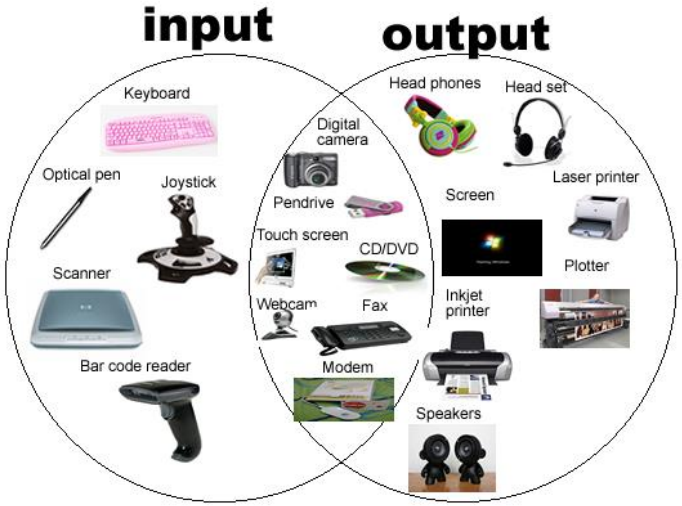
COMP 2021

Unix and Script Programming



Perl I/O

STDIN in Scalar and List Context

- Reading from STDIN is easy, and we have done it many times.

```
$a = <STDIN>;
```

- In a scalar context, this gives the next line of input, or undef if there are are no more lines (the user has hit CTRL-d).

- In a list context, this gives all remaining lines (all the lines until the user hits CTRL-d).

```
@a = <STDIN>;
```

Each element is one line, and includes the terminating newline (the last line may or may not have a newline).

Valid Input in `STDIN`

- Typically, a program will read in one line at a time and process the line:

```
while(defined($line = <STDIN>)){
        # process line
        print "$line";
}
```

- `defined()` returns `true` unless its argument is `undef`.
- As long as a line has been read, `<STDIN>` returns a defined value, and the loop continues.
- When `<STDIN>` has no more lines to read (it hits `CTRL-d`), it returns `undef`, terminating the loop.

STDIN Example

- Here is an example of program input and output.
- White text is typed by the user, red text is printed by the program or shell.

```
$ cat line1.pl
#!/usr/local/bin/perl5 -w
while(defined($line = <STDIN>)){
        # process line
        print "$line";
}
$ line1.pl
hi
hi
test
test
[CTRL-d]
$
```

STDIN Example

- We can also read and print the lines altogether.

```
$ cat line2.pl
#!/usr/local/bin/perl5 -w
@lines = <STDIN>;
# process lines
foreach my $i (@lines){
      print "$i";
}
$ line2.pl
hi
test
[CTRL-d]
hi
test
$
```

$_ in STDIN

- Perl has a shortcut for reading a value from <STDIN> into $_.
- Whenever a condition consists solely of <STDIN>, Perl automatically copies the line into $_.

```
while(<STDIN>){
# same as: while(defined($_ = <STDIN>)){
chomp;  # same as: chomp "$_";
print "$_\n";
}
```

- $_ is the default for many operations, such as chomp and print.

Input from <>

- Another way to read input is with <>.
- <> returns a single line in a scalar context, or all the remaining lines in a list context.
- However, unlike <STDIN>, <> gets its data from the file (or files) specified on the command line.

```
$ cat file1
This is file1.
$ cat file2
This is file2.
$ cat mycat.pl
#!/usr/local/bin/perl5 -w
while(<>){
      print;
}
$ mycat.pl file1 file2
This is file1.
This is file2.
$
```

Input from <>

- If you do not specify any filenames on the command line, <> reads from standard input automatically.

```
$ cat mycat.pl
#!/usr/local/bin/perl5 -w
while(<>){
        print;
}
$ mycat
hi
hi
test
test
[CTRL-d] $
```

- <> is the easiest way to read files in Perl (we will learn more advanced ways later).

Output to STDOUT

- We have also often used `print`.
- The `print` function takes a list of strings, and sends each to `STDOUT`, without adding any characters between them.

```
print "Hi Cindy!\n";
print "Hi ", "Cindy!", "\n";
print("Hi ", "Cindy!", "\n");
```

- Sometimes, you will need to add parentheses to `print`, especially when the first thing starts with a left parenthesis:

```
print (1+1), "COMP2021 is fun\n";   # problem!
print ((1+1), "COMP2021 is fun\n"); # ok
print 1+1, "COMP2021 is fun";       # also ok
```

`printf` Formatted Output

- You may wish to have more control over your output than `print` provides.

- The `printf` function takes a list of arguments, where the first argument is a *format control string*, which defines how to print the remaining arguments (like `printf` in C).

```
printf "%s %d %f\n", $s, $n, $real;
```

- Format types:

```
strings                  %s
integer numbers          %d
floating-point numbers   %f
```

`printf` Formatted Output

```
$ cat printf.pl
#!/usr/local/bin/perl5 -w
$s = "1234567890";
$n = 12345;
$real = 1234567.123;
printf "%15s %5d %10.2f\n", $s, $n, $real;
$ printf.pl
     1234567890 12345 1234567.12
$
```

- This example prints `$s` in a 15-character field, then space, then `$n` as a decimal integer in a 5-character field, then another space, then `$real` as a floating-point value with 2 decimal places in a 10-character field, and finally a newline.

COMP 2021

Unix and Script Programming

Perl File I/O

Filehandles

- A *filehandle* is the name for an I/O connection between your Perl program and the outside world.
- `STDIN` is a filehandle, representing the connection between Perl programs and the UNIX standard input.
- Filehandles can also refer to external files.
- Filehandles are usually ALL UPPERCASE.

Opening and Closing Filehandles

- To open a filehandle called `INPUTFILE` for reading the file named `infile` (assuming it is in the current working directory):

```
open(INPUTFILE, "<infile");
```

- To close the filehandle `INPUTFILE` after finished reading:

```
close(INPUTFILE);
```

Opening Filehandles

- To open a filehandle called `OUTPUTFILE` for writing the file `outfile` in the current working directory:

```
open(OUTPUTFILE, ">outfile");
```

- To open a filehandle called `APPENDFILE` for appending to the file `outfile` in the current working directory:

```
open(APPENDFILE, ">>outfile");
```

Reading with Filehandles

- Once a filehandle is open for reading, you can read lines from it just as you can from `STDIN`.
- For example, to read lines from the file `nameID`:

```
$ cat nameid
Mary 001
Tom  002
$ cat readfile.pl
#!/usr/local/bin/perl5 -w
open(INFILE, "<nameid");
while(<INFILE>){
        chomp;
        print "The name and ID are: $_\n";
}
close(INFILE);
$ readfile.pl
The name and ID are: Mary 001
The name and ID are: Tom 002
```

Writing with Filehandles

- If you want to `print` to a filehandle, you must place the filehandle immediately after the `print` keyword and before the arguments (no comma):

```
$ cat writefile1.pl
#!/usr/local/bin/perl5 -w
open(CINDYFILE, ">greeting");
print CINDYFILE "Hi Cindy!\n";
print CINDYFILE "Bye Cindy!\n";
close(CINDYFILE);
$ writefile1.pl
$ cat greeting
Hi Cindy!
Bye Cindy!
$
```

STDOUT vs. Writing with Filehandles

- **Do not get confused between STDOUT and output to a file:**

```
$ cat writefile2.pl
#!/usr/local/bin/perl5 -w
open(CINDYFILE, ">greeting");
print "Hi Cindy!\n";
print STDOUT "Hi again Cindy!\n";
print CINDYFILE "Bye Cindy!\n";
close(CINDYFILE);
$ writefile2.pl
Hi Cindy!
Hi again Cindy!
$ cat greeting
Bye Cindy!
$
```

Copy Among Files

- How to copy data from one file into another:

```
$ cat cindy1
Hi Cindy!
$ cat copyfile.pl
#!/usr/local/bin/perl5 -w
$a = "cindy1";
$b = "cindy2";
open(IN, “<$a”);
open(OUT, ">$b");
while(<IN>){
        print OUT $_;
}
close(IN);
close(OUT);
$ copyfile.pl
$ cat cindy2
Hi Cindy!
$
```

Error Checking

- Perl allows you to check the result of `open` and report an error if something went wrong (`open` returns `false` if there is a problem):

```
$ cat cindy
cat: cannot open cindy
$ cat errorcheck1.pl
#!/usr/local/bin/perl5 -w
$file = "cindy";
unless(open(IN, $file)){
        print "Sorry could not open $file\n";
}else{
        print "$file contains:\n";
        while(<IN>){
                print;
        }
}
close(IN);
$ errorcheck1.pl
Sorry could not open cindy
$
```

die

- `die` is a shortcut for error checking.
- `die` is like `exit`, but it prints an error message before stopping the program:

```
$ cat cindy
cat: cannot open cindy
$ cat errorcheck2.pl
#!/usr/local/bin/perl5 -w
$file = "cindy";
unless(open(IN, $file)){
        die "Sorry could not open file $file\n";
}
print "$file contains:\n";
while(<IN>){
        print;
}
close(IN);
$ errorcheck2.pl
Sorry could not open file cindy
$
```

`die` with Line Number

- If you leave off the `\n`, `die` will also attach the line number where it died:

```
$ cat cindy
cat: cannot open cindy
$ cat errorcheck3.pl
#!/usr/local/bin/perl5 -w
$file = "cindy";
unless(open(IN, $file)){
        die "Sorry could not open $file";    # no \n
}
print "$file contains:\n";
while(<IN>){
        print;
}
close(IN);
$ errorcheck3.pl
Sorry could not open cindy at errorcheck3.pl line
4.
$
```

die

- You can write it even shorter using || ("open that file or die!"):

```
$ cat cindy
cat: cannot open cindy
$ cat errorcheck4.pl
#!/usr/local/bin/perl5 -w
$file = "cindy";
open(IN, $file) || die "Sorry could not open
$file";
print "$file contains:\n";
while(<IN>){
      print;
}
close(IN);
$ errorcheck4.pl
Sorry could not open cindy at errorcheck4.pl
line 3.
$
```

die with $!

- Another useful thing in die is the $! variable, which contains the error string from the operating system:

```
$ cat cindy
cat: cannot open cindy
$ cat errorcheck5.pl
#!/usr/local/bin/perl5 -w
$file = "cindy";
open(IN, $file) || die "Sorry could not open $file:
$!\n";
print "$file contains:\n";
while(<IN>){
        print;
}
close(IN);
$ errorcheck5.pl
Sorry could not open cindy: No such file or directory
$
```

-x File Tests

- Perl also allows you to test the properties of files.
- Some common file tests:

```
-r    File or directory is readable
-w    File or directory is writable
-x    File or directory is executable
-o    File or directory is owned by user
-e    File or directory exists
-z    File exists and has zero size
      (directories are never empty)
-s    File exists and has nonzero size
      (returns size in bytes)
-f    Entry is a plain file
-d    Entry is a directory
-l    Entry is a symbolic link
```

-x File Tests

- File tests allow you to avoid overwriting an existing file with the same name.

```
$ cat cindy
Hi cindy!
$ cat test.pl
#!/usr/local/bin/perl5 -w
$file = "cindy";
if(-e $file){
        die "File already exists!\n";
}
open(OUT, ">$file") || die "Could not open $file:
$!\n";
print OUT "Bye cindy!\n";
close(OUT);
$ test.pl
File already exists!
$
```

Pop-up Exercise

- Given a file of special format, read and process it
  - File name as command line argument
  - Exit if file doesn't not exist
  - Extract every element individually
  - Report error if a line doesn't comply with format

Mary	mary@stu.ust.hk	20140088
Tom	tom@stu.ust.hk	20140999
Jerry	jerry@stu.ust.hk	20143366

A,1
B,2
C,3
D,4
E,5
F,6
G,7
H,8
I,9
J,10

You can Explore More

- Search a particular pattern in a file
- Read out all files under a directory
- Read file contents in the middle of the file