s/%([\dA-Fa-f][\dA-Fa-f])/pack ("C", hex ($1))/eg;
   # Use $key as index for $parameters hash, $value as value
   $parameters{$key} = $value;
}

# Print out the obligatory content type line
print "Content-type: text/plain\n\n";

# Tell the user what they said
print "Your birthday is on " . $parameters{birthday} . ".\n";
```

CGI.pm

- Interface for parsing and interpreting query strings passed to CGI scripts
- Methods for creating generating HTML
- Methods to handle errors in CGI scripts

Script Using CGI.pm

```
#!/usr/local/bin/perl -w

use CGI;

my $query = CGI->new();

my $bday = $query->param("birthday");

print $query->header(-type => 'text/html');
print "Your birthday is $bday.";
```

The Mailer Form

**Formatting tags have been stripped

```
<FORM method="POST"
   action="http://www.cims.nyu.edu/~ernestl/cgi-
   bin/mailer.cgi">
From: <INPUT type="text" name="from" size="32">
To: <INPUT type="text" name="rcpt" size="32">
Subject: <INPUT type="text" name="subject" size="32">
Message:<TEXTAREA name="msg" rows="10" cols="60">
Enter your message here.
</TEXTAREA>
<INPUT type="submit" value="Send">
</FORM>
```

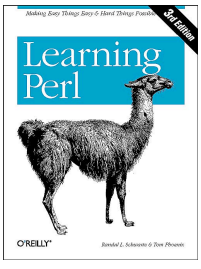
The Mailer CGI Script

```
use CGI;
use Mail::Mailer;

my $query = CGI->new();
my $from_user = $query->param("from");
my $to_user = $query->param("rcpt");
my $subject = $query->param("subject");
my $msg = $query->param("msg");

$mailer = Mail::Mailer->new("sendmail");
$mailer->open({ From    => $from_user,
                To      => $to_user,
                Subject => $subject
              }) or die "Can't open mailer: $!\n";
print $mailer $msg;
$mailer->close( );

print $query->header(-type => "text/html");
print $query->start_html(-title => "Message Sent");
print $query->p("Your message to $to_user has been sent!\n");
print $query->end_html();
```

mod_perl

- Perl interpreter embedded in Apache web server
- No need to start new process for every CGI request

Further Reading



Software Development Tools

Types of Development Tools

- Compilation and building: **make**
- Managing files: **RCS**, **SCCS**, **CVS**
- Editors: **vi**, **emacs**
- Archiving: **tar**, **cpio**, **pax**, **RPM**
- Configuration: **autoconf**
- Debugging: **gdb**, **dbx**, **prof**, **strace**, **purify**
- Programming tools: **yacc**, **lex**, **lint**, **indent**

Make

- **make**: A program for building and maintaining computer programs
  - developed at Bell Labs around 1978 by S. Feldman (now at IBM)

- Instructions stored in a special format file called a "**makefile**".

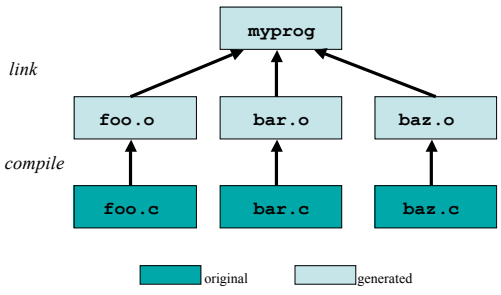
Make Features

- Contains the build instructions for a project
  - Automatically updates files based on a series of dependency rules
  - Supports multiple configurations for a project
- Only re-compiles necessary files after a change (conditional compilation)
  - Major time-saver for large projects
  - Uses timestamps of the intermediate files
- Typical usage: executable is updated from object files which are in turn compiled from source files

Compilation Phases

Component	Input	Output
preprocessor	source code	pre-processed source code
compiler	pre-processed source code	assembly source code
assembler	assembly source code	object file
linker	object files	executable file

Dependency Graph



Makefile Format

- Rule Syntax:

```
<target>: <dependency list>
        <command>
```

  - The `<target>` is a list of files that the command will generate
  - The `<dependency list>` may be files and/or other targets, and will be used to create the target
  - It ***must*** be a **tab** before `<command>`, or it won't work
  - The first rule is the default `<target>` for `make`

Examples of Invoking Make

- **`make -f makefile`**
- **`make target`**
- **`make`**
  - looks for file **makefile** or **Makefile** in current directory, picks first target listed in the **makefile**

Make: Sequence of Execution

- Make executes all commands associated with *target* in **makefile** if one of these conditions is satisfied:
  - file ***target*** does not exist
  - file ***target*** exists but one of the source files in the ***dependency list*** has been modified more recently than ***target***

Example Makefile

```
# Example Makefile
CC=g++
CFLAGS=-g -Wall -DDEBUG

foobar: foo.o bar.o
        $(CC) $(CFLAGS) -o foobar foo.o bar.o

foo.o: foo.cpp foo.h
        $(CC) $(CFLAGS) -c foo.cpp

bar.o: bar.cpp bar.h
        $(CC) $(CFLAGS) -c bar.cpp

clean:
        rm foo.o bar.o foobar
```

Make Power Features

- Many built-in rules
  - e.g. C compilation
- "Fake" targets
  - Targets that are not actually files
  - Can do just about anything, not just compile
  - Like the "*`clean`*" target
- Forcing re-compiles
  - *`touch`* the required files
  - *`touch`* the Makefile to rebuild everything

Make Patterns and Variables

- Variables (macros):
  - **`VAR = <rest of line>`** Set a variable
  - **`$(VAR)`** Use a variable
- Suffix Rules
  - **`.c.o:`** specifies a rule to build **x.o** from **x.c**
  - Default:

```
.c.o:
        $(CC) $(CFLAGS) -c $<
```

- Special:
  - **`$@:`** target
  - **`$<:`** dependency list
  - **`$*:`** target with suffix deleted

Version Control

- Provide the ability to store/access and protect all of the versions of source code files
- Provides the following benefits:
  - If program has multiple versions, it keeps track only of differences between multiple versions.
  - Multi-user support. Allows only one person at the time to do the editing.
  - Provides a way to look at the history of program development.

Version Control Systems

- **SCCS**: UNIX Source Code Control System
  - Rochkind, Bell Labs, 1972.
- **RCS**: Revision Control System
  - Tichy, Purdue, 1980s.
- **CVS**: Concurrent Versions System
  - Grune, 1986, Berliner, 1989.

RCS Basic Operations

- Set up a directory for RCS:
  - `mkdir RCS`
- Check in a new file into the repository
  - `ci filename`
- Check out a file from the repository for reading
  - `co filename`
- Check out a file from the repository for writing
  - `co -l filename`
  - Acquires lock
- Compare local copy of file to version in repository
  - `rcsdiff [-r<ID>] filename`

RCS Keywords

- Keywords in source files are expanded to contain RCS info at checkout
  - `$keyword$` → `$keyword: value $`
  - Use `ident` to extract RCS keyword info
- $*Author*$ Username of person checked in the revision
- $*Date*$ Date and time of check-in
- $*Id*$ A title that includes the RCS filename, revision number, date, author, state, and (if locked) the person who locked the file
- $*Revision*$ The revision number assigned

SCCS Equivalents

Function	RCS	SCCS
Setup	`mkdir RCS`	`mkdir SCCS`
Check in new foo.c	`ci foo.c`	`sccs create foo.c`
Check in update to foo.c	`ci foo.c`	`sccs delta foo.c`
Get read-only foo.c	`co foo.c`	`sccs get foo.c`
Get writeable foo.c	`co -l foo.c`	`sccs edit foo.c`
Version history of foo.c	`rlog foo.c`	`sccs print foo.c`
Compare foo.c to v1.1	`rcsdiff -r1.1 foo.c`	`sccs diffs -r1.1 foo.c`

CVS Major Features

- No exclusive locks like RCS
  - No waiting around for other developers
  - No hurrying to make changes while others wait
  - Avoid the "lost update" problem
- Client/Server model
  - Distributed software development
- Front-end tool for RCS with more functions

CVS Repositories

- All revisions of a file in the project are in the repository (using RCS)
- Work is done on the checkout (working copy)
- Top-level directories are modules; checkout operates on modules
- Different ways to connect

CVSROOT

- Environment Variable
- Location of Repository
- Can take different forms:
  - Local file system: **`/usr/local/cvsroot`**
  - Remote Shell: **`user@server:/usr/local/cvsroot`**
  - Client/Server: **`:pserver:user@server:/usr/local/cvsroot`**

Getting Started

- **`cvs [basic-options] <command> [cmd-options] [files]`**
- Basic options:
  - **`-d <cvsroot>`** Specifies CVSROOT
  - **`-H`** Help on command
  - **`-n`** Dry run
- Commands
  - import, checkout
  - update, commit
  - add, remove
  - status, diff, log
  - tag...

Setting up CVS

- Importing source
  - Generates a new module
  - **cd** into source directory
  - **`cvs -d<cvsroot> import <new-module> <vendor-branch> <release-tag>`**
  - **`cvs -d<cvsroot> checkout <module-name>`**

Managing files

- Add files: **add** (**`cvs add <filename>`**)
- Remove files: **remove** (**`cvs remove <filename>`**)
- Get latest version from repository: **update**
  - If out of sync, merges changes. Conflict resolution is manual.
- Put changed version into repository: **commit**
  - Fails if repository has newer version (need update first)
- View extra info: **status**, **diff**, **log**
- Can handle binary files (no merging or diffs)
- Specify a symbolic tag for files in the repository: **tag**

tar: Tape ARchiver

- **tar**: general purpose archive utility (not just for tapes)
  - Usage: **`tar [options] [files]`**
  - Originally designed for maintaining an archive of files on a magnetic tape.
  - Now often used for packaging files for distribution
  - If any files are subdirectories, **tar** acts on the entire subtree.

tar: archiving files options

- **`c`** creates a tar-format file
- **`f filename`** specify filename for tar-format file,
  - Default is `/dev/rmt0`.
  - If **`-`** is used for filename, standard input or standard output is used as appropriate
- **`v`** verbose output
- **`x`** allows to extract named files

tar: archiving files (continued)

- `t` generates table of contents
- `r` unconditionally appends the listed files to the archive files
- `u` appends only files that are more recent than those already archived
- `L` follow symbolic links
- `m` do not restore file modification times
- `l` print error messages about links it cannot find

cpio: copying files

- **cpio:** copy file archives in from or out of tape or disk or to another location on the local machine
- Similar to **tar**
- Examples:
  - **Extract:** `cpio -idtu [patterns]`
  - **Create:** `cpio -ov`
  - **Pass-thru:** `cpio -pl directory`

cpio (continued)

- `cpio -i [dtum] [patterns]`
  - Copy in (extract) files whose names match selected patterns.
  - If no pattern is used, all files are extracted
  - During extraction, older files are not extracted (unless `-u` option is used)
  - Directories are not created unless `-d` is used
  - Modification times not preserved with `-m`
  - Print the table of contents: `-t`

cpio (continued)

- `cpio -ov`
  - Copy out a list of files whose names are given on the standard input. `-v` lists files processed.
- `cpio -p [options] directory`
  - Copy files to another directory on the same system. Destination pathnames are relative to the named directory
  - Example: To copy a directory tree:
    - `find . -depth -print | cpio -pdumv /mydir`

pax: replacement for cpio and tar

- **P**ortable **A**rchive e**X**change format
- Part of POSIX
- Reads/writes **cpio** and **tar** formats
- Union of **cpio** and **tar** functionality
- Files can come from standard input or command line
- Sensible defaults
  - `pax -wf archive *.c`
  - `pax -r < archive`

Distributing Software

- Pieces typically distributed:
  - Binaries
  - Required runtime libraries
  - Data files
  - Man pages
  - Documentation
  - Header files
- Typically packaged in an archive:
  - e.g., `perl-solaris.tgz` or `perl-5.8.5-9.i386.rpm`

Packaging Source: autoconf

- Produces shell scripts that automatically configure software to adapt to UNIX-like systems.
  - Generates configuration script (configure)
- The configure script checks for:
  - programs
  - libraries
  - header files
  - typedefs
  - structures
  - compiler characteristics
  - library functions
  - system services

  and generates makefiles

Installing Software From Tarballs

```
tar xzf <gzipped-tar-file>
cd <dist-dir>
./configure
make
make install
```

Debugging

- The ideal:

  do it right the first time
- The reality:

  bugs happen
- The goal:

  exterminate, quickly and efficiently

Debuggers

- Advantages over the “old fashioned” way:
  - you can step through code as it runs
  - you don’t have to modify your code
  - you can examine the entire state of the program
    - call stack, variable values, scope, etc.
  - you can modify values in the running program
  - you can view the state of a crash using core files

Debuggers

- The **GDB** or **DBX** debuggers let you examine the internal workings of your code while the program runs.
  - Debuggers allow you to set *breakpoints* to stop the program's execution at a particular point of interest and examine variables.
  - To work with a debugger, you first have to recompile the program with the proper debugging options.
  - Use the **`-g`** command line parameter to **cc, gcc,** or **CC**
    - Example: **`cc -g -c foo.c`**

Using the Debugger

- Two ways to use a debugger:
  1. Run the debugger on your program, executing the program from within the debugger and see what happens
  2. Post-mortem mode: program has crashed and core dumped
     - You often won't be able to find out exactly what happened, but you usually get a stack trace.
     - A stack trace shows the chain of function calls where the program exited ungracefully
     - Does not always pinpoint what caused the problem.

GDB, the GNU Debugger

- Text-based, invoked with:

  `gdb [<programfile> [<corefile>|<pid>]]`

- Argument descriptions:

`<programfile>`	executable program file
`<corefile>`	core dump of program
`<pid>`	process id of already running program

- Example:

  `gdb ./hello`

- Compile `<programfile>` with `-g` for debug info

Basic GDB Commands

- General Commands:

`file [<file>]`	selects `<file>` as the program to debug
`run [<args>]`	runs selected program with arguments `<args>`
`attach <pid>`	attach gdb to a running process `<pid>`
`kill`	kills the process being debugged
`quit`	quits the gdb program
`help [<topic>]`	accesses the internal help documentation

- Stepping and Continuing:

`c[ontinue]`	continue execution (after a stop)
`s[tep]`	step one line, entering called functions
`n[ext]`	step one line, without entering functions
`finish`	finish the function and print the return value

GDB Breakpoints

- Useful breakpoint commands:

`b[reak] [<where>]`	sets breakpoints. `<where>` can be a number of things, including a hex address, a function name, a line number, or a relative line offset
`[r]watch <expr>`	sets a watchpoint, which will break when `<expr>` is written to [or read]
`info break[points]`	prints out a listing of all breakpoints
`clear [<where>]`	clears a breakpoint at `<where>`
`d[elete] [<nums>]`	deletes breakpoints by number

Playing with Data in GDB

- Commands for looking around:

`list [<where>]`	prints out source code at `<where>`
`search <regexp>`	searches source code for `<regexp>`
`backtrace [<n>]`	prints a backtrace `<n>` levels deep
`info [<what>]`	prints out info on `<what>` (like local variables or function args)
`p[rint] [<expr>]`	prints out the evaluation of `<expr>`

- Commands for altering data and control path:

`set <name> <expr>`	sets variables or arguments
`return [<expr>]`	returns `<expr>` from current function
`jump <where>`	jumps execution to `<where>`

Tracing System Calls

- Most operating systems contain a utility to monitor system calls:
  - Linux: **strace**, Solaris: **truss**, SGI: **par**

```
   27mS[  1]               : close(0) OK
   27mS[  1]               : open("try.in", O_RDONLY, 017777627464)
   29mS[  1]               : END-open() = 0
   29mS[  1]               : read(0, "1\n2\n|/bin/date\n3\n|/bin/sleep 2", 2048) = 31
   29mS[  1]               : read(0, 0x7fff26ef, 2017) = 0
   29mS[  1]               : getpagesize() = 16384
   29mS[  1]               : brk(0x1001c000) OK
   29mS[  1]               : time() = 1003207028
   29mS[  1]               : fork()
   31mS[  1]               : END-fork() = 1880277
   41mS[  1]              (1864078): was sent signal SIGCLD
   31mS[  2]               : waitsys(P_ALL, 0, 0x7fff2590, WTRAPPED|WEXITED, 0)
   42mS[  2]               : END-waitsys(P_ALL, 0, {signo=SIGCLD, errno=0,
code=CLD_EXITED, pid=1880277, status=0}, WTRAPPED|WEXITED, 0) = 0
   42mS[  2]               : time() = 1003207028
```