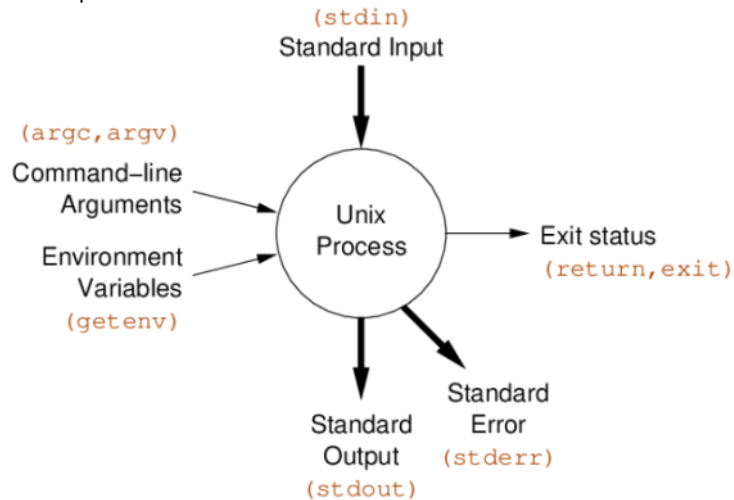


Filters

Monday, 3 June 2019 8:40 PM

Unix Processes

A Unix process executes in this environment.



Components of process environment (C programmer's view):

- `char *argv[]` - command line arguments
- `int argc` - size of `argv[]`
- `char *env[]` - name-value pairs from parent process
- `FILE *stdin` - input byte-stream, e.g. `getchar()`
- `FILE *stdout` - output byte-stream, e.g. `putchar()`
- `FILE *stderr` - output byte-stream, e.g. `fputc(c, stderr)`
- `exit(int)` - terminate program, set exit status
- `return int` - terminate `main()`, set exit status

C code for cat

```
// write bytes of stream to stdout
void process_stream(FILE *in) {
    while (1) {
        int ch = fgetc(in);
        if (ch == EOF) {
            break;
        }
        if (fputc(ch, stdout) == EOF) {
            fprintf(stderr, "cat:");
            perror("");
            exit(1);
        }
    }
}

// process files given as arguments
// if no arguments process stdin
int main(int argc, char *argv[]) {
    if (argc == 1) {
        process_stream(stdin);
    } else {
        for (int i = 1; i < argc; i++) {
            FILE *in = fopen(argv[i], "r");
            if (in == NULL) {
                fprintf(stderr, "cat: %s: ", argv[i]);
                perror("");
                return 1;
            }
            process_stream(in);
            fclose(in);
        }
    }
    return 0;
}
```

C code for wc

```
// count lines, words, chars in stream
void count_file(FILE *in) {
    int n_lines = 0, n_words = 0, n_chars = 0;
    int in_word = 0, c;
    while ((c = fgetc(in)) != EOF) {
        n_chars++;
        if (c == '\n') {
            n_lines++;
        }
        if (isspace(c)) {
            in_word = 0;
        } else if (!in_word) {
            in_word = 1;
            n_words++;
        }
    }
    printf("%6d %6d %6d", n_lines, n_words, n_chars);
}
```

Filters

A **filter** is a program that transforms a data stream. On Unix, filters are commands that:

- Read text from their standard input or specified files
- Perform useful transformations on the text stream
- Write the transformed text to their standard output

Shell I/O redirection can be used to specify filter source and destination.

filter < abc > xyz



Alternatively, most filters allow multiple sources to be specified

filter abc def ghi > xyz



In isolation, filters are reasonably useful. In combination they provide a very powerful problem-solving toolkit.

Filters are normally used in combination via a pipeline:

filter1 | filter2 | ... | filterN



Unix filters use common conventions for command line arguments:

- input can be specified by a list of file names
- if no files are mentioned, the filter reads from standard input (which may have been connected to a file)
- the filename "-" corresponds to standard input

If filter does not cope with named sources,
use cat at the start of the pipeline

```
# read from the file data1
filter data1
# or
filter < data1

# read from the files data1 data2 data3
filter data1 data2 data3

# read from data1, then stdin, then data2
filter data1 - data2
```

Filters normally perform variations on task. A selection of the variation is accomplished via the command-line options.

- Options are introduced by a - ("minus" or "dash")
- Options have a "short" form; - followed by a single letter (e.g. -v)
- Short form options can usually be combined (e.g. -va vs -a -v)
- -help or (-?) often gives a list of command-line options

Most filters have many options for controlling their behaviour. The Unix **man**ual entries describe how each option works. To find what filters are available: `man -k keyword`

The solution to all your problems: **RTFM**

Delimited Input

Many filters are able to work with text data formatted as fields (columns in spreadsheet terms). Such filters typically have an option for specifying the delimiter or field separator. (Unfortunately, they often make different assumptions about the default column separator)

Example (tab-separated columns): Example (colon-separated columns, old Unix password file):

```
John 99
Wen 75
Andrew 50
Wenjie 95
Yang 33
Sowmya 76
```

```
root:ZHo1HAHZw8As2:0:0:root:/root:/bin/bash
jas:nJz3ru5a/44Ko:100:100:John Shepherd:/home/jas:/bin/bash
cs1521:iZ3s09005eZY6:101:101:COMP1521:/home/cs1521:/bin/bash
cs2041:rX9KwSPqkLyA:102:102:COMP2041:/home/cs2041:/bin/bash
cs3311:mLRiCIvmtI902:103:103:COMP3311:/home/cs3311:/bin/bash
```

Example (verticalbar-separated columns, enrolment file):

```
COMP1511|2252424|Abbot, Andrew John |3727|1|M
COMP2511|2211222|Abdurjh, Saeed |3640|2|M
COMP1511|2250631|Accent, Aac-Ek-Murhg |3640|1|M
COMP1521|2250127|Addison, Blair |3971|1|F
COMP4141|2190705|Allen, David Peter |3645|4|M
COMP4960|2190705|Allen, David Pater |3645|4|M
```

cat: the simplest filter

The **cat** command copies its input to output unchanged (identity filter).

When supplied a list of file names, it concatenates them onto stdout.

Some options:

| | |
|----|--|
| -n | number output lines (starting from 1) |
| -s | squeeze consecutive blank lines into single blank line |
| -v | display control-characters in visible form (e.g. ^C) |

The **tac** command copies files, but reverses the order of lines.

tr: transliterate characters

The **tr** command converts text char-by-char according to a mapping.

```
tr 'sourceChars' 'destChars' < dataFile
```

Each input character from *sourceChars* is mapped to the corresponding character in *destChars*.

Example:

```
tr 'abc' '123' < someText
```

Has *sourceChars*='abc', *destChars*='123', so:

a → 1, b → 2, c → 3

Note: tr doesn't accept file name on command line

Characters that are not in *sourceChars* are copied unchanged to output. If there is no corresponding character (i.e. *destChars* is shorter than *sourceChars*), then the last char in *destChars* is used.

Shorthands are available for specifying character lists:

E.g. 'a-z' is equivalent to 'abcdefghijklmnopqrstuvwxyz'

Note: newlines will be modified if the mapping specification requires it.

Some options:

| | |
|----|--|
| -c | map all characters <i>not</i> occurring in <i>sourceChars</i> (complement) |
| -s | squeeze adjacent repeated characters out (only copy the first) |
| -d | delete all characters in <i>sourceChars</i> (no <i>destChars</i>) |

Examples:

```
# map ALL upper-case letters to lower-case equivalents
```

wc: the word counter

The **wc** command is a summarizing filter. It is useful with other filters to count things.

Some options:

| | |
|----|--|
| -c | counts the number of characters (incl. n) |
| -w | counts the number of words (non-white space) |
| -l | counts the number of lines |

Some filters find counting so useful that they define their own options for it (e.g. `grep -c`)

```
tr 'A-Z' 'a-z' < text

# simple encryption (a->b, b->c, ... z->a)
tr 'a-zA-Z' 'b-ZaB-ZA' < text

# remove all digits from input
tr -d '0-9' < text

# break text file into individual words, one per line
tr -cs 'a-zA-Z0-9' '\n' < text
```

head/tail: select lines

head prints the first *n* (default 10) lines of input. E.g. head file prints first 10 lines of file. The -n option changes number of lines head/tail prints.

tail prints the last *n* lines of input. E.g. tail -n 30 file prints last 30 lines of file.

We can combine head and tail to select a range of lines. E.g. head -n 100 | tail -n 20 copies lines 81..100 to output.

egrep: select lines matching a pattern

The **egrep** command only copies to output those lines in the input that match a specified pattern.

The pattern is supplied as a regular expression on the command line (and should be quoted using single-quotes).

Some options:

| | |
|--------------|---|
| -i | Ignore upper/lower-case difference in matching |
| -v | Only display lines that <i>do not</i> match the pattern |
| -w | Only match pattern if it makes a complete word |
| --color=auto | Highlights which patterns were matched |

egrep is one of a group of related filters using different kinds of pattern match:

- **grep** uses a limited form of POSIX regular expressions (no + ? | or parentheses)
- **egrep** or **grep -E** (extended grep) implements the full regex syntax
- **fgrep** or **grep -F** finds any of several (maybe even thousands of) fixed strings using an optimised algorithm.
- **grep -P** Perl-like extended regular expressions

(The name grep is an acronym for **G**lobally search with **R**egular **E**xpressions and **P**rint)

A **regular expression (regex)** defines a set of strings. A *regular expression* is usually thought of as a pattern. It specifies a possibly infinite set of strings. They can be succinct and powerful. Regular expressions libraries are available for most languages.

In the Unix environment:

- a lot of data is available in plain text format
- many tools make use of regular expressions for searching
- effective use of regular expressions makes you more productive

A POSIX standard for regular expressions defines the "pattern language" used by many Unix tools.

The Basics:

Regular expressions specify complex patterns concisely & precisely.

- **Default:** a character matches itself. E.g. **a** has no special meaning so it matches the character **a**.
- **Repetition:** **p*** denotes zero or more repetitions of **p**.
- **Alternation:** **pattern1 | pattern2** denotes the union of **pattern1** and **pattern2**.
E.g. **perl | python | ruby** matches any of the strings **perl**, **python** or **ruby**
- Parentheses are used for **grouping** e.g. **a(a)*** denotes a comma separated list of **a**'s.
- The special meanings of characters can be removed by **escaping** them with **** e.g. ***** matches the ***** character anywhere in the input.

The above 5 special characters **()*\|** are sufficient to express any regular expression but many more features are present for convenience & clarity.

Pattern for matching single characters:

- The special pattern **.** (dot) matches any single character.
- Square brackets provide convenient matching of any **one** of a set of characters.
[listOfCharacters] matches any single character from the list of characters. E.g. **[aeiou]** matches any vowel.
- A shorthand is available for ranges of characters **[first - last]**
Examples: **[a-e]** **[a-z]** **[0-9]** **[a-zA-Z]** **[A-Za-z]** **[a-zA-Z0-9]**

- The matching can be inverted [*listOfCharacters*]
E.g. `^[a-e]` matches any character *except* one of the first five letters
- Other characters lose their special meaning inside bracket expressions

Anchoring Matches:

We can insist that a pattern appears at the start or end of a string

- the start of the line is denoted by `^` (uparrow) E.g. `^[abc]` matches either a or b or c at the start of a string.
- the end of the line is denoted by `$` (dollar) E.g. `cat$` matches cat at the end of a string

Repetition:

We can specify repetitions of patterns

- `p*` denotes zero or more repetitions of `p`
- `p+` denotes one or more repetitions of `p`
- `p?` denotes zero or one occurrence of `p`

E.g. `[0-9]+` matches any sequence of digits (i.e. matches integers)

E.g. `[-'a-zA-Z]+` matches any sequence of letters/hyphens/apostrophes (this pattern could be used to match words in a piece of English text, e.g. it's, John, ...)

E.g. `^[^X]*X` matches any characters up to and including the first X

If a pattern can match several parts of the input, the first match is chosen.

Examples:

| Pattern | Text (with match underlined) |
|---------------------|------------------------------|
| <code>[0-9]+</code> | i= <u>1234</u> j=56789 |
| <code>[ab]+</code> | <u>aabbabababaa</u> cabba |
| <code>[+]+</code> | C <u>++</u> is a hack |

Capture Groups:

By default anything inside `()` is considered a capturing group; like storing what is found in a variable. Capturing groups are found from left to right of the regex and are numbered 1 to n.

To stop a pattern being a capturing group you put `?:` right after the opening bracket `(`.

E.g.

Pattern = `"(.*) b(?:an)*a"`

String = "I like bananas!"

Capture group 1: `(.*)` = I like

Capture group 2: `(?:an)*` = anan

Note that `(?:an)` is not a capture group

Recursion:

The recursive syntax is simple. In PCRE and Perl, you just add `(?R)` anywhere you like in your pattern. Basically, `(?R)` means "paste the entire regular expression right here, replacing the original `(?R)`".

E.g. `a(?R)?z` matches az, aazz, any pattern a with the same number of z

cut: vertical slice

The **cut** command prints selected parts of input lines. It can:

- can select fields (assumes tab-separated columnated input)
- can select a range of character positions

Some options:

| | |
|---------------------------|---|
| <code>-flistOfCols</code> | print only the specified fields (tab-separated) |
| <code>-clistOfPos</code> | print only chars in the specified position |
| <code>-d'c'</code> | use character c as the field separator |

Lists are specified as ranges (e.g. 1-5) or comma-separated (e.g. 2,4,5).

Examples:

```
# print the first column
cut -f1 data
# print the first three columns
cut -f1-3 data
# print the first and fourth columns
cut -f1,4 data
# print all columns after the third
cut -f4- data
# print the first three columns, if '|' -separated
cut -d'|' -f1-3 data
```

```
# print the first five chars on each line
cut -c1-5 data
```

Unfortunately, there's no way to refer to "last column" without counting the columns

paste: combine files

The **paste** command displays several text files "in parallel" on output. If the inputs are files a, b, c

- the first line of output is composed of the first lines of a, b, c
- the second line of output is composed of the second lines of a, b, c

Lines from each file are separated by a tab character or specified delimiter(s).

If files are different lengths, output has all lines from longest file, with empty strings for missing lines.

You can interleave lines instead with -s (serial) option.

Example: using paste to rebuild a file broken up by cut.

```
# assume "data" is a file with 3 tab-separated columns
cut -f1 data > data1
cut -f2 data > data2
cut -f3 data > data3
paste data1 data2 data3 > newdata
# "newdata" should look the same as "data"
```

sort: sort lines

The **sort** command copies input to output but ensures that the output is arranged in some particular order of lines. By default, sorting is based on the first characters in the line.

Other features of sort:

- understands that text data sometimes occurs in delimited fields. (so, can also sort fields (columns) other than the first (which is the default))
- can distinguish numbers and sort appropriately
- can ignore punctuation or case differences
- can sort files "in place" as well as behaving like a filter
- capable of sorting *very large* files

Some options:

| | |
|-------|--|
| -r | sort in descending order (reverse sort) |
| -n | sort numerically rather than lexicographically |
| -d | dictionary order: ignore non-letters and non-digits |
| -t'c' | use character c to separate columns (default: space) |
| -k'n' | sort on column n |

Note: the '' around the separator char are usually not necessary, but are useful to prevent the shell from mis-interpreting shell meta-characters such as '|'.

Hint: to specify TAB as the field delimiter with an interactive shell like bash, type CTRL-v before pressing the TAB key

Examples:

```
# sort numbers in 3rd column in descending order
sort -nr -k3 data
# sort the password file based on user name
sort -t: -k5 /etc/passwd
```

uniq: remove or count duplicates

The **uniq** command by default removes all but one copy of *adjacent* identical lines.

Some options:

| | |
|----|--|
| -c | also print number of times each line is duplicated |
| -d | only print (one copy of) duplicated lines |
| -u | only print lines that occur uniquely (once only) |

Surprisingly useful tool for summarising data, typically after extraction by cut. Always preceded by sort (why?).

```
# extract first field, sort, and tally
cut -f1 data | sort | uniq -c
```

xargs: run commands with arguments from standard input

Some options:

| | |
|--------------|--|
| -n max-args | Use at most max-args arguments per command line. |
| -P max-procs | Run up to max-procs processes at a time |

| | |
|----------------|---|
| -i replace-str | Replace occurrences of replace-str with words read from stdin |
|----------------|---|

```
# remove home directories of users named Andrew:
grep Andrew /etc/passwd | cut -d: -f6 | xargs rm -r
```

join: database operator

join merges two files using the values in a field in each file as a common key. The key field can be in a different position in each file, but the files must be ordered on that field. The default key field is 1.

Some options:

| | |
|------|--|
| -1 k | key field in first file is k |
| -2 k | key field in second file is k |
| -a N | print a line for each unpairable line in file N (1 or 2) |
| -i | ignore case |
| -t c | tab character is c |

Given these two data files with tab-separated fields

```
$ cat data1
Bugs Bunny      1953
Daffy Duck      1948
Donald Duck     1939
Goofy           1952
Mickey Mouse    1937
Nemo             2003
Road Runner     1949
$ cat data2
Warners Bugs Bunny
Warners Daffy Duck
Disney Goofy
Disney Mickey Mouse
Pixar Nemo
$ join -t' ' -2 2 -a 1 data1 data2
Bugs Bunny 1953 Warners
Daffy Duck 1948 Warners
Donald Duck 1939
Goofy 1952 Disney
Mickey Mouse 1937 Disney
Nemo 2003 Pixar
Road Runner 1949
```

sed: stream editor

The **sed** command provides the power of interactive-style editing in “filter-mode”. How to use it:

```
sed -e 'EditCommands' DataFile
sed -f EditCommandFile DataFile
```

How sed works:

- read each line of input
- check if it matches any patterns or line-ranges
- apply related editing commands to the line
- write the transformed line to output

The editing commands are very powerful and use the actions of many of the filters looked at so far. In addition, sed can:

- partition lines based on patterns rather than columns
- extract ranges of lines based on patterns or line numbers

Option -n (no printing):

- applies all editing commands as normal
- displays no output, unless p appended to edit command

Option -i lets you edit the file in place

Editing commands:

| | |
|--------------------|---|
| p | print the current line |
| d | delete (don't print) the current line |
| s/RegExp/Replace/ | substitute first occurrence of string matching RegExp by Replace string |
| s/RegExp/Replace/g | substitute all occurrences of string matching RegExp by Replace string |

All editing commands can be qualified by line addresses or line selector patterns to limit lines where command is applied:

| | |
|-----------------------|---|
| LineNo | selects the specified line |
| StartLineNo,EndLineNo | selects all lines between specified line numbers |
| /RegExp/ | selects all lines that match RegExp |
| /RegExp1/,/RegExp2/ | selects all lines between lines matching reg exps |

```
# print all lines
sed -n -e 'p' < file

# print the first 10 lines
sed -e '10q' < file
sed -n -e '1,10p' < file

# print lines 81 to 100
sed -n -e '81,100p' < file

# print the last 10 lines of the file?
sed -n -e '$-10,$p' < file # does NOT work

# print only lines containing 'xyz'
sed -n -e '/xyz/p' < file

# print only lines NOT containing 'xyz'
sed -e '/xyz/d' < file

# show the passwd file, displaying only the
# lines from "root" up to "nobody" (i.e. system accounts)
sed -n -e '/^root/,/^nobody/p' /etc/passwd

# remove first column from ':'-separated file
sed -e 's/[^:]*://' datafile

# reverse the order of the first two columns
sed -e 's/\([^:]*\):\([^:]*\):\(.*\)$/\2:\1:\3/'
```

find: search for files

The **find** commands allows you to search for files based on specified properties (a filter for the file system)

- searches an entire directory tree, testing each file for the required property
- takes some action for all "matching" files (usually just print the file name)

How to use it:

find StartDirectory Tests Actions

where:

- the *Tests* examine file properties like name, type, modification date
- the *Actions* can be simply to print the name or execute an arbitrary command on the matched file

```
# find all the HTML files below /home/jas/web
find /home/jas/web -name '*.html' -print

# find all your files/dirs changed in the last 2 days
find ~ -mtime -2 -print

# show info on files changed in the last 2 days
find ~ -mtime -2 -type f -exec ls -l {} \;

# show info on directories changed in the last week
find ~ -mtime -7 -type d -exec ls -ld {} \;

# find directories either new or '07' in their name
find ~ -type d \( -name '*07*' -o -mtime -1 \) -print

# find all {\it{new}} HTML files below /home/jas/web
find /home/jas/web -name '*.html' -mtime -1 -print
```



```
# find background colours in my HTML files
find ~/web -name '*.html' -exec grep -H 'bgcolor' {} \;

# above could also be accomplished via ...
grep -r 'bgcolor' ~/web

# make sure that all HTML files are accessible
find ~/web -name '*.html' -exec chmod 644 {} \;

# remove any really old files ... Danger!
find /hot/new/stuff -type f -mtime +364 -exec rm {} \;
find /hot/new/stuff -type f -mtime +364 -ok rm {} \;
```

Filter Summary by Type

- **Horizontal slicing** - select subset of lines: cat, head, tail, *grep, sed, uniq
- **Vertical slicing** - select subset of columns: cut, sed
- **Substitution**: tr, sed
- **Aggregation, simple statistics**: wc, uniq
- **Assembly** - combining data sources: paste, join
- **Reordering**: sort
- **Viewing** (always end of pipeline): more, less
- **File system filter**: find
- **Programmable filters**: sed, (and perl)

Shell

Friday, 7 June 2019 7:29 PM

A **command interpreter** is a program that executes another program. Its aim is to allow users to execute the commands provided on a computer system.

Command interpreters come in two flavours:

- Graphical (e.g. Windows or Mac desktop) - they are easy for naïve users to start using the system
- Command-line (e.g. Unix shell) - they are programmable, powerful tools for expert users

On Unix/Linux, bash has become the de facto standard shell.

What Shells Do

All Unix shells have the same basic mode of operation:

```
loop
  if (interactive) print a prompt
  read a line of user input
  apply transformations to line
  split line into words (/s+/)
  use first word in line as command name
  execute that command,
    using other words as arguments
end loop
```

Note that "line of user input" could be a line from a file. In that case, the shell is reading a *script* of commands and acting as a kind of programming language interpreter.

The "transformations" applied to input lines include:

- variable expansion ... e.g. \$1 \${x-20}
- file name expansion ... e.g. *.c enr.07s?

To "execute that command" the shell needs to:

- find file containing named program (PATH)
- start new process for execution of program

Command Search PATH

If we have a script called bling in the current directory, we might be able to execute it with any of these:

| | |
|-------------|--|
| \$ sh bling | # <i>file need not be executable</i> |
| \$./bling | # <i>file must be executable</i> |
| \$ bling | # <i>file must be executable and . in PATH</i> |

Shell searches for programs to run using the colon-separated list of directories in the variable PATH. Beware: only append the current directory to end of your path, e.g:

```
$ PATH=.:$PATH
$ cat >cat <<eof
#!/bin/sh
echo miaou
eof
$ chmod 755 cat
$ cat /home/cs2041/public_html/index.html
miaou
$
```

Note ./cat is being run rather /bin/cat

Much harder to discover if it happens with another shell script which runs cat.

Safer still: **DO NOT** put . in your PATH

Shell as an Interpreter

The shell can be viewed as a programming language interpreter. As with all interpreters, the shell has:

- A state - a collection of variables and their values
- Control - its current location, execution flow

The shell differs from most interpreters since it:

- Modifies the program code before finally executing it
- Has an infinitely extendible set of basic operations

Basic operations in shell scripts are a sequence of *words*

CommandName Arg1 Arg2 Arg3 ...

A *word* is defined to be a sequence of:

- Non-whitespace characters (e.g. x, y1, aVeryLongWord)
- Characters enclosed in double-quotes (e.g. "abc", "a b c")
- Characters enclosed in single-quotes (e.g. 'abc', 'a b c')

Note that the types of quotes differ

Shell Scripts

Consider a file called "hello" containing

```
#!/bin/sh
echo Hello, World
```

How do we execute it?

```
$ sh hello      # execute the script
```

Or

```
$ chmod +x hello # make the file executable
$ ./hello        # execute the script
```

The next simplest shell program is: "Hello, YourName"

```
#!/bin/sh
echo -n "Enter your name: "
read name
echo Hello, $name
```

Shell variables:

```
$ read x      # read a value into variable x
$ y=John      # assign a value to variable y
$ echo $x     # display the VALUE of variable x
$ z="$y $y"   # assign two copies of y to variable z
```

Note: spaces matter - do not put spaces around the = symbol

More on shell variables:

- Do not need to declare variables; simply use them
- Are local to the current execution of the shell
- All variables have type string
- Initial value of variable is an *empty string*
- Note that x=1 is equivalent to x="1"

Examples:

```
$ x=5
$ y="6"
$ z=abc
$ echo $(( $x + $y ))
11
$ echo $(( $x + $z ))
5
$ x=1
$ y=fred
```

```

$ echo $x$y
1fred
$ echo $xy # the aim is to display "1y"
$ echo "$x"y
1y
$ echo ${x}y
1y
$ echo ${j-10} # give value of j or 10 if no value
10
$ echo ${j=33} # set j to 33 if no value (and give $j)
33
$ echo ${x:?No Value} # display "No Value" if $x not set
1
$ echo ${xx:?No Value} # display "No Value" if $xx not set
-bash: xx: No Value

```

Some shell built-in variables with pre-assigned values are:

| | |
|------|--|
| \$0 | The name of the command |
| \$1 | The first command-line argument |
| \$2 | The second command-line argument |
| \$3 | The third command-line argument |
| \$# | Count the command-line argument |
| \$* | All of the command-line arguments (together) |
| \$@ | All of the command-line arguments (separately) |
| \$? | Exit status of the most recent command |
| \$\$ | Process ID of this shell |

The last one is useful for generating unique filenames

If a string contains * ? [] it is matched against pathnames:

- * matches zero or more of any character
- ? matches any one character
- [] matches any one character between the []

The string is replaced with all matching pathnames. If no matches are found the string is left unchanged (usually configurable)

To numerically add variables:

| Input: | Output: |
|---|---------|
| #!/bin/sh x=1 y=2 echo \$((x + y)) | 3 |

Tip: debugging for shell scripts

- The shell transforms commands before executing
- Can be useful to know what commands are executed
- Can be useful to know what transformations are produced
- Set -x (for execution trace) to show each command after transformation

Quoting

Quoting can be used for three purposes in the shell:

- to group a sequence of words into a single "word"
- to control the kinds of transformations that are performed
- to capture the output of commands (back-quotes)

The three different kinds of quotes have three different effects:

| | |
|------------------|--|
| single-quote (') | grouping, turns off all transformations |
| double-quote (") | grouping, no transformations except \$ and ' |
| backquote (`) | no grouping, capture command results |

Single-quotes are useful to pass shell meta-characters in args: e.g. `grep 'S.*[0-9]+$' < myfile`

Use **double-quotes** to:

- construct strings using the values of shell variables
e.g. `"x=$x, y=$y"` like Java's `("x=" + x + ", y=" + y)`
- prevent empty variables from "vanishing"
e.g. use `test "$x" = "abc"` rather than `test $x = "abc"` in case `$x` is empty
- for values obtained from the command line or a user
e.g. use `test -f "$1"` rather than `test -f $1` in case `$1` contains a path with spaces e.g. `C:/Program Files/app/data`

In general you put your variables in double-quotes for safety :)

Back-quotes capture the output of commands as shell values. For ``Command``, the shell:

1. Performs variable substitution (as for double-quotes)
2. Executes the resulting command and arguments
3. Captures the standard output from the command
4. Converts it to a single string
5. Uses this string as the value of the expression

You can use `$(expr arg1 arg2 ...)` instead of backquotes

Example: convert GIF files to PNG format.

Original and converted files share the same prefix (e.g. `/x/y/abc.gif` is converted to `/x/y/abc.png`)

```
#!/bin/sh
# ungif - convert gifs to PNG format
for f in "$@"
do
    dir=$(dirname "$f")
    prefix=$(basename "$f" .gif)
    outfile="$dir/$prefix.png"
    giftopnm "$f" | pnmtopng > "$outfile"
done
```

| Stream | File descriptor |
|--------|-----------------|
| stdin | 0 |
| stdout | 1 |
| stderr | 2 |

`2>&1` means you print whatever goes to stderr to stdout

Connecting Commands

The shell provides *I/O redirection* to allow us to change where processes read from and write to.

| | |
|---------------------------------------|------------------------------------|
| <code>< infile</code> | connect stdin to the file infile |
| <code>> outfile</code> | connect stdout to the file outfile |
| <code>» outfile</code> | append stdout to the file outfile |
| <code>2> outfile</code> | connect stderr to the file outfile |
| <code>2>&1 > outfile</code> | connect stderr+stdout to outfile |

Beware: `>` truncates file before executing command.

Always have backups!

Many commands accept lists of input files:

e.g. `cat file1 file2 file3`

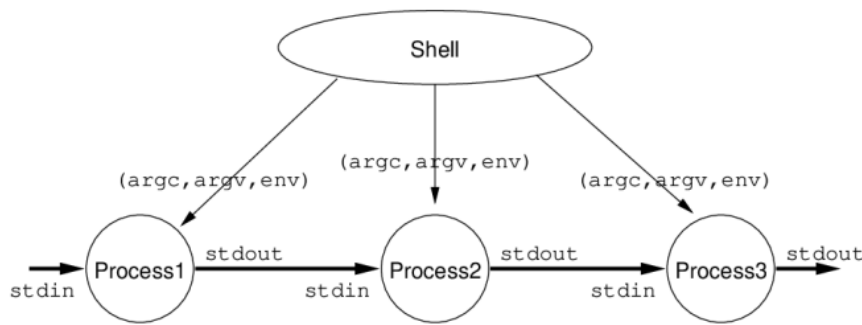
These commands also typically adopt conventions:

- Read contents of stdin if there are no filename arguments
- Treat the filename as meaning stdin

E.g. `cat -n < file` and `cat a - b - c`

If a command does not allow the use of multiple files as input, use:
E.g. `cat file1 file2 file3 | Command`

The shell sets up the environment for each command in a pipeline and connects them together.



Exit Status and Control

Process exit status is used for control in shell scripts:

- **zero exit status** means command *successful* → **true**
- **non-zero exit status** means error occurred → **false**

Mostly, exit status is simply ignored (e.g. when interactive)

One application of exit statuses:

- AND lists - `cmd1 && cmd2 && ... && cmdn`
(`cmdi+1` is executed only if `cmdi` succeeds (zero exit status))
- OR lists - `cmd1 || cmd2 || ... || cmdn`
(`cmdi+1` is executed only if `cmdi` fails (non-zero exit status))

Testing

The test command performs a test or combination of tests and:

- returns a zero exit status if the test succeeds
- returns a non-zero exit status if the test fails

It provides a variety of useful testing features:

- string comparison (`=`, `!=`)
- numeric comparison (`-eq` `-ne` `-lt` `-gt` `-le` `-ge`)
- checks on files (`-f` `-x` `-r`)
- boolean operators (`-a` `-o` `!`)

Examples:

```
# does the variable msg have the value "Hello"?
test "$msg" = "Hello"

# does x contain a numeric value larger than y?
test "$x" -gt "$y"

# Error: expands to "test hello there = Hello"?
msg="hello there"
test $msg = Hello

# is the value of x in range 10..20?
test "$x" -ge 10 -a "$x" -le 20

# is the file xyz a readable directory?
test -r xyz -a -d xyz

# alternative syntax; requires closing ]
[ -r xyz -a -d xyz ]
```

Note: use of quotes, spaces around values/operators

[is an alias for the **test** command

Sequential Execution

Sequential execution combines commands in pipelines and AND and OR lists.

Commands are executed sequentially if they are separated by semicolon or newline.

```
cmd1 ; cmd2 ; ... ; cmdn
```

```
cmd1
```

```
cmd2
```

```
...
```

```
cmdn
```

Grouping

Commands can be grouped using (...) or { ... }

- (cmd₁ ; ... cmd_n) are executed in a new sub-shell.
- { cmd₁ ; ... cmd_n } are executed in the current shell.

The exit status of a group is the exit status of last command.

Beware: state of sub-shell (e.g. \$PWD, other variables) is lost after (...), hence

```
$ cd /usr/share
$ x=123
$ ( cd $HOME; x=abc; )
$ echo $PWD $x
/usr/share 123
$ { cd $HOME; x=abc; }
$ echo $PWD $x
/home/cs2041 abc
```

If Command

The if-then-else construct allows conditional execution:

```
if testList{1}
then
    commandList{1}
elif testList{2}
then
    commandList{2}
    ...
else
    commandList{n}
fi
```

Keywords **if**, **else** etc, are only recognised at the start of a command (after newline or semicolon)

Examples:

```
# Check whether a file is readable
if [ -r $HOME ] # neater than: if test -r $HOME
then
    echo "$0: $HOME is readable"
fi

# Test whether a user exists in passwd file
if grep "^$user" /etc/passwd > /dev/null
then
    # do something if they do exist ...
else
    echo "$0: $user does not exist"
fi
```

Case Command

case provides multi-way choice based on patterns:

```
case word in
pattern{1}) commandList{1} ;;
```

```

pattern{2}-2) commandList{2}-2 ;;
...
pattern{n}) commandList{n} ;;
esac

```

The *word* is compared to each *pattern_i* in turn.

For the first matching pattern, corresponding *commandList_i* is executed and the statement finishes.

Patterns are those used in filename expansion (* ? []).

Examples:

```

# Checking number of command line args
case $# in
0) echo "You forgot to supply the argument" ;;
1) ... process the argument ... ;;
*) echo "You supplied too many arguments" ;;
esac
# Classifying a file via its name
case "$file" in
*.c) echo "$file looks like a C source-code file" ;;
*.h) echo "$file looks like a C header file" ;;
*.o) echo "$file looks like a an object file" ;;
...
?)  echo "$file's name is too short to classify" ;;
*)  echo "I have no idea what $file is" ;;
esac

```

Loop Commands

while loops iterate based on a test command list:

```

while testList
do
    commandList
done

```

for loops set a variable to successive words from a list:

```

for var in wordList
do
    commandList # ... generally involving var
done

```

Examples of while:

```

# Check the system status every ten minutes
while true
do
    uptime ; sleep 600
done

# Interactively prompt the user to process files
echo -n "Next file: "
while read filename
do
    process < "$filename" >> results
    echo -n "Next file: "
done

```

Examples of for:

```

# Compute sum of a list of numbers from command line
sum=0
for n in "$@" # use "$@" to preserve args

```



```

do
    sum='expr $sum + "$n"'
done

# Process files in $PWD, asking for confirmation
for file in *
do
    echo -n "Process $file? "
    read answer
    case "$answer" in
        [yY]*) process < $file >> results ;;
        *) ;;
    esac
done

```

Environment Variables

Environment variables are passed into a program, but are usually ignored.

If you want a variable to be used out of the shell script you need to export it. Through this answer can be used on any variable that you run.

```

$ answer=42
$ export answer

```

If you want your script to robust, you need to unset ALL environment variables and then set all the variables you need

LD_LIBRARY_PATH is the variable that sets where you look for libraries for your program
You can change the libraries associated with your programming.

Useful shell commands

| | |
|----------|--|
| rsync | A way of doing file transfers efficiently |
| cron | Is a unix command that makes programs run at a specific time |
| strace | Prints a list of system calls |
| Time | Time how long a program takes to run user - time to execute user code system - elapsed - |
| expr | |
| curl | Too to transfer data from or to a server using one of its supported protocols. Options: -s silent curl http://localhost/ |
| wget | Tool to download files from the web. Supports HTTP, HTTPS, and FTP, as well as retrieval through HTTP proxies Options: -o log all messages to logfile -q turn off output wget -O- http://localhost/ |
| uuidgen | Creates and prints a universally unique identifier (UUID) |
| dev/null | is a black hole. It's a special file on linux, where you can write any number of bytes to it and it will have no effect. If you read from it you will get zero bytes |

Perl Intro

Wednesday, 19 June 2019 7:41 PM

Practical Extraction and Report Language (PERL) is an interpreted language developed by Larry Wall in the late 80s. It has grown to become a replacement for awk, sed, grep, other filters, shell scripts, C programs and others. Perl has been influential for other languages; PHP, Python, Ruby.

Some of the language design principle for Perl:

- Make it easy/concise to express common idioms
- Provide many different ways to express the same thing
- Use defaults wherever possible
- Exploit powerful concise syntax & accept ambiguity/obscurity in some cases
- Create a large language that users will learn subsets of

Many of these conflict with design principles of languages for teaching.

The end result is:

- A language which makes it easy to build useful systems
- Readability can sometimes be a problem (because the language is too rich?)
- Being an interpreted language means slow/high power consumption (although it is remarkably efficient)
- Modest footprint - can be used embedded - but not ideal

In general, Perl is easy to write, concise, powerful and obscure programs.

We will only cover a subset of **Perl 5**, the first stable widely used version of Perl. It has a huge number of software libraries available. [CPAN](#) has over 60 000.

Running Perl

Perl programs can be invoked in several ways ...

- giving the filename of the Perl program as a command line argument:

```
perl PerlCodeFile.pl
```

- giving the Perl program itself as a command line argument:

```
perl -e 'print "Hello, world\n";'
```

- using the #! notation and making the program file executable:

```
chmod 755 PerlCodeFile
./PerlCodeFile
```

Advisable to *always* use -w option.

This causes Perl to print warnings about common errors.

```
perl -w PerlCodeFile.pl
perl -w -e 'PerlCode'
```

You can use options with #!

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

```
# Some Perl code
```

You can also get warnings via a pragma:

```
use warnings;
```

To catch other possible problems. Some programmers always use strict, others find it too annoying.

```
use strict;
```

Syntax Conventions

Perl uses non-alphabetic characters to introduce various kinds of program entities (i.e. set a context in which to interpret identifiers)

| Char | Kind | Example | Description |
|------|------------|-----------|--|
| # | Comment | # comment | Rest of the line is a comment |
| \$ | Scalar | \$count | Variable containing a simple value |
| @ | Array | @counts | List of values, indexed by <i>integers</i> |
| % | Hash | %marks | Set of values indexed by <i>strings</i> |
| & | Subroutine | %doIt | Callable Perl code (& optional) |

Any unadorned identifiers are either

- Names of built-in (or other) functions (e.g. `chomp`, `split`)
- Control-structures (e.g. `if`, `for`, `foreach`)
- Literal strings (like the shell)

The latter can be confusing to C/Java/PHP programmers

`$x = abc;` is the same as `$x = "abc";`

Variables

Perl provides these basic kinds of variables:

- **Scalars** - a single atomic value (number or string)
- **Arrays** - a list of values, indexed by a number
- **Hashes** - a group of values, indexed by a string

Variables do not need to be declared or initialised. If it is not initialised, a scalar is the empty string (0 in a numeric context)

Beware: of spelling mistakes in variable names. e.g:

`print "abc=$acb\n";` rather than `print "abc=$abc\n";`

Use warnings (-w) and easy to spell variable names

Many scalar operations have a "default source/target". If you don't specify an argument, variable `$_` is assumed. This makes it

- Often very convenient for writing brief programs
- Sometimes confusing to new users

`$_` performs a similar role to "it" in English. E.g. "The dog ran away. It ate a bone. It had lots of fun."

Arithmetic & Logical Operators

Perl arithmetic and logical operators are similar to C.

Numeric: `==` `!=` `<` `<=` `>` `>=` `<=>`

String: `eq` `ne` `lt` `le` `gt` `ge` `cmp`

Most C operators are present and have similar meanings, e.g:

`+` `-` `*` `/` `%` `++` `-` `+=`

Perl string concatenation operator `.` is equivalent to using C's `malloc` and `strcat`

C `strcmp` equivalent to Perl `cmp`

Scalars

Examples:

| | |
|--------------------------------|---|
| <code>\$x = '123';</code> | <code># \$x assigned string "123"</code> |
| <code>\$y = "123 ";</code> | <code># \$y assigned string "123 "</code> |
| <code>\$z = 123;</code> | <code># \$z assigned integer 123</code> |
| <code>\$i = \$x + 1;</code> | <code># \$x value converted to integer</code> |
| <code>\$j = \$y + \$z;</code> | <code># \$y value converted to integer</code> |
| <code>\$a = \$x == \$y;</code> | <code># numeric compare \$x,\$y (true)</code> |
| <code>\$b = \$x eq \$y;</code> | <code># string compare \$x,\$y (false)</code> |
| <code>\$c = \$x.\$y;</code> | <code># concat \$x,\$y (explicit)</code> |
| <code>\$c = "\$x\$y";</code> | <code># concat \$x,\$y (interpolation)</code> |

Note: `$c = $x $y` is invalid (Perl has no empty infix operator unlike predecessor languages such as `awk`, where `$x $y` meant string concatenation)

A very common pattern for modifying scalars is:

```
$var = $var op expression
```

Compound assignments for the most common operators allow you to write

```
$var op= expression
```

Examples:

```
$x += 1;    # increment the value of $x
$y *= 2;    # double the value of $y
$a .= "abc" # append "abc" to $a
```

Perl Truth Values

- False: "" (empty string) and '0'
- True: everything else.

Be careful, subtle consequences:

- False: 0.0, 0x0
- True: '0.0' and "0\n"

Logical Operators

Perl has two sets of logical operators, one like C, the other like "English".

The second set has very low precedence, so can be used between statements.

| Operation | Example | Meaning |
|-----------|-----------------------------|--|
| And | <code>x && y</code> | false if x is false, otherwise y |
| Or | <code>x y</code> | true if x is true, otherwise y |
| Not | <code>! x</code> | true if x is not true, false otherwise |
| And | <code>x and y</code> | false if x is false, otherwise y |
| Or | <code>x or y</code> | true if x is true, otherwise y |
| Not | <code>not x</code> | true if x is not true, false otherwise |

The lower precedence of or/and enables common Perl idioms.

```
if (!open FILE, '<', "a.txt") {
    die "Can't open a.txt: $!";
}
```

is often replaced by Perl idiom

```
open FILE, '<', "a" or die "Can't open a: $!";
```

Note this doesn't work:

```
open FILE, '<', "a" || die "Can't open a: $!";
```

because its equivalent to:

```
open FILE, '<', ("a" || die "Can't open a: $!");
```

Stream Handles

Input & output are accessed via *handles* - similar to FILE * in C.

```
$line = <IN>; # read next line from stream IN
```

Output file handles can be used as the first argument to print:

```
print OUT "Andrew\n"; # write line to stream OUT
```

Note: no comma after the handle

Predefined handles for stdin, stdout, stderr

```
# STDOUT is default for print so can be omitted
print STDOUT "Enter a number: ";
$number = <STDIN>;
if (number < 0) {
    print STDERR "bad number\n";
}
```

Files

When opening files, handles can be explicitly attached to files via the open command:

```
open DATA, '<', 'data'; # read from file data
open RES, '>', 'results'; # write to file results
open XTRA, '>>', 'stuff'; # append to file stuff
```

Handles can even be attached to pipelines to read/write to Unix commands:

```
open DATE, "date|"; # read output of date command
open FEED, "|more"); # send output through "more"
```

Opening a file may fail - always check:

```
open DATA, '<', 'data' or die "Can't open data: $!";
```

Example: reading and writing a file

```
open OUT, '>', 'a.txt' or die "Can't open a.txt: $!";
print OUT "42\n";
close OUT;
open IN, '<', 'a.txt' or die "Can't open a.txt: $!";
$answer = <IN>;
close IN;
print "$answer\n"; # prints 42
```

If you supply a uninitialized variable Perl will store an anonymous file handle in it:

```
open my $output, '>', 'answer' or die "Can't open ...";
print $output "42\n";
close $output;
open my $input, '<', 'answer' or die "Can't open ...";
$answer = <$input>;
close $input;
print "$answer\n"; # prints 42
```

Use this approach for larger programs to avoid collision between file handle names.

Close

Handles can be explicitly closed with `close(HandleName)`

- All handles closed on exit.
- Handles also closed if open is done with the same handle name (which is good for lazy coders)
- Data on output streams may be not written (buffered) until close - hence close ASAP

Calling `<>` without a file handle gets unix-filter behaviour.

- It treats all command-line arguments as file names
- It opens and reads from each of them in turn
- If there are no command line arguments, then `<> == <STDIN>`

So this is cat in Perl:

```
#!/usr/bin/perl
# Copy stdin to stdout
while ($line = <>) {
    print $line;
}
```

Control Structures

All single Perl statements must be terminated by a semicolon, e.g.

```
$x = 1;
print "Hello";
```

All statements with control structures must be enclosed in braces, e.g.

```
if ($x > 9999) {
```

```
    print "x is big\n";
}
```

You don't need a semicolon after a statement group in {...}.
Statement blocks can also be used like anonymous functions.

Selection is handled by if ... elsif ... else

```
if ( boolExpr{1} ) {
    statements{1};
} elsif ( boolExpr{2} ) {
    statements{2};
}
...
else { statements{n} }
statement if ( expression );
```

Iteration is handles by while, until, for, foreach. The equivalent to break is last.

```
while ( boolExpr ) {
    statements
}
until ( boolExpr ) {
    statements
}
for ( init ; boolExpr ; step ) {
    statements
}
foreach var (list) {
    statements
}
```

Example: computing $\text{pow} = k^n$

```
# Method 1 ... while
$pow = $i = 1;
while ($i <= $n) {
    $pow = $pow * $k;
    $i++;
}

# Method 2 ... For
$pow = 1;
for ($i = 1; $i <= $n; $i++) {
    $pow *= $k;
}

# Method 3 ... Foreach
$pow = 1;
foreach $i (1..$n) {
    $pow *= $k;
}

# Method 4 ... foreach $_
$pow = 1;
foreach (1..$n) { $pow *= $k; }

# Method 5 ... builtin operator
$pow = $k ** $n;
```

Function Calls

All Perl function calls:

- are call by value (like C) (except scalars aliased to @_)
- are expressions (although often ignore return value)

Notation(s) for Perl function calls:

```
&func(arg{1}, arg{2}, ... arg{n})  
func(arg{1}, arg{2}, ... arg{n})  
func arg{1}, arg{2}, ... arg {n}
```

Terminating

For normal termination, call: `exit 0`

The `die` function is used for abnormal termination:

1. Accepts a list of arguments
2. Concatenates them all into a single string
3. Appends the file name and line number
4. Prints this string
5. Terminates the Perl interpreter

Example:

```
if (! -r "myFile") {  
    die "Can't read myFile: $!\n";  
}  
# or  
die "Can't read myFile: $!\n" if ! -r "myFile";  
# or  
-r "myFile" or die "Can't read myFile: $!\n"
```

Perl and External Commands

Perl is shell-like in the ease of invoking other commands/programs.

Several ways of interacting with external commands/programs:

| | |
|-----------------------------|---|
| <code>'cmd';</code> | capture entire output of <code>cmd</code> as single string |
| <code>system "cmd"</code> | execute <code>cmd</code> via the shell and capture its exit status only |
| <code>open F, "cmd "</code> | collect <code>cmd</code> output by reading from a stream |

External command examples:

```
$files = 'ls $d'; # output captured  
  
$exit_status = system "ls $d"; # output to stdout  
  
open my $files, '-|', "ls $d"; # output to stream  
while (<$files>) {  
    chomp;  
    @fields = split; # split words in $_ to @_  
    print "Next file is $fields[$#fields]\n";  
}
```

File Test Operators

Perl provides an extensive set of operators to query file information:

| Test | File is... |
|-----------------|------------|
| <code>-r</code> | readable |
| <code>-w</code> | writable |
| <code>-x</code> | executable |
| <code>-e</code> | exists |

| | |
|----|-------------------|
| -z | Has zero size |
| -s | Has non-zero size |
| -f | Plain file |
| -d | directory |
| -l | Symbolic link |

Used in checking I/O operations, e.g. `-r "dataFile" && open my $data, '<', "dataFile";`

Special Variables

Perl defines numerous special variables to hold information about its execution environment.

These variables typically have names consisting of a single punctuation character e.g. `$!` `$@` `$#` `$$` `$%` ... (English names are also available)

The `$_` variable is particularly important:

- acts as the default location to assign result values (e.g. `<STDIN>`)
- acts as the default argument to many operations (e.g. `print`)

Careful use of `$_` can make programs concise, uncluttered, but careless use of `$_` can make programs cryptic

| | |
|--------------------|---|
| <code>\$_</code> | default input and pattern match |
| <code>@ARGV</code> | list (array) of command line arguments |
| <code>\$0</code> | name of file containing executing Perl script (cf. shell) |
| <code>\$i</code> | matching string for <i>i</i> th regexp in pattern |
| <code>\$!</code> | last error from system call such as <code>open</code> |
| <code>\$.</code> | line number for input file stream |
| <code>\$/</code> | line separator, none if undefined |
| <code>\$\$</code> | process number of executing Perl script (cf. shell) |
| <code>%ENV</code> | lookup table of environment variables |

Example (echo in Perl):

```
for ($i = 0; $i < @ARGV; $i++) {
    print "$ARGV[$i] ";
}
print "\n";
```

or

```
foreach $arg (@ARGV) {
    print "$arg ";
}
print "\n";
```

or

```
print "@ARGV\n";
```

- ALWAYS USE PERL'S `-w` FLAG: `#!/usr/bin/perl -w`
- `$` for scalar values
- `print` is a built-in function so we don't need parentheses around it
- Single quotes print out exact what is inside
- Perl looks inside the double quotes, replaces whatever looks like a variable with its value
- Back quotes lets perl run shell commands
- Two angle brackets `==` function call
- `chomp $x` removes a new line char at the end of `x`
- Perl detect errors by seeing that a variable was only used once
- Perl arrays grow as needed. You can't buffer overflow in perl
- `$line =~ /Hermione/` checks if the line has Hermione in it like `grep`

- Split takes regular expression and string `split /|/, $line`
- Join is split's friend `join("Andrew", @fields)` joins Andrew with array fields

Perl Arrays

Wednesday, 19 June 2019 7:41 PM

Array (Lists)

An **array** is a sequence of scalars, indexed by position (0,1,2,...).

The whole array is denoted by `@array`

Individual array elements are denoted by `$array[index]`

`$#array` gives the *index of the last element*.

Example:

```
$a[0] = "first string";
$a[1] = "2nd string";
$a[2] = 123;

# or, equivalently,
@a = ("first string", "2nd string", 123);
print "Index of last element is $#a\n";
print "Number of elements is ", $#a+1, "\n";

@a = ("abc", 123, 'x');

# numeric context ... gives list length
$n = @a; # $n == 3

# string context ... gives space-separated elems
$s = "@a"; # $s eq "abc 123 x"

# scalar context ... gives list length
$t = @a; # $t eq "3"

# print context ... gives joined elems
print @a; # displays "abc123x"
```

In Perl, interpretation is context-dependent.

Arrays do not need to be declared, and they grow and shrink as needed.

"Missing" elements are interpolated, e.g.

```
$abc[0] = "abc"; $abc[2] = "xyz";
# reference to $abc[1] returns ""
```

You can assign to a whole array; can assign from a whole array, e.g.

```
@numbers = (4, 12, 5, 7, 2, 9);
($a, $b, $c, $d) = @numbers;
```

Since assignment of list elements happens in parallel ...

```
($x, $y) = ($y, $x); # swaps values of $x, $y
```

Array slices, e.g.

```
@list = (1, 3, 5, 7, 9);
print "@list[0,2]\n"; # displays "1 5"
print "@list[0..2]\n"; # displays "1 3 5"
print "@list[4,2,3]\n"; # displays "9 5 7"
print "@list[0..9]\n"; # displays "1 3 5 7 9"
```

Array values interpolated into array literals:

```
@a = (3, 5, 7);
@b = @a; # @b = (3,5,7);
@c = (1, @a, 9); # @c = (1,3,5,7,9);
```

```
@a == (@a) == ((@a)) ...
```

Arrays can be accessed element-at-a-time using the for loop:

```
@nums = (23, 95, 33, 42, 17, 87);
$sum = 0;
for ($i = 0; $i < @nums; $i++) { # @nums gives length,
    $sum += $nums[$i];           # $#nums gives the last index of the array
}

$sum = 0;
foreach $num (@nums) {
    sum += $num; # $num is the actual array element and can be changed
}

For $i (0..$#nums) {
    sum += $nums[$i];
}
```

push and pop act on the "right-hand" end of an array:

```
# Value of @a
@a = (1,3,5); # (1,3,5)
push @a, 7; # (1,3,5,7)
$x = pop @a; # (1,3,5,7), $x == 7
$y = pop @a; # (1,3,5), $y == 5
```

Other useful operations on arrays:

| | |
|------------------|---------------------------------------|
| @b = sort(@a) | returns sorted version of @a |
| @b = reverse(@a) | returns reversed version of @a |
| shift(@a) | like pop(@a), but from left-hand end |
| unshift(@a,x) | like push(@a,x), but at left-hand end |

Lists as Strings

Recall the marks example from earlier on; we used "54,67,88" to effectively hold a list of marks.

We can turn this into a real list using the *split* operation.

Syntax: `split(/pattern/,string)` returns a list

The *join* operation allows us to convert from **list to string**:

Syntax: `join(string_pattern,List)` returns a string

(Don't confuse this with the join filter in the shell. Perl's join acts more like paste.)

```
$marks = "99,67,85,48,77,84";

@listOfMarks = split(/,/,$marks);
# assigns (99,67,85,48,77,84) to @listOfMarks

$sum = 0;
foreach $m (@listOfMarks) {
    $sum += $m;
}

$newMarks = join(':',@listOfMarks);
# assigns "99:67:85:48:77:84" to $newMarks
```

Complex splits can be achieved by using a full regular expression rather than a single delimiter character. If part of the regexp is parenthesised, the corresponding part of each delimiter is retained in the resulting list.

```
split(/[#@]+/, 'ab##@#c#d@@e'); #gives (ab,c,d,e)
split(/([#@]+)/, 'ab##@#c#d@@e'); #gives (ab,##@#,c,#,d,@,@,
split(/([#@])+/, 'ab##@#c#d@@e'); #gives (ab,#,c,#,d,@,e)
```

And as a specially useful case, the empty regexp is treated as if it matched between every character, splitting the string into a list of single characters:

```
split(/, 'hello'); # gives (h, e, l, l, o)
```

Associative Arrays (Hashes)

As well as arrays indexed by numbers, Perl supports arrays indexed by strings: **hashes**.

Conceptually, as hash is a set (not list) of (key, value) pairs.

We can deal with an entire hash at a time via %hashName, e.g.

```
# Key Value
%days = ( "Sun" => "Sunday",
          "Mon" => "Monday",
          "Tue" => "Tuesday",
          "Wed" => "Wednesday",
          "Thu" => "Thursday",
          "Fri" => "Friday",
          "Sat" => "Saturday" );
```

Individual components of a hash are accessed via \$hashName{keyString}

Examples:

```
$days{"Sun"} # returns "Sunday"
$days{"Fri"} # returns "Friday"
$days{"dog"} # is undefined (interpreted as "")
$days{0}     # is undefined (interpreted as "")

# inserts a new (key,value)
$days{dog} = "Dog Day Afternoon"; # bareword OK as key

# replaces value for key "Sun"
$days{"Sun"} = Soonday; # bareword OK as value
```

Consider the following two assignments:

```
@f = ("John", "blue", "Anne", "red", "Tim", "pink");
%g = ("John" => "blue", "Anne" => "red", "Tim" => "pink")
```

The first produces an array of strings that can be accessed via position, such as \$f[0]

The second produces a lookup table of names and colours, e.g. \$g{"Tim"}.

(In fact the symbols => and comma have identical meaning in a list, so either could have been used. However, **always** use the arrow form exclusively for hashes.)

Consider iterating over each of these data structures:

| | |
|--|--|
| <pre>foreach \$x (@f) { print "\$x\n"; }</pre> | <pre>foreach \$x (keys %g) { print "\$x => \$g{\$x}\n"; }</pre> |
| <pre>John blue Anne red Tim pink</pre> | <pre>Anne => red Tim => pink John => blue</pre> |

The data comes out of the hash in a fixed but arbitrary order (due to the hash function).

There are several ways to examine the (*key*, *value*) pairs in a hash:

```
foreach $key (keys %myHash) {  
    print "($key, $myHash{$key})\n";  
}
```

or, if you just want the values without the keys

```
foreach $val (values %myHash) {  
    print "(?, $val)\n";  
}
```

or, if you want them both together

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

Note that each method produces the keys/values in the same order. It's illegal to change the hash within these loops

Example (collecting marks for each student):

- a data file of (*name*, *mark*) pairs, space-separated, one per line
- out should be (*name*, *marksList*), with comma-separated marks

```
while (<>) {  
    chomp;                # remove newline  
    ($name, $mark) = split; # separate data fields  
    $marks{$name} .= ",$mark"; # accumulate marks  
}  
  
foreach $name (keys %marks) {  
    $marks{$name} =~ s/,//; # remove comma prefix  
    print "$name $marks{$name}\n";  
}
```

The delete function removes an entry (or entries) from an associative array.

To remove a single pair:

```
delete $days{"Mon"}; # "I don't like Mondays"
```

To remove multiple pairs:

```
delete @days{ ("Sat", "Sun") }; # Oh noes - no weekend!
```

To clean out the entire hash:

```
foreach $d (keys %days) { delete $days{$d}; }  
# or, more simply  
%days = ();
```

@lines = <STDIN> reads all lines in stdin and puts it in the array lines

A variable can have

- A list context - multiple values expected
- A scalar context - one value is expected

Reverse is a function that takes in a list,

Perl Regex

Thursday, 27 June 2019 11:16 AM

Because Perl makes heavy use of strings, regular expressions are a very important component of the language. They can be used:

- In conditional expressions to test whether a string matches a pattern
e.g. checking the contents of a string

```
if ($name =~ /[0-9]/) { print "name contains digit\n"; }
```

- In assignments to modify the value of a string
e.g. convert McDonald to MacDonald

```
$name =~ s/Mc/Mac/;
```

e.g. convert to upper case

```
$string =~ tr/a-z/A-Z/;
```

Perl extends POSIX regular expressions with some shorthand:

| | |
|-----------------|--|
| <code>\d</code> | Matches any digit, i.e. <code>[0-9]</code> |
| <code>\D</code> | Matches any non-digit, i.e. <code>[^0-9]</code> |
| <code>\w</code> | Matches any "word" char, i.e. <code>[a-zA-Z_0-9]</code> |
| <code>\W</code> | Matches any non "word" char, i.e. <code>[^a-zA-Z_0-9]</code> |
| <code>\s</code> | Matches any whitespace, i.e. <code>[\t\n\r\f]</code> |
| <code>\S</code> | Matches any non-whitespace, i.e. <code>[^ \t\n\r\f]</code> |

Perl also adds some new anchors to regexps:

| | |
|-----------------|-----------------------------------|
| <code>\b</code> | Matches at a word boundary |
| <code>\B</code> | Matches except at a word boundary |

And generalised the repetition operators:

| | |
|------------------------|--|
| <code>patt*</code> | Matches 0 or more occurrences of <i>patt</i> |
| <code>patt+</code> | Matches 1 or more occurrences of <i>patt</i> |
| <code>patt?</code> | Matches 0 or 1 occurrence of <i>patt</i> |
| <code>patt{n,m}</code> | Matches between <i>n</i> and <i>m</i> occurrences of <i>patt</i> |

The default semantics for pattern matching is "first, then largest".

E.g. `/ab+/` matches **abbb**abbbb not **abbb**abbbb or abbb**abbbb**

A pattern can also be qualified so that it looks for the shortest match.

If the repetition operator is followed by `?` the "first, then shortest" string that matches the pattern is chosen.

E.g. `/ab+?/` would match **abbb**abbbb

Regular expressions can be formed by interpolating strings in between `/.../`.

Example:

```
$pattern = "ab+";  
$replace = "Yod";  
$text = "abba";  
  
$text =~ s/$pattern/$replace/;  
  
# converts "abba" to "Yoda"
```

Note: Perl doesn't confuse the use of `$` in `$var` and `abc$`, because the anchor occurs at the end.

Using Matching Results

In a scalar context matching & substitute operators return how many times the match/substitute succeeded. This allows them to be used as the controlling expression in if/while statements.

For example:

```
print "Destroy the file system? "  
$answer = <STDIN>;  
if ($answer =~ /yes||ok|affirmative/i) {  
    system "rm -r /";  
}  
  
s/[aeiou]//g or die "no vowels to replace";
```

In a list context the matching operators returns a list of the matched strings.

For example:

```
$string = "-5==10zzz200_";  
@numbers = $string =~ /\d+/g;  
print join(",", @numbers), "\n";  
# prints 5,10,200
```

If the regex contains ()s only the captured text is returned

```
$string = "Bradley, Marion Zimmer";  
($family_name, $given_name) = $string =~ /([^\,]*), (\S+)/;  
print "$given_name $family_name\n";  
# prints Marion Bradley
```

Pattern Matcher

A Perl script to accept a pattern and a string and show the match (if any):

```
#!/usr/bin/perl  
$pattern = $ARGV[0];    print "pattern=/$pattern/\n";  
$string = $ARGV[1];    print "string =\"$string\"\n";  
$string =~ /$pattern/;  print "match =\"$&\"\n";
```

You might find this a useful tool to test out your understanding of regular expressions

Perl Functions

Tuesday, 2 July 2019 10:59 PM

Perl Library Functions

Perl has hundreds of functions for all kinds of purposes:

- File manipulation, database access, network programming, etc.

It has an especially rich collection of functions for strings.

E.g. `lc`, `uc`, `length`.

Consult on-line Perl manuals, reference books, example programs for further information.

Defining Functions

Perl functions (or subroutines) are defined via `sub`, e.g.

```
sub sayHello {  
    print "Hello!\n";  
}
```

And used by calling, with or without `&`, e.g.

```
&sayHello; # arg list optional with &  
sayHello(); # more common, show empty arg list explicitly
```

Function arguments are passed via a list variable `@_`, e.g.

```
sub mySub {  
    @args = @_;  
    print "I got ", @args+1, " args\n";  
    print "They are (@args)\n";  
}
```

Note that `@args` is a global variable. To make it local, precede by `my`, e.g.

```
my @args = @_;
```

We can achieve a similar effect to the C function by using array assignment in Perl.

| C code: | Perl code: |
|--|---|
| <pre>int f(int x, int y, int z) { int res; ... return res; }</pre> | <pre>sub f { my (\$x, \$y, \$z) = @_; my \$res; ... return \$res; }</pre> |

Lists (arrays and hashes) with any scalar arguments to produce a single argument list.

This means you can only pass a single array or hash to a Perl function and it must be the last argument.

```
sub good {  
    my ($x, $y, @list) = @_;  
}
```

This will not work (x and y will be undefined):

```
sub bad {  
    my (@list, $x, $y) = @_;  
}
```

And this will not work (list2 will be undefined):

```
sub bad {  
    my (@list1, @list2) = @_;  
}
```

References

References

- Are like C pointers (they refer to some other object)
- Can be assigned to scalar variables

- Are dereferenced by evaluating the variable

Example:

```
$aref = [1,2,3,4];
print @$aref;      # displays whole array
... $$aref[0];     # access the first element
... ${$aref}[1];   # access the second element
... $aref->[2];     # access the third element
```

Parameter Passing

Scalar variables are aliased to the corresponding elements of `@_`. This means a function can change them. E.g. this code sets `x` to 42.

```
sub assign {
    $_[0] = $_[1];
}
assign($x, 42);
```

Arrays and hashes are passed by value. If a function needs to change an array or hash, you can pass a reference. You can also use references if you need to pass multiple hashes or arrays.

```
%h = (jas=>100, arun=>95, eric=>50);
@x = (1..10);

mySub(3, \%h, \@x);
mysub(2, \%h, [1,2,3,4,5]);
mysub(5, {a=>1,b=>2}, [1,2,3]);
```

Notation:

- `[1, 2, 3]` gives a reference to `(1,2,3)`
- `{a=>1, b=>2}` gives a reference to `(a=>1, b=>2)`

Perl Prototypes

Perl prototypes declare the expected parameter structure for a function. Unlike other languages (e.g. C), a Perl prototype's main purpose is not type checking. A Perl prototype's main purpose is to allow more convenient calling of functions. You also get some error checking - sometimes useful, sometimes less so. Some programmers recommend against using prototypes. In this course prototype usage is optional. You can use prototypes to define functions that can be called like built-ins

Prototypes can cause a reference to be passed when an array is given as a parameter. If we define our version of `push` like this:

```
sub mypush {
    my ($array_ref, @elements) = @_;
    if (@elements) {
        @$array_ref = (@$array_ref, @elements);
    } else {
        @$array_ref = (@$array_ref, $_);
    }
}
```

It has to be called like this:

```
mypush(\@array, $x);
```

But if we add this prototype:

```
sub mypush2(\@@)
```

It can be called just like the builtin `push`:

```
mypush @array, $x;
```

Recursive Functions

Example:

```
sub fac {
```

```

my ($n) = @_ ;
return 1 if $n < 1;
return $n * fac($n-1);
}

```

which behaves as

```

print fac(3); # displays 6
print fac(4); # displays 24
print fac(10); # displays 3628800
print fac(20); # displays 2.43290200817664e+18

```

Eval

The Perl built-in function `eval` evaluates (executes) a supplied string as Perl. For example, this Perl will print 42:

```

$perl = '$answer = 6 * 7;';
eval $perl;
print "$answer\n";

```

and this Perl will print 55:

```

@numbers = 1..10;
$perl = join("+", @numbers);
print eval $perl, "\n";

```

and this Perl also prints 55:

```

$perl = '$sum=0; $i=1; while ($i <= 10) {$sum+=$i++}';
eval $perl;
print "$sum\n";

```

and this Perl could do anything:

```

$input_line = <STDIN>;
eval $input_line;

```

Pragma and Modules

Perl provides a way of controlling some aspects of the interpreter's behaviour (through *pragmas*) and is an extensible language through the use of compiled modules, of which there is a large number. Both are introduced by the *use* keyword.

- `use English`; allow names for built-in vars, e.g., `$NF = $.` and `$ARG = $_`.
- `use integer`; truncate all arithmetic operations to integer, effective to the end of the enclosing block.
- `use strict 'vars'`; insist on all variables declared using `my`

Standard Modules

These are several thousand standard Perl modules into packages. The package name is prefixed with `::`

Examples:

- `use DB_File`; functions for maintaining an external hash
- `use Getopt::Std`; functions for processing command-line flags
- `use File::Find`; find function like the shell's `find`
- `use Math::BigInt`; unlimited-precision arithmetic
- `use CGI`;

CPAN

Comprehensive Perl Archive Network (CPAN) is an archive of 150,000+ Perl packages. Hundreds of mirrors, including <http://mirror.cse.unsw.edu.au/pub/CPAN/>

Command line tools to quickly install packages from CPAN

Perl Modules

Saturday, 13 July 2019 11:56 PM

These are several thousand standard Perl modules available via **use** keyword. The module name is prefixed with ::

Examples:

- use DB_File; - functions for maintaining an external hash
- use Getopt::Std; - functions for processing command-line flags
- use [File::Find](#); - find function like the shell's find
- use Math::BigInt; - unlimited-precision arithmetic

Defining a Module

```
use base 'Exporter';
our @EXPORT = qw/min max/;
use List::Util qw/reduce/;

sub sum {
    return reduce {$a + $b} @_;
}

sub min {
    return reduce {$a < $b ? $a : $b} @_;
}
# must return true to indicate loading succeeded
1;
```

Using a Module

```
use Example_Module qw/max/;

# As max is in our import list
# it can be used without module name
print max(42,3,5), "\n";

# We don't import min explicitly
# so it needs the module name
print Example_Module::min(42,3,5), "\n";
```

The directory containing Example_Module.pm must be in listed environment variable **PERL5LIB**
PERL5LIB is colon separated list of directory equivalent to Shell **PATH**

Pragmas

Perl provides a way of controlling some aspects of the interpreter's behaviour (through *pragmas*) also introduced by the *use* keyword.

- use English; - allow names for built-in vars, e.g., \$NF = \$. and \$ARG = \$_.
- use integer; - truncate all arithmetic operations to integer, effective to the end of the enclosing block.
- use strict 'vars'; - insist on all variables declared using my.

Git

Thursday, 27 June 2019 11:15 AM

Version Control

A **version control system** allows software developers to:

- work simultaneously and cooperatively on a system
- document when, who, & why changes made
- discuss and approve changes
- recreate old versions of a system when needed
- multiple versions of system can be distributed, tested, merged

This allows change to be managed/controlled in a systematic way.

VCSs also try to minimise resource use in maintaining multiple versions.

Version Control Through the Ages

| | |
|---|---|
| 1970's - <i>SCCS</i> (source code control system) | <ul style="list-style-type: none">• first version control system• centralized VCS - single central repository• introduced idea of multiple versions via delta's• single user model: lock - modify - unlock• only one user working on a file at a time |
| 1980's - <i>RCS</i> (revision control system) | <ul style="list-style-type: none">• similar functionality to SCCS (essentially a clean open-source re-write of SCCS)• centralized VCS - single central repository• single user model: lock - modify - unlock• only one user working on a file at a time• still available and in use |
| 1990 - <i>CVS</i> (concurrent version system) | <ul style="list-style-type: none">• centralized VCS - single central repository• locked check-out replaced by copy-modify-merge model• users can work simultaneously and later merge changes• allows remote development essential for open source projects• web-accessible interface promoted wide-distributed projects• poor handling of file metadata, renames, links |
| Early 2000's - <i>Subversion</i> (svn) | <ul style="list-style-type: none">• depicted as "CVS done right"• many CVS weakness fixed• solid, well documented, widely used system but essentially the same model as CVS• centralized VCS - single central repository• svn is suitable for assignments/small-medium projects• easier to understand than distributed VCSs, well supported |
| Early 2000s - <i>Bitkeeper</i> | <ul style="list-style-type: none">• distributed VCS - multiple repositories, no "master"• every user has their own repository• written by Larry McVoy• Commercial system but allowed limited use for Linux kernel until dispute over licensing issues• Linus Torvalds + others then wrote GIT open source distributed VCS• Other open source distributed VCS's appeared, e.g: bazaar (Canonical!), darcs (Haskell!), Mercurial |
| Present - Git | <ul style="list-style-type: none">• distributed VCS - multiple repositories, no "master"• every user has their own repository• created by Linus Torvalds for Linux kernel• external revisions imported as new branches• flexible handling of branching• various auto-merging algorithms• Andrew recommends you use git unless good reason not to |

- not better than competitors but better supported/more widely used (e.g. github/bitbucket)
- at first stick with a small subset of commands
- substantial time investment to learn to use Git's full power

Many VCS use the notion of a **repository**. It stores all versions of all objects (files) managed by the VCS. A repository can be a single file, directory tree, database etc. It may possibly be accessed by a filesystem, http, ssh or a custom protocol. The repository may be structured as a collection of projects.

Git

Git uses the sub-directory `.git` to store the repository. Inside `.git` there are:

- **blobs** file contents identified by SHA-1 hash
- **tree objects** links blobs to info about directories, link, permissions (limited)
- **commit objects** links trees objects with info about parents, time, log message
- Create repository **git init**
- Copy existing repository **git clone**

The project must be in a single directory tree. Usually you don't want to track all files in the directory tree.

Do not track binaries, derived files, temporary files, large static files.

Use `.gitignore` files to indicate files you never want to track

Be careful: **git add directory** will every file in *file* and sub-directories

git commit

A git commit is a snapshot of all the files in the project. You can return the project to this state using **git checkout**

Beware if you accidentally add a file with confidential info to git, you'll need to remove it from all commits.

git add copies file to staging area for next commit

Use **git commit -a** if you want commit current versions of all files being tracked

Commits have parent commit(s), most have 1 parent

Merge produce commit with 2 parents, first commit has no parent

Merge commits labelled with SHA-1 hash

git branch

Git branch is pointer to a commit. It allows you to name a series of commits. Git branch provides convenient tracking of versions of parallel version of projects.

New features can be developed in a branch and eventually merged with other branches.

The default branch is called `master`.

HEAD is a reference to the last commit in the current branch

git branch name creates branch *name*

git checkout name changes all project files to their version on this branch

git merge

This merges branches. You can use **git mergetool** to show conflicts. You can also configure your own mergetool - many choices meld, kdiff3, p4merge etc. Kaleidoscope is a popular mergetool on OSX

git push

git push repository-name branch adds commits from your *branch* to remote repository *repository-name*

You can set defaults, e.g. **git push -u origin master** then run **git push**

git remote lets you give names to other repositories

Note **git clone** sets origin to be name for cloned repository

git fetch/pull

git fetch *repository-name branch* adds commits from *branch* in remote repository *repository-name*
Usually **git pull** - combines fetch and merge

Making a Git Repo Public via GitHub

GitHub popular repo hosting site (see competitors e.g. bitbucket). GitHub is free for small number of public repos. GitHub and competitors also let you setup collaborators, wiki, web pages, issue tracking.
Web access to git repo e.g. <https://github.com/mirrors/linux>

To publish your code to the world:

1. Create github account - assume you choose *2041rocks* as your login
2. Create a repository - assume you choose *my_code* for the repo name
3. Add your ssh key (.ssh/id_rsa.pub) to github (Account Settings - SSH Public Keys - Add another public key)

```
$ cd ~/cs2041/ass2
$ git remote add origin git@github.com:2041rocks/my_code.git
$ git push -u origin master
```

Now anyone anywhere can clone your repository by

```
git clone git@github.com:2041rocks/my_code.git
```

Labelling Versions

How to name an important version? (unique identifier)

Common approach: *file name + version "number"* (e.g. Perl 5.8.1)

There is no "standard" for *a.b.c* version numbers, but typically:

- *a* is major version number (changes when functionality/API changes)
- *b* is minor version number (changes when internals change)
- *c* is patch level (changes after each set of bug fixes are added)

Examples: Oracle 7.3.2, 8.0.5, 8.1.5, ...

How to store multiple versions (e.g. v3.2.3 and v3.2.4)?

We *could* simply store one complete file for each version.

Alternative approach:

- store complete file for version 3.2.3
- store differences between 3.2.3 and 3.2.4

A *delta* is the set of differences between two successive versions of a file.

Many VCSs store a combination of complete versions and deltas for each file.

Using Deltas

Creating version *N* of file *F* (F_N) from a collection of

- complete copies of *F* whose versions $< N$
- deltas for all versions in between complete copies

is achieved via:

```
get list of complete copies of F
choose highest complete version V << N
f = copy of F{V}
foreach delta between V .. N {
  f = f + delta
}
# f == F{N} (i.e. version N of F)
```

Programs like patch can apply deltas.

Delta Bandwidth Efficiency

An example of why deltas are useful:

- Google Chrome for Windows upgrades in the background almost daily
- Google Chrome is 10MB
- bsdiff delta = 700KB
- google custom delta (Courgette) = 80KB

- 200 full upgrades/year = 2GB/year
- 200 bsdiff upgrades/year = 140Mb/year
- 200 Courgette upgrades/year = 16Mb/year

Intro to the Web

Wednesday, 10 July 2019 5:15 PM

Resources:

<https://github.com/getify/You-Dont-Know-JS>

<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference>

JavaScript runs server side and in the browser. It is a simple, weakly typed, script language that is easy to pick up and develop with.

We will focus on JavaScript in the browser, language fundamentals, and frontend fundamentals.

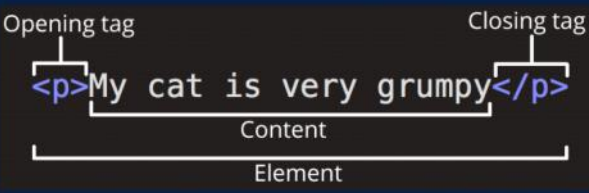
The Modern Web

The modern web is made of

- HTML - the scaffold for any modern webpage. A form of markup language used to format web documents
 - Lightweight, scaffold for webpages
 - Collection of tags and attribute-value pairs
 - Limited but core functionality

HTML Elements

The most basic selector, apply to the corresponding HTML element. eg:



Opening tag

Closing tag

Content

Element

Image Source: MDN

Tags

Tags define the type of element we are rendering, and help to semantically structure the document.

```
<html></html>, <body></body>, <main></main>
<!-- This is a comment -->

<!-- These tags are for structuring text -->
<p></p>, <h1></h1>, <h2></h2>

<!-- These tags are for lists -->
<ul>unordered</ul> or <ol>ordered</ol>
<li>list items</li>

<!-- These tags are for content structure -->
<nav></nav>, <article></article>, <section></section>, <form></form>

<!-- These tags are special tags -->
<img/>, <input/>, <button></button>
<!--
    Some people just prefer to use divs everywhere.
    Those people are bad people
-->
<div></div>
```

Attributes

Attributes control how an element behaves. There are many 'standard' attributes that apply to all elements. Eg. class, style, id.

```
<!-- Will render a text input -->
<input type="text" name="firstName"
placeholder="Jeff">
<!-- Will render a password field -->
<input type="password" name="password">

<!-- When this form is submitted it'll do a get
request on /some/path -->
<form action="/some/path" method="get"></form>

<!-- This image has a src attribute of fake.jpg -->

```

- CSS - the de facto standard for applying styles and basic animations
 - Arose to extend HTML functionality
 - "Style elements" were a poor, cumbersome way to style - CSS abstracted out style from markup using selectors

Property/Value Pairs

CSS uses a selector combined with a key-value property pair to style elements. There are hundreds of possible styles, but common ones include: margin, padding, color, font... to name a few.

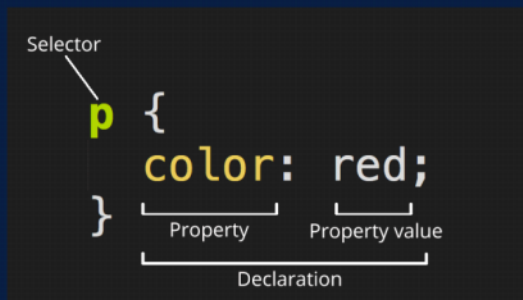


Image Source: MDN

Tag Selectors

The most basic selector, apply to the corresponding HTML element. eg:

```
p {
  color: black;
  font-weight: normal;
}

h1 {
  color: red;
  font-weight: bold;
}
```

Class Selectors

A more extensible approach to styling, denoted by the leading '.' in the selector. eg:

```
.box {
  width: 100px;
  height: 100px;
}

.red {
  background-color: red;
}

.blue {
  background-color: blue;
}
```

Pseudo Selectors

Make use of the : operator and must be combined with another selector. Examples, include element states or positions like :hover, :focus, :after, :before.

```
.box {
  width: 100px;
  height: 100px;
}

/* box with our red class will turn red on hover */
.box.red:hover {
  background-color: red;
}

/* A normal box will turn black */
.box:hover {
  background-color: black;
}
```

IDs and special selectors

IDs are denoted by a leading #, other special selectors include > and +.

```
/*
  Highest precedence but applies to only
  _one_ element.
*/

#nav {
  font-weight: bold;
}

/*
  The p child of the log will have opacity 0.5
*/
#logo > p {
  opacity: 0.5;
}
```

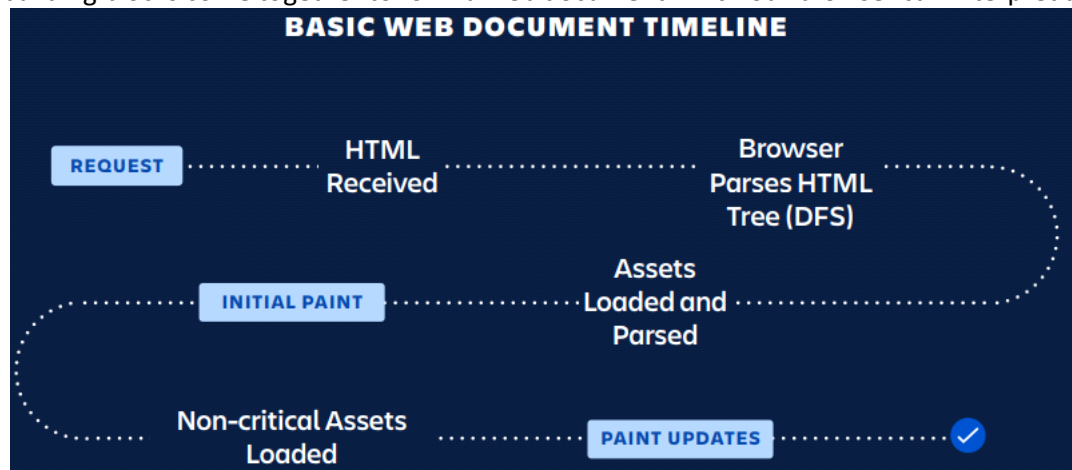
Specificity is a weight that is applied to a given CSS declaration, determined by the number of each selector type in the matching selector. By combining selectors and specificity, we can apply style rules as generally or specifically as we choose.

CSS Specificity

1. **Inline styles** - style rules declared inline are always of highest precedence
2. **Order of declaration** - later rules of equal specificity take precedence over earlier rules
3. **ID, Class, tag** - highest order specificity takes precedence, more selectors add specificity
 - JavaScript - facilitates 'dynamic web pages'
 - Initially facilitates animation, form validation, warnings, prompts
 - Extended in mid-2000s to support XHR/AJAX to become a core part of modern web

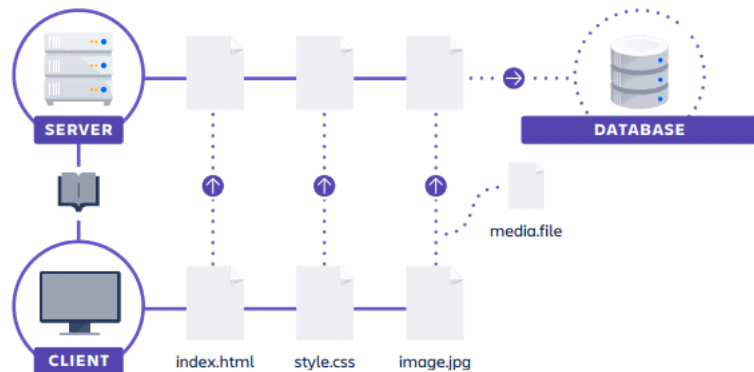
Putting them all together

These three building blocks come together to form a web document which our browser can interpret and update.



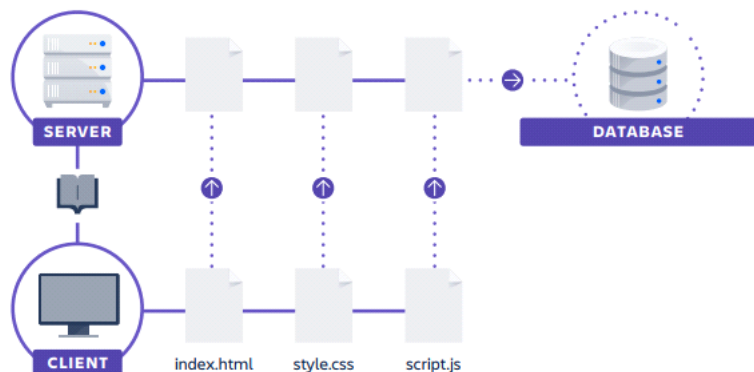
WEB BASICS

Server gets a request from your browser, serves HTML, which in turn asks for any additional assets.



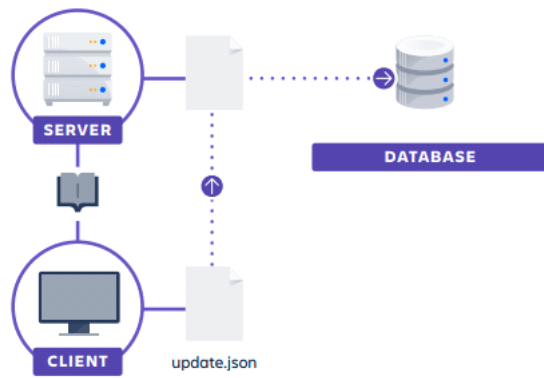
MODERN WEB

Similar initial setup but less reliant on server for CPU processing. Updates can happen in the background.



MODERN WEB

Updates only need to send critical information; significantly smaller payloads, no need to re-render the whole page.



JavaScript Overview

Wednesday, 10 July 2019 6:05 PM

<https://comp2041unsw.github.io/js/render.html?p=resources.md>
<https://comp2041unsw.github.io/js/render.html?p=notes/index.md>

Intro to JavaScript

Plain text - Scripts are provided and executed as plain text. They don't need special preparation or compilation to run.

Engine - Scripts then rely on an 'engine' to compile them into byte code. Engines are environment dependent. Mozilla (SpiderMonkey) and Chrome (V8) have different engines for example.

Single threaded - JavaScript relies on a concept of run-to-completion. Tasks are executed synchronously, and queued if multiple tasks need to run at the same time.

JS in the browser

JavaScript is only as powerful as its environment. In the browser its focus is UI manipulation.

It can:

- manipulate CSS, HTML and the web document directly and efficiently;
- react to user interaction;
- make web requests;
- store data in the browser;
- mine bitcoin.

It can't:

- Read or write from your operating system without explicit permission (eg. file upload).
- Arbitrarily request data from any site — permissions required.

Loading a script

Scripts are included on an HTML page in three main ways.

1. Inline

```
<!DOCTYPE html>
<html lang="en">
<body>
  <script>
    console.log('Hello Andrew!');
  </script>
</body>
</html>
```

2. From an external source

```
<!DOCTYPE html>
<html lang="en">
<body>
  <script src="script.js"></script>
</body>
</html>

// inside our script.js
console.log('Hello Andrew!');
```

3. From a module

```
<!DOCTYPE html>
<html lang="en">
<body>
  <!-- We'll see how this works a bit later -->
  <script type="module" src="script.js"></script>
</body>
</html>
```

Language and Syntax

The console:

```
// functionally similar
console.log('Hello World');
console.warn('Some warning');
console.error('Some bad thing');
```

Prompt and alert:

```
// outputs a prompt to the browser window
const message = prompt('Can I have a sheep?');
// outputs a dialog message to the browser window
alert('alert!: ', message);
// returns a boolean
const yesOrNo = confirm('Do you want more soup?');
```

Comments:

```
/**
 * Multiline comments
 */
// single line comments
// c-like
```

Semicolons are not necessary since JavaScript can usually infer where they belong, but it is always good to have them in case.

Variables

We can declare variables using: `var`, `const`, `let`... and an implicit fourth global

let can be re-assigned and declared ahead of time. It is a common way to declare variables

const stops a variable from being reassigned. Its value cannot be directly mutated, but if it points to a reference, the underlying object can still be changed.

Examples:

Strings

```
// strings can be declared like this
const aString = 'hello world'
// or like this
const anotherThing = "Hello World"
// or like this
const someAnotherThing = `hello world`

// strings can be concatenated
// with the + operator (like python)
const firstName = 'Alex',
  lastName = 'Hinds'
const fullName = 'Alex' + ' ' + 'Hinds' // Alex Hinds
// OR
const fullNameWithTemplate = `${firstName} ${lastName}`;
// strings also have a number of methods natively
available
anotherThing.toUpperCase() // HELLO WORLD
anotherThing.replace(/[aeiouAEIOU]/g, ''); // Hll Wrld
```

Numbers

```
/** numbers */
const anInt = 1;
// floats
const aFloat = 1.2;
```

```
// exponents
const exp = 1e2;

/** numbers */
const anInt = 123;
/** aFloat */
const aFloat = 1.2;
// exponents
const exp = 1e2;
const longFloat = 1.231215;
// Like strings numbers also have some handy
builtin
const shortFloat = longFloat.toFixed(2);
// toString translates to baseX representation
const binaryRepresentation = anInt.toString(2);
// there are many more
```

Boolean

```
// standard declaration
const isTrue = true;
const isFalse = false;
// not operator to coerce a boolean
const isTrueAlso = !false;
const isFalseAlso = !true;
// most types can be coerced with a double !!
const emptyString = '';
const falseValue = !!'';

const noValue; // undefined
// rarely something you do
const noValue = undefined;
// different to undefined,
const definitivelySet = null;
```

When considering booleans,

| | |
|-------|--------------------------------------|
| True | Non-zero int, non-empty string, true |
| False | 0, empty string, false, undefined |

A variable is considered undefined if it is declared but not given a value. This is different to null, where we can explicitly set the variable value to null.

Functions

There are two ways of writing functions in JavaScript.

The basic method is to declare function, followed by the function name and its parameters:

```
function checkAge(age) {
    if (age > 18) {
        return true;
    } else {
        return confirm('Can you legally drink?');
    }
}

let age = prompt('How old are you?', 18);
if (checkAge(age)) {
    alert('Access granted');
} else {
    alert('Access denied');
}
```

The other method is to use arrows:

```

/* think about what this functions do */
const multiply = a => b => a * b;
const pluck = key => object => object[key];

// let's say tax of 10% for GST and a 5 % first customer
discount
const discount = multiply(0.95);
const tax = multiply(1.10);

// the format required for sum
const sum = (num, secondNum) => num + secondNum;

```

Arrays

Declaration:

```

let arr = new Array();
let arr = [];

```

Access:

```

let fruits = ["Apple", "Orange", "Plum"];
alert( fruits[0] ); // Apple
alert( fruits[1] ); // Orange
alert( fruits[2] ); // Plum

```

There are two core ways to manipulate arrays in JavaScript

1. Impure manipulations (the underlying array is effected)
2. Pure manipulations (the method returns a new array)

There is no 'right' way to use these methods, but just be aware of the above when you make your manipulations.

```

// impure manipulations
const a = [];
a.push(1) // a = [ 1 ]
a.unshift(2) // a

// pure manipulations
const b = []
const c = b.concat(1, 2, 3)
// c = [ 1, 2, 3 ] b unchanged
const d = c.map((value) => value * 2);
// d = [ 2, 4, 6]

```

Objects

```

// note const limitations apply only to reassigning
// the Object. Object properties can still be altered
// with a const declaration.
const myObject = {};
// or like this
const myOtherObject = new Object();
const myUser = {
  id: 'UAAAAAAA',
  displayName: 'Jane Citizen',
  age: 25,
};
// You can also assign & read properties:
console.log(myUser.age);
// > 25
console.log(myUser['age']);
// > 25
// setting attributes

```

```

myUser.age = 29;
myUser.address = '123 Fake Street';

// note the use of this in this special constructor
// also note the caps (a convention for constructor functions)
function Person(firstName, lastName, age) {
    this.firstName = firstName;
    this.lastName = lastName;
    this.age = age;
}

Person.prototype.getFullName = function() {
    return `${this.firstName} ${this.lastName}`;
};

Person.prototype.canDrinkAlcohol = function() {
    return this.age >= 18;
};

// now if we call the constructor function we get this
new Person('Jeff', 'Goldblum', 50);
// => Person { firstName: 'Jeff', lastName: 'Goldblum', age: 50 }

class Person {
    constructor(firstName, lastName, age) {
        this.firstName = firstName;
        this.lastName = lastName;
        this.age = age;
    }

    getFullName() {
        return `${this.firstName} ${this.lastName}`;
    }

    canDrinkAlcohol() {
        return this.age >= 18;
    }
}

new Person('Jeff', 'Goldblum', 50)

```

This

The object that 'this' refers to changes every time the execution context is changed.

<https://medium.com/quick-code/understanding-the-this-keyword-in-javascript-cb76d4c7c5e8>

Control Flow and Loops

If/else statements

```

if (condition) {
    // do something
}
if (condition) {
    // do something
} else {
    // do something
}

// as with c, one liners don't require brackets. (don't do this though!)
if (condition)

```



```

    // do something
else
    // something else

// And of course ternary
const x = condition ? 22 : 0;

```

Switch

```

switch(expression) {
  case x:
    // code block
    break;
  case y:
    // code block
    break;
  default:
    // code block
}

```

Loops

```

// same as c
let index = 0;
while (index < array.length) {
  let value = array[index];
  index++;
}

do {
  let value = array[index];
  index++;
} while (index < array.length);

// Iteration over an array
for (let index = 0; index < array.length; index++) {
  // do something with item
  // very similar to c
  let value = array[index];
}

// Iteration over an object/array
for (const property in items) {
  // do something with item
  // very similar to python
  let value = items[property];
}

let string1 = "";
const object1 = {a: 1, b: 2, c: 3};
for (let property1 in object1) {
  string1 += object1[property1];
}
console.log(string1);
// expected output: "123"

iterable = [10, 20, 30];
for (let value of iterable) {
  value += 1;
  console.log(value);
}
// 11

```

```
// 21
// 31

// OR like this.. other ways too
for (let char of "test") {
    // triggers 4 times: once for each character
    alert( char ); // t, then e, then s, then t
}
```

Errors

try {} catch (e) {} is your go-to error handler. Note, unhandled errors in JS will cause your scripts to crash!

```
try {
    throw new Error('Whoops!');
} catch (e) {
    console.log(e.name + ': ' + e.message);
}
```

Import/Export

```
// lib/math.js
export function sum (x, y) { return x + y }
export var pi = 3.141593
// someApp.js
import * as math from "lib/math"
console.log("2*Pi = " + math.sum(math.pi, math.pi))
// otherApp.js
import { sum, pi } from "lib/math"
console.log("2*Pi = " + sum(pi, pi))

// relative path
import defaultExport from "module-name";
// alias the default export
import { default as alias } from "module-name";
// import all exports under the 'name' namespace
import * as name from "module-name";
// import namedExport
import { namedExport } from "module-name";
// alias using 'as'
import { namedExport as alias } from "module-name";
// combining a few imports from the same module
import defaultExport, { export1 , export2 } from "module-name";

// AND The export versions
export default Export;
// re-export a default import as an alias
export { default as alias } from "module-name";
// export all exports under the 'name' namespace
export * from "module-name";
// export namedExport
export { namedExport } from "module-name";
// alias using 'as' from an external module
export { namedExport as alias } from "module-name";
// combining a few exports from the same module
export { defaultExport as default, export1 , export2 };
```

Leveraging Built-in Libraries

Libraries are maintained by professional developers working on browsers. They also don't need to be downloaded by your users.

See: <http://es6-features.org/>

Other Notes:

== try make things equal even if they aren't. Use === for equality evaluation

Const noValue; is not allowed since it is not assigned a value instead do let noValue;

```
Function sayHello() {  
  console.log('Hello');  
}
```

```
Const sayHello = function() {  
  Console.log('Hello');  
}
```

```
Const sayHello = () => {  
  Console.log('Hello');  
}
```

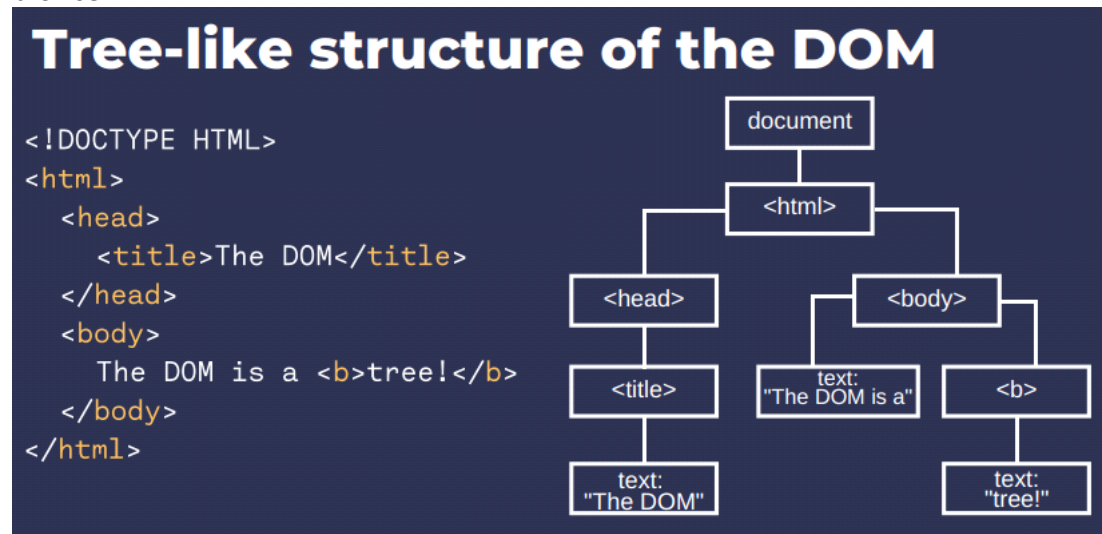
```
Const sayHelloName = (name) => {  
  Console.log(`Hello, ${name}`)  
}  
Const sayHelloName = (name, age) => {  
  Console.log(`Hello, ${name}, you're ${age}`)  
}
```

```
Const sum = (a, b) ==> a + b;  
Is the same as  
Function sum(a,b) {  
  Return a+b;  
}
```

The DOM API & Events

Tuesday, 16 July 2019 10:35 PM

The DOM (Document Object Model) is an interface that allows JavaScript to interact with HTML through the browser.



DOM Data Types

The DOM has the following data types.

- **Document** - the type of the `document` object. It represents the root of the entire DOM
- **Element** - a node in the DOM tree. Objects of this type implement an interface that allows interacting with the document
- **NodeList** - an array of elements, like the kind that is returned by the method `getElementsByTagName()`

Understanding DOM Elements

An **Element** is the base class for all types of objects in the Document. Different HTML tags/elements correspond to different Element types in JS. Different Element types include: `HTMLInputElement`, `HTMLSpanElement`, `HTMLDivElement`, `HTMLScriptElement`, `HTMLHeadingElement`, `HTMLImageElement`...

The Element interface provides us with many properties and methods, e.g.

`clientHeight`, `clientWidth`, `clientLeft`, `clientTop`, `scrollTo()`, `setAttribute()`

Specific types of elements may also have different attributes/methods e.g. for `HTMLInputElement`

`focus()` - focuses the cursor on this input element

`select()` - selects all the text in the input element

Interacting with the DOM in JavaScript

Reading the DOM

```
// Returns an html element with the given id
document.getElementById(id);

// Returns a DOM HTMLCollection of all matches
document.getElementsByTagName(name);
document.getElementsByClassName(classname);

// Returns the first Element that matches the selector
document.querySelector(query);
```

Writing to the DOM

```
// Create a new div element
let element = document.createElement("div");

// Create a new text node
```

```
let textNode = document.createTextNode("Some text");

// Adding and removing elements
element.appendChild(textNode);
element.removeChild(textNode);

// Making changes to attributes
button.setAttribute("disabled", "");
```

Changing the Style of an Element

An element has a "style" property which corresponds to the "style" attribute of the HTML element. This can be modified in the JavaScript.

```
// Changing element.style
element.style.left = "50px"; // Note: don't forget units!

// Adding 5px to the left value
let newLeft = parseInt(element.style.left, 10) + 5 + "px";
element.style.left = newLeft;
element.style.backgroundColor = "red"; // Note: camelCase
```

Getting the Style of an Element

Note: `element.style.left` will only be present on an element if the left property was set in inline styles, or by scripting (not if it was set in CSS)

```
// Getting computed style
let computedStyle = window.getComputedStyle(element, null)
let bgColor = computedStyle.getPropertyValue("background-color")
```

Getting the classes of an Element

Another way of modifying the style of the element is to change the classnames which exist on the element. This can be done using the "classList" property

```
// Changing element.classList
element.classList.add("class");
element.classList.remove("class");
element.classList.toggle("class");
element.classList.contains("class"); // returns true if class
exists on element
```

Scrolling

```
// Get the current scroll position of the page
console.log(window.scrollX);
console.log(window.scrollY);

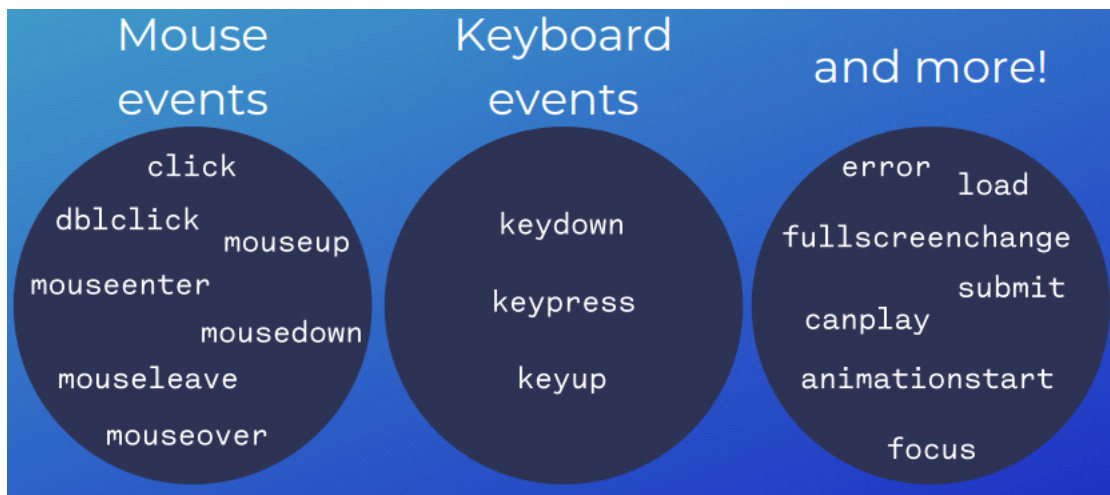
// Scroll to a position on the page:
window.scrollTo({
  top: 100,
  left: 0,
  behavior: "smooth",
});
```

Events

An **event** is a signal that a "thing" has happened to a DOM element.

This "thing" can be the element **loading**, a **click**, or a **key press**. We can use events to run JavaScript code in response to user actions.

Some examples of events:



Cross Platform Compatibility

Event handlers need to:

- Be touch-only-device friendly
- Have responsive design
- Have good performance on low-end devices

We should be aware of the following event for cross-platform compatibility.

1. Touch-only devices cannot hover - avoid using hover events to reveal core functionality
2. Fullscreen API does not work on iOS/Safari, so check before using onfullscreen change

We can usually check if a device is a touch only device by

1. Checking its screen size
2. Adding an event listener for touchstart
3. `window.matchMedia('(hover: none)').matches`

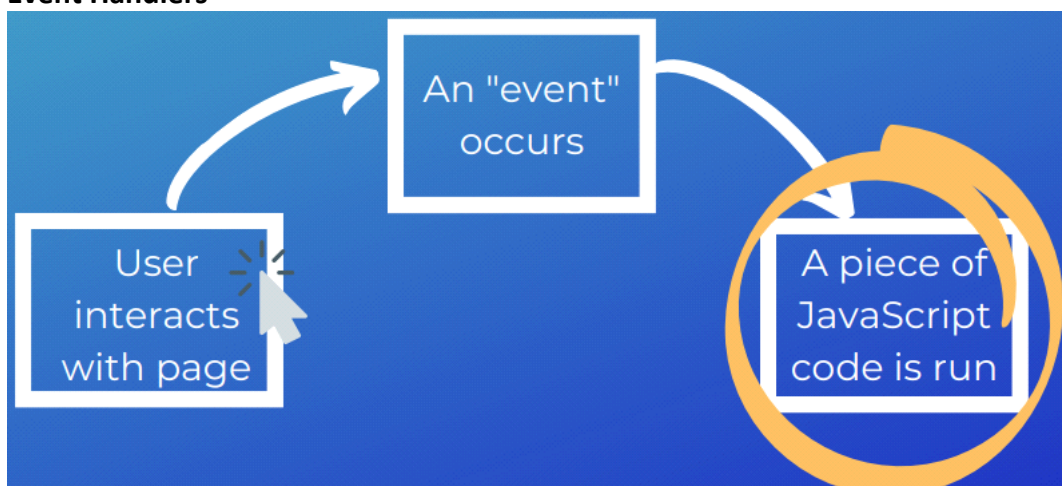
Increasing Performance

Some ways we can make our webapp more performant is by

- Minifying your assets - make your code as short as possible
- Avoid un-performant CSS
- Choose appropriate image resolutions - choose the smallest resolutions that still make your app look high-quality
- Avoiding unnecessary packages

Dev Tools are available on your browser to measure load and performance times of any webpage

Event Handlers



We can add event handlers:

1. In **HTML**

```
<input
  value="Click me"
  onclick="alert('Clicked!')"
  type="button"
>
```

2. As a DOM Property

```
let element = document.getElementById('btn');

element.onclick = () => {
  alert('Button was clicked!');
};
```

Be careful to not add parentheses to a function, when assigning that function to a variable. You will be executing the function instead of assigning it. E.g

```
function doSomething() {
  alert('hello');
}
element.onclick = doSomething(); // incorrect
// vs
element.onclick = doSomething;   // correct
```

3. Using addEventListener/removeEventListener

```
// Definition:
target.addEventListener(
  type, // e.g. 'click', 'mousemove'
  listener, // the callback
  [options]
);

// Example:
let element = document.getElementById('btn');
let handler = () => {
  alert('button was clicked');
}

element.addEventListener('click', handler);
element.removeEventListener('click', handler);
```

"this" binding

this is all about **where** a function is invoked

When a function is in an object, **this** refers to the object itself.

When an event is triggered, **this** is bound to the element on which the listener is attached

Event Interface

The parameter to an event handler is an **event object**.

```
document.addEventListener('mousemove', (event) => {
  console.log(event.clientX);
  console.log(event.clientY);
});
```

The event object represents the even that has taken place, and has properties describing details of the event.

Event Interface Properties

Some of the properties on the event interface include:

```
event.currentTarget // current element handler is running on
event.timeStamp // time the event was created (in ms)
event.type // name of the event, e.g. 'click'
```

Different types of events have specific properties:

```
event.clientX // A MouseEvent has the X and Y coordinate  
event.key // A KeyboardEvent has the keycode of the key that was pressed
```

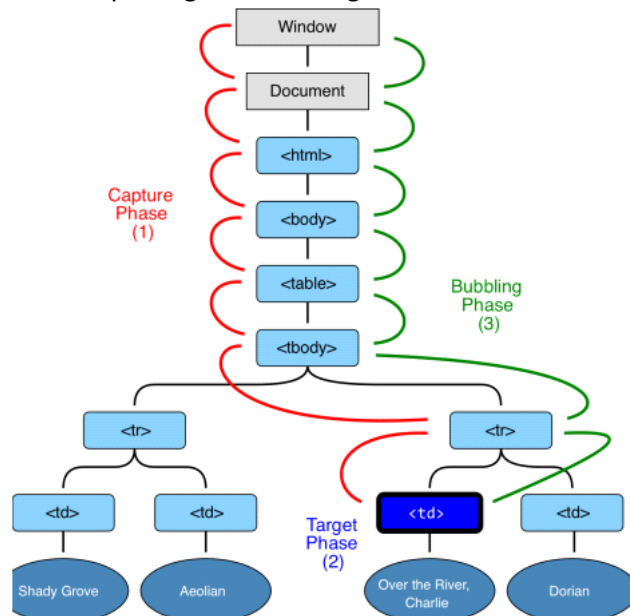
The Event Loop

The event loop is a single-threaded loop which runs in the browser, and manages all events.

When an event is triggered, the event is added to the queue.

JavaScript uses a run-to-completion model, meaning it will not handle a new event until the current event has completed

Event Capturing and Bubbling



Prevent Default

Some types of DOM elements have default behaviour, e.g.

1. Clicking an input checkbox toggles the checkbox
2. Images have a default drag and drop behaviour to allow you to drag them into another location
3. Key presses into a text input field has the default behaviour of entering that text into the input field

To stop the default behaviour of an event, use:

```
event.preventDefault();
```

Keypress vs key down

CSS Animations, Forms, Local Storage

Sunday, 21 July 2019 8:54 PM

CSS Animations

CSS has a wide variety of animations available. There are a number of properties you can use to control what your animation looks like:

```
animation-name // custom
animation-duration // length of the animation e.g. 2s
animation-delay // delay before the animation starts e.g. 2s
animation-iteration-count // e.g. 2, infinite
animation-direction // e.g. normal, alternate, reverse
animation-timing-function // e.g. ease, linear
animation-fill-mode // e.g. forwards, backwards
animation // the shorthand for the above
```

When using the shorthand notation, there is a specific order in which properties are used:

```
/* @keyframes duration | timing-function | delay | iteration-count
| direction | fill-mode | play-state | name */
animation: 3s ease-in 1s 2 reverse both paused slidein;
```

Keyframes

Keyframes are a way of specifying the values of CSS properties at intermediate steps throughout an animation.

Animate margins for a *slide effect*

```
@keyframes slidein {
  from {
    margin-left: 100%;
    width: 300%;
  }
  to {
    margin-left: 0%;
    width: 100%;
  }
}

@keyframes identifier {
  0% { top: 0; }
  50% { top: 30px; }
  50% { top: 10px; }
  100% { top: 0; }
}
```

Animate opacity for a *fade effect*

```
@keyframes fadein {
  from {
    opacity: 0;
  }
  to {
    opacity: 1;
  }
}

animation: fadein 0.4s ease-in-out
```

Animate scale for a *zoom effect*

```
@keyframes zoom {
  from {
    transform: scale(1.5);
  }
  to {
    transform: scale(1);
  }
}

animation: fade .3s ease-in reverse;
```

Animate rotation for a *spin effect*

```
@keyframes rotate {
  from {
    transform: rotate(360deg);
  }
  to {
    transform: rotate(0deg);
  }
}

animation: rotation 0.7s ease-in;
```

setTimeout is a way to add a function that runs after a certain amount of time

```
window.setTimeout(() => {
  console.log('timed out');
}, 3000)
```

If you want the timeout to reoccur use setInterval

```
window.setInterval(() => {
  console.log('timed out');
}, 3000)
```

Forms

A **HTML *<form> element*** is a way of defining a form which is used to get user input. They consist of different types of **input elements**:

- Text fields
- Checkboxes
- Radio buttons
- Submit buttons

We specify the type of input element by using the type attribute: **<input type="text">**

```
<form>
  First name: <br>
  <input type="text" name="firstname">
  Last name: <br>
  <input type="text" name="lastname">
</form>
```

document.forms

When you have forms in your document, they can be found in a special document property called **document.forms**.

This is a "named collection". i.e it is both named and ordered. We can use both the name or the

number in the document to get the form.

```
document.forms.test // the form with name="test"
document.forms["test"] // also the form with name="test"
document.forms[0] // the first form in the document
```

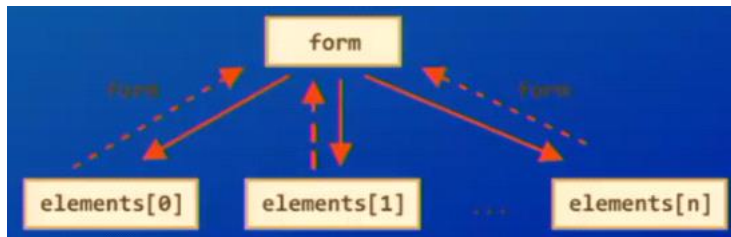
form.elements

Each form has a field form.elements, which has all of the elements in that form. It is also a named collection.

| HTML | JS |
|--|---|
| <pre><form> <input type="text" name="fname"> <input type="radio" name="age" value="10"> <input type="radio" name="age" value="20"> </form></pre> | <pre>const form = document.forms[0]; // element with the name "fname" form.elements.fname; // shorter notation: form.fname; // since there are multiple elements with the name "age", this returns a collection const ages = form.elements.age;</pre> |

Backreferences

Each form element also stores a backreference to the form it came from. Just use `element.form`



Form Values

To get the value of a form element:

```
To get the value of a form element:

// To get the text for an input element or textarea:
input.value

// To get a boolean for a checkbox or radio button
input.checked

// For a <select> tag
select.options // the collection of <option>s
select.value // the value of the currently selected option,
select.selectedIndex // the index of the currently selected option
```

Submit Buttons and onsubmit

`<input type="submit">` defines a button for submitting the form data to a form-handler. Clicking on a submit button triggers a "submit" event on the form. We can listen to this even and run form validation.

```
form.addEventListener('submit', (event) => {
  event.preventDefault();
  // form validation can go here
});
```

Local Storage

`Window.localStorage` is an API that allows you to read and write to a storage object in the document. The stored data in this storage object is **persisted between sessions**. This means we can save and retrieve data when a user closes their tab or browser.

```
// Add a data item given the key and value
localStorage.setItem(key, value);

// Retrieves an item from localStorage given a key
const value = localStorage.getItem(key);

// Remove an item with a given key from localStorage
localStorage.removeItem(key);

// Remove all items from localStorage
localStorage.clear();
```

Extra notes

Query something by:

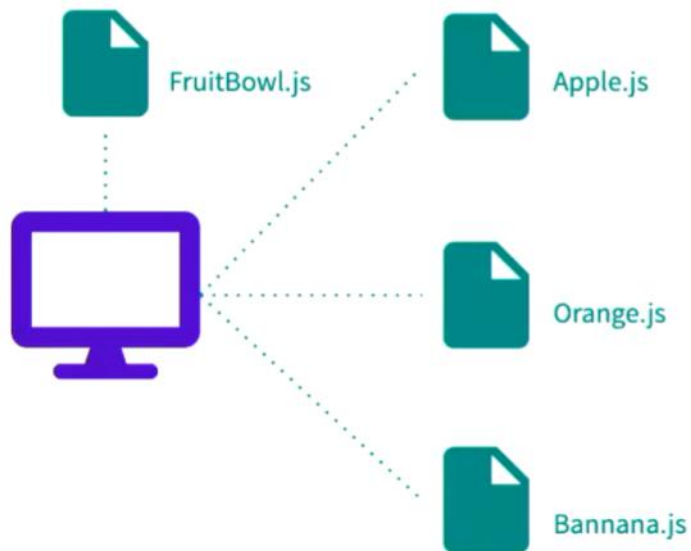
- Id - #name
- Class - .name
- Tag - name

Also return the first match

Async Programming

Wednesday, 24 July 2019 5:54 PM

Modules Refresher



FruitBowl.js

```
import Apple from './Apple.js';
import Orange from './Orange.js';
import Bannana from './Bannana.js';

makeFruitBowl(Apple, Orange, Bannana)
```

```
Apple = {
  yummy: true
}
export Apple

Orange = {
  scurvy: null
}
export Orange

Banana = {
  ew: 'yes'
}
export default Banana
```

```
// Import apple and Banana, but rename Banana
to B
import {Apple, Banana as B} from 'fruits.js';
// import the default export from fruits.js
and name it O
import O from 'fruits.js';
```

Note: {} means you are exporting a specific item out of the .js file

Basic Client Server Interactions



Client

Your home PC, laptop, phone, washing machine etc.



Server

A dedicated machine run by a website to receive and process requests



Client

Sends a request to a server for a page, lets say google.com

Server

Processes the request (what is this client asking for, are they allowed to see this?)



Client

Sends a request to a server for a page, lets say google.com

Server

Decides what to send back, creates a response of HTML, CSS and JS and shoots it off
This is called **Server Side Rendering**



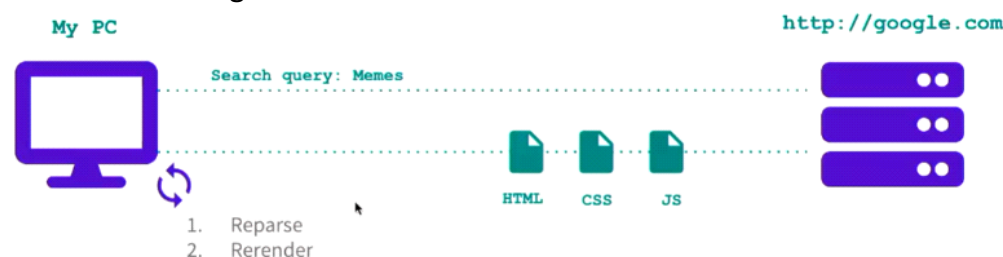
Client ⚙️

Needs to spend time waiting for this (sometimes heavy) file to download and then spend time parsing and rendering it

⚙️ Server

Needs to spend time and resources forming a full page response and sending it over

Forms With Full Page Reload



Client

Forms and interactivity were built on top of this, a request with some information is sent and the response triggers a reparse and rerender

Server

And the server would use this information to create a brand new page and return that

AJAX (Asynchronous Javascript and XML)



Client

via js sends just a small request to autocomplete 'M' and adds the suggestions to the page without doing a full render

Server

Quickly processes this request and sends back just some suggestions

AJAX is a development technique in which a frontend client sends and receives data to the backend **asynchronously** (in the background) without interfering with the display and behaviour of the existing page.

Overall it:

- Smoothens UI
- Improves speed
- Allows for offline web apps
- Separates frontend and backend responsibilities

A **progressive web app** is an app, which when disconnected from the internet it does not *cry*. The app is completely functional until you need to interact with the server

The Event Loop

Synchronous operations can only occur one at a time. You must wait for the current action to finish before starting a new one. E.g. the old server-client model where each request had to be completed before you sent a new one.



Asynchronous operations allow multiple operations to occur at the same time. Programmers can parallelise their program and do many things at one. E.g. the AJAX model where you can fire off 10 requests before the first one returns.



Concurrency Models

Doing multiple things at the same time requires a concurrency model. That is how we distribute our work and reason about our code. There are three main ways:

• Threads/pre-emptive multitasking

Here you write a bunch of code and spin up threads. Each thread runs the code assuming it is the only thing running. At any random time, the thread can be stopped, its state saved and the cpu is tasked with some other bits of work.

We say that a thread can be pre-empted, so we need mutex's and semaphores to guard critical sections and variables to make sure we don't get interrupted if we are doing something fiddly. This is the typical multitasking mode found in many languages like Java and Python.

This model is not useful for us since all transformations to the DOM **must happen in a predictable order** to avoid broken pages.

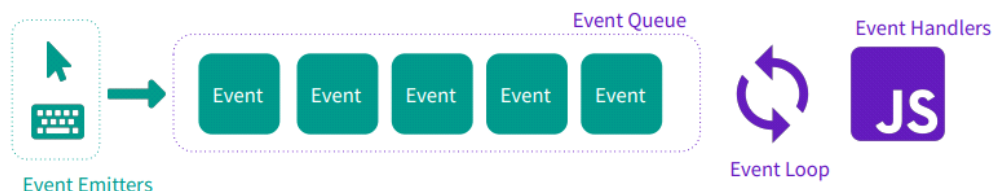
• Coroutines/cooperative multitasking

Coroutines are separated flows of execution which run in tandem. Coroutines can cooperatively **yield** to other coroutines. **Yield** saves the state of co-routine A, and resumes B's state from its previous yield point. There is no pre-emption between yields, so there is no need for mutex's or semaphores to guard critical sections.

This is better but often designed with the idea of having predefined processes that share the CPU and isn't as suited to the dynamic way events are generated on the web

• Event driven

This is the way JS handles concurrency, the core of this model is the idea of an **event loop**. This is a queue of events that have occurred, events are popped off to be handled and pushed on as they occur .



Each event is given to its respective handler and the loop waits for the handler to finish processing before handling the next event. In this way our javascript runs start to finish in a predictable way (no need for locks) but we can still have many events at the same time.

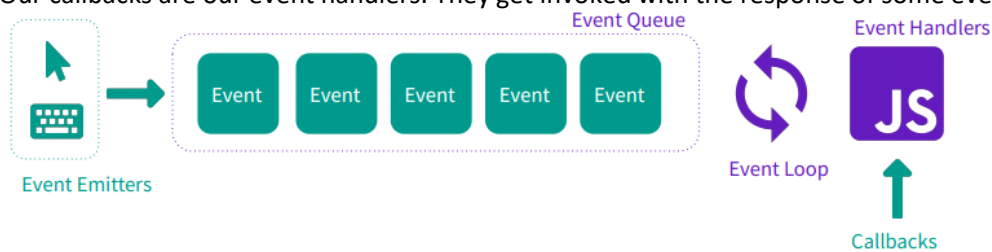
What is a callback?

A call back is a function that is called when pending work has completed. You can use either named function or an anonymous lambda.

```
fs.readFile('foo.txt', (result, err) => {
  // Do something with result here
})
```

```
function onComplete(result, err) {
  // Do something with result here
}
fs.readFile('foo.txt', onComplete)
```

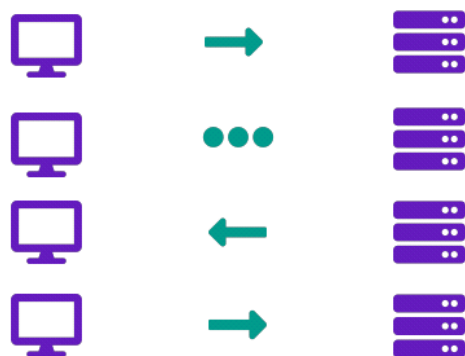
Our callbacks are our event handlers. They get invoked with the response of some event



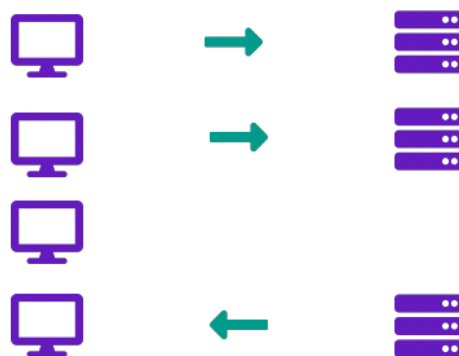
Network and Resource Fetching

Only 1 task can run at a time so event loops suck for parallelized workloads, i.e CPU bound tasks like large calculations. It is however incredibly well suited to I/O bound tasks, that is tasks that spend lots of time waiting for a server or device to respond. We can fire off a request to a server and then continue doing other work, when the server responds an event is generated which we can then handle.

Synchronous Way



Asynchronous Way



Blocking

Remember JS can only run 1 thing at a time so if that 1 thing takes 6 seconds then during that time things can be added to the queue but nothing gets popped off.



By not waiting for a response and expecting the event loop to trigger a handler when the time is right we allow ourselves to keep the page running (we can also do heavy tasks in [webworkers](#) but we'll cover that later)



XMLHTTP and Callback Hell

Writing code that is async is a difficult problem and there have been many diff paradigms over the life of JS. The first is to use callbacks with XMLHttpRequest

```
console.log('Loading...');
const request = new XMLHttpRequest();
// 'false' makes the request synchronous
request.open('GET', 'url', false);
request.send(null);
if (request.status === 200) {
  console.log(request.responseText);
}
```

```
console.log('Loading...');
const xhr = new XMLHttpRequest();
xhr.open("GET", "https://api.ipify.org?format=json"
true);
xhr.onload = function (e) {
  if (xhr.readyState === 4) {
    if (xhr.status === 200) {
      console.log(xhr.responseText);
    }
  }
};
xhr.send(null);
```

Error Handling with callbacks

Errors are fairly common in web, maybe you weren't logged in so your request failed, maybe your token expired, maybe the connection dropped. Your network interactions WILL fail so you have to be able to gracefully handle this.

Chaining

Another common thing in web is wanting to do things in a specific order, as a simple example lets say we want to animate some text being written.

Callback Hell

Callback hell makes it

- difficult to reason about the code and what it's doing
- difficult to debug
- difficult to extend / edit
- looks ugly
- fragile, a single missed callback in that mess causes a failure
- difficult to accurately catch and resolve errors

Avoid callback hell by

- Not nesting functions, give each one a name and place at the top scope
- Keep the code shallow wherever possible
- Avoid overuse of callbacks for things that can be synchronous
- Use promises

Avoiding Callback Hell

```
function sayHello(callback) {
  console.log('Hello!');
  callback();
}
sayHello(() => {
  console.log('We said hello.');
```

```
function sayHello() {
  console.log('Hello!');
}
sayHello();
console.log('We said hello.');
```

Fetch and Promise Heaven

Promises are an abstraction around async work, which give us access to "future" values.

A Promise is an object representing the eventual completion or failure of an asynchronous operation.

A Promise can be in a number of states

- Fulfilled (or Resolved): The action relating to the promise succeeded.
- Rejected: The action relating to the promise failed.
- Pending: Hasn't yet fulfilled or rejected.
- Settled: Has been fulfilled or rejected.

Chaining With Promises

.then can return a promise in itself allowing you to do multiple async actions in sequence

```
doSomething(result => {
  doSomethingElse(result, newResult => {
    doThirdThing(newResult, finalResult => {
      console.log(finalResult);
    }, handleFailure);
  }, handleFailure);
}, handleFailure);
```

```
doSomething()
  .then(result => doSomethingElse(result))
  .then(newResult => doThirdThing(newResult))
  .then(finalResult => {
    console.log(finalResult);
  })
  .catch(handleFailure);
```

Catching promises

A promise chain stops if there's an exception and looks down the chain for a catch handler instead.

```
doSomething()
  .then(result => doSomethingElse(result))
  .then(newResult => doThirdThing(newResult))
  .then(finalResult => console.log(`Got the final result: ${finalResult}`))
  .catch(failureCallback);
```

```
try {
  const result = syncDoSomething();
  const newResult = syncDoSomethingElse(result);
  const finalResult = syncDoThirdThing(newResult);
  console.log(`Got the final result: ${finalResult}`);
} catch(error) {
  failureCallback(error);
}
```

A catch statement can be used to continue the chain after a failure as it also returns a promise that resolves

```
fetchUsers()
  .then(() => {
    throw new Error('Something failed');
    console.log('Do this');
  })
  .catch(() => {
    console.log('Do that');
  })
  .then(() => {
    console.log('Do this, no matter what happened before');
  });
```

```
Do that
Do this, no matter what happened before
```

Fetch

You could wrap XMLHttpRequest to give you all the nice features of promises or you can just use fetch. It brings the magic of promises by default as well as being a bit less confusing to read/use.

```
fetch('http://example.com/movies.json')
  .then(response => response.json())
  .then(myJson => {
    console.log(myJson);
  });
```

Fetch/Promise Notes

- Will not throw an error if a HTTP request fails
 - if a request returns 400 or 500 etc. it assumes this may be expected
 - You will have to detect this downstream and raise a result if it is **not** expected
- Request.JSON() returns a promise which returns a json object, not just a json object
- All requests are assumed to be GET unless you specify otherwise
- **You can't cancel a Promise**
 - This poses potential issues when using promises for long running tasks like heavy downloads
- You shouldn't use promises

- When you have a callback situation where the callback is designed to be called multiple times
- For situations where the action often does not often finish or occur
- You can still get into promise hell if you don't use them correctly, make sure if you ever have a further async action you return a promise and keep your logic flat

Avoiding Promise Hell

```
fetchBook()
  .then(book => {
    return formatBook(book)
    .then(book => {
      return sendBookToPrinter(book);
    });
  });
```

```
fetchBook()
  .then(book => formatBook(book))
  .then(book => sendBookToPrinter(book));
```

More Advanced Fetch Use Cases

```
const url = "api.com/posts";
const payload = {
  title: 'My fun day!',
  content: 'Had a fun day it\'s in the title'
}
const options = {
  method: 'POST',
  headers: {
    'Content-Type': 'application/json'
  },
  body: JSON.stringify(payload)
}

fetch(url, options)
  .then(r => console.log('WOW!'));
```

```
const options = {
  method: 'POST',
  headers: {
    'Content-Type': 'application/json',
    // get this token after you login
    'Authorization': 'Token 2dcc8215'
  },
  body: JSON.stringify({governmentSecrets: 'yes'})
}

fetch('nsa.secret.com', options)
  .then(r => console.log('WOW!'));
```

Promise++ And Async Await

Promise Helpers

What if I want to fetch 4 things and need ALL 4 before processing them? Use `promise.all()`!

```
function handle(f) {
  fruits.push(r)
  if (fruits.length === 4)
    make_fruit_salad(fruits)
}
fruits = []
fetch('myapi/apple').then(r => handle(r))
fetch('myapi/orange').then(r => handle(r))
fetch('myapi/banana').then(r => handle(r))
fetch('myapi/pear').then(r => handle(r))
```

```
fruits = [
  fetch('myapi/apple'),
  fetch('myapi/orange'),
  fetch('myapi/banana'),
  fetch('myapi/pear')
]

Promise.all(fruits)
  .then(fruits => make_fruit_salad(fruits))
```

What if I want to fetch 4 things and just want whatever is faster? Use `promise.race()`!

```
function handle(f) {
  if (fruit === null) {
    fruit = f
    eat(f)
  }
}
fruit = null;
fetch('myapi/apple').then(r => handle(r))
fetch('myapi/orange').then(r => handle(r))
fetch('myapi/banana').then(r => handle(r))
fetch('myapi/pear').then(r => handle(r))
```

```
fruits = [
  fetch('myapi/apple'),
  fetch('myapi/orange'),
  fetch('myapi/banana'),
  fetch('myapi/pear')
]

Promise.race(fruits)
  .then(fruit => eat(fruits))
```

Async Await

This is just Syntactic sugar around Promises which helps you write code that feels more synchronous.

Rather than registering some handler for an event we can **await** it. This basically returns from the function and resumes it when the response is ready.

Our code becomes so much easier to reason about and additionally takes away the additional energy it takes to write things correctly.

You'll find similar paradigms in other languages like python and you'll find this concept invaluable when working with websockets

```
function sleep(time) {
  return new Promise((resolve, reject) => {
    setTimeout(function(){resolve(time);}, time);
  });
}

// you can only await in a async function
async function animate() {
  document.getElementById('output').innerText += 'the...'
  // wait until the promise
  // resolves before continuing
  await sleep(1000)
  document.getElementById('output').innerText += 'suspense'
}
animate()
```

```
function sleep(time) {
  return new Promise((resolve, reject) => {
    setTimeout(function(){resolve(time);}, time);
  });
}

// you can only await in a async function
function animate() {
  document.getElementById('output').innerText += 'the...'
  // wait until the promise
  // resolves before continuing
  return sleep(1000).then(_ => {
    document.getElementById('output').innerText += 'suspense'
    return new Promise.resolve()
  })
}
animate()
```

Modern Web Design

This is an overview as the field is both very complex and fastly evolving, many JS libraries have been made over the years to do a couple of things:

- Address Javascripts shortcomings
 - better api (jquery for example used \$('') rather than 'document.getElementById etc.')
 - modules (before they were implemented)
 - Adding more type safety
- Modularise the process of building UI
 - Makes things easier to manage and reuse
 - Helps decrease coupling
- Speed up DOM interactions
 - use smart code to minimise the number of times you touch the DOM to a bare minimum to further increase speed
- Make JS dev faster and less verbose

Frameworks

Some Libraries you'll hear about



A library from facebook where all your UI is contained in a virtual DOM in neat components that then, on changing, trigger a small number of DOM operations to make the REAL dom match. Also can be ported to a native app.



Similar to react with the virtual DOM but more of a focus on HTML, CSS and JS where React focuses more heavily on doing all these things within their higher level language (called JSX). Vue has HTML templates and the ability to be integrated into a project gradually without needing a full rewrite.

In addition Vue has more built in features that React leans on third party apps for. Can not be native just yet.



From google, Angular solves similar issues to the above too, allowing you to break down code into components and use templating to generate HTML. It however does not have a virtual DOM and has strong opinions on how you need to structure your code, where react/vue give you some freedom with how to model data and connect it to your view, angular forces you to do it through it's systems.

Transpilers

Since browsers for the time being mostly only run JS many projects have a pipeline where higher level (and often better) languages are transpiled down into JS



Transpiles newer JS into backwards compatible JS



A superset of JS with types that transpiles down to normal JS

SPA/PWA

SPA stands for **single page app** and it's where a website is a single HTML page that uses JS to swap out the entire view rather than doing a fetch for another HTML file. It cuts out slow full page requests entirely

Progressive Web Apps are usually SPA's which are designed to replace standard apps, that is to say a app that works in a way that removes the need for a desktop app or mobile app build in a lower level language.

They can do things like push notifications, work offline etc. But there isn't a strong definition. Regardless they are popular because you have 1 code base for all devices

Web Workers

Web Workers are a simple means for web content to run scripts in background threads. The worker thread can perform tasks without interfering with the user interface.

```
const worker = new Worker('incremento.js');
worker.addEventListener('message', (e) => {
  console.log(e.data)
});
worker.postMessage(1)

// we get 2 printed to the console
```

```
onmessage = function(e) {
  const x = e.data + 1;
  postMessage(x);
}
```

Web Servers in Perl

Wednesday, 31 July 2019 9:19 PM

TCP/IP from Perl Intro

To establish a TCP/IP connection the server running on host `williams.cse.unsw.edu.au` does this:

```
use IO::Socket;
$server = IO::Socket::INET->new(LocalPort => 1234, Listen => SOMAXCONN) or die;
$c = $server->accept();
```

The client running anywhere on the internet does this:

```
use IO::Socket;
$host = "williams.cse.unsw.edu.au";
$c = IO::Socket::INET->new(PeerAddr=>$host, PeerPort=>1234) or die;
```

The `$c` is effectively a bidirectional file handle.

Time Server and Client

A simple TCP/IP server which supplies the current time as an ASCII string.

```
use IO::Socket;
$server = IO::Socket::INET->new(LocalPort => 4242, Listen => SOMAXCONN) or die;
while ($c = $server->accept()) {
    printf "[Connection from %s]\n", $c->peerhost;
    print $c scalar localtime, "\n";
    close $c;
}
```

Simple client which gets the time from the server on `$ARGV[0]` and prints it.

```
use IO::Socket;
$server_host = $ARGV[0] || 'localhost';
$server_port = 4242;
$c = IO::Socket::INET->new(PeerAddr => $server_host, PeerPort => $server_port) or die;
$time = <$c>;
close $c;
print "Time is $time\n";
```

See NTP for how to seriously distribute time across networks.

Well-Known TCP/IP Ports

To connect via TCP/IP you need to know the port. Particular services often listen to a standard TCP/IP port on the machine they are running. E.g

| | |
|-----|--|
| 21 | ftp |
| 22 | ssh (secure shell host) |
| 23 | telnet |
| 25 | SMTP (email) |
| 80 | HTTP (Hypertext Transfer Protocol) |
| 123 | NTP (Network Time Protocol) |
| 443 | HTTPS (Hypertext Transfer Protocol over SSL/TLS) |

Web servers normally listen to port 80 or 443.

Uniform Resource Locator (URL)

Syntax:

```
scheme://domain:port/path?query_string#fragment_id
```

For example:

```
http://en.wikipedia.org/wiki/URI_scheme#Generic_syntax
http://www.google.com.au/search?q=COMP2041&hl=en
```

Given a http URL a web browser extracts the host name from the URL and connects to port 80 (unless another port is specified). It then sends the remainder of the URL to the server.

The HTTP syntax of such a request is simple:

```
GET path HTTP/version
```

Simple Web Client in Perl

A very simple web client does not render the HTML, GUI or anything...

See HTTP::Request::Common for a more general solution

```
use IO::Socket;
foreach $url (@ARGV) {
    $url =~ /http:\/\/([^\s]+)(:([0-9]+))?(.*)/ or die;
    $c = IO::Socket::INET->new(PeerAddr => $1, PeerPort => $2 || 80) or die;
    # send request for web page to server
    print $c "GET $4 HTTP/1.0\n\n";
    # read what the server returns
    my @webpage = <$c>;
    close $c;
    print "GET $url =>\n", @webpage, "\n";
}
```

Notice the web server returns some header lines then data

```
$ cd /home/cs2041/public_html/lec/cgi/examples
$ ./webget.pl http://cgi.cse.unsw.edu.au/
GET http://cgi.cse.unsw.edu.au/ =>
HTTP/1.1 200 OK
Date: Sun, 21 Sep 2014 23:40:41 GMT
Set-Cookie: JSESSIONID=CF09BE9CADA20036D93F39B04329DB
Last-Modified: Sun, 21 Sep 2014 23:40:41 GMT
Content-Type: text/html; charset=UTF-8
Content-Length: 35811
Connection: close

<!DOCTYPE html>
<html lang='en'>
<head>
...
```

Web Server in Perl

This Perl web server just prints details of incoming requests & always returns a 404 (not found) status.

```
use IO::Socket;
$server = IO::Socket::INET->new(LocalPort => 2041,
    ReuseAddr => 1, Listen => SOMAXCONN) or die;
while ($c = $server->accept()) {
    printf "HTTP request from %s =>\n\n", $c->peerhost;
    while ($request_line = <$c>) {
        print "$request_line";
        last if $request_line !~ /\S/;
    }
    print $c "HTTP/1.0 404 This server always 404s\n";
    close $c;
}
```

```
use IO::Socket;
```



```

$server = IO::Socket::INET->new(LocalPort => 2041,
    ReuseAddr => 1, Listen => SOMAXCONN) or die;
$content = "Everything is OK - you love COMP(2041|9044).\n";
while ($c = $server->accept()) {
    printf "HTTP request from %s =>\n\n", $c->peerhost;
    while ($request_line = <$c>) {
        print "$request_line";
        last if $request_line !~ /\S/;
    }
    my $request = <$c>;
    print "Connection from ", $c->peerhost, ": $request";
    $request =~ /^GET (.+) HTTP\/1.[01]\s*$/;
    print "Sending back /home/cs2041/public_html/$1\n";
    open my $f, '<', "/home/cs2041/public_html/$1";
    $content = join "", <$f>;
    print $c "HTTP/1.0 200 OK\nContent-Type: text/html\n";
    print $c "Content-Length: ", length($content), "\n";
    print $c $content;
    close $c;
}

```

Below is a simple webserver in Perl does the fundamental job of serving web pages but it has bugs, security holes and huge limitations.

```

while ($c = $server->accept()) {
    my $request = <$c>;
    $request =~ /^GET (.+) HTTP\/1.[01]\s*$/;
    open F, "</home/cs2041/public_html/$1";
    $content = join "", <F>;
    print $c "HTTP/1.0 200 OK\n";
    print $c "Content-Type: text/html\n";
    print $c "Content-Length: ", length($content), "\n";
    print $c $content;
    close $c;
}

```

mime-types

Web servers typically determine a file's type from its extension (suffix) and pass this back in a header line. On Unix-like system files /etc/mime.types contains lines mapping extensions to mime-types

| | |
|-----------------|----------------|
| application/pdf | pdf |
| image/jpeg | jpeg jpg jpe |
| text/html | html htm shtml |

It may also be configured within a web server e.g. cs2041's .htaccess file contains:

```
AddType text/plain pl py sh c cgi
```

It is easy to read /etc/mime.types specifications into a hash:

```

open MT, '<', "/etc/mime.types") or die;
while ($line = <MT>) {
    $line =~ s/#.*//;
    my ($mime_type, @extensions) = split /\s+/, $line;
    foreach $extension (@extensions) {
        $mime_type{$extension} = $mime_type;
    }
}

```

Here's our previous simple web server with code added to use the mime_type hash to return the appropriate Content-type:


```

$url =~ s/^(^|\/)\.\.(\|/|$/)/g;
my $file = "/home/cs2041/public_html/$url";
# prevent access outside 2041 directory
$file =~ s/^(^|\/)\.\.(\|/|$/)/g;
$file .= "/index.html" if -d $file;
if (open my $f, '<', $file) {
    my ($extension) = $file =~ /\.\.(\w+)/;
    print $c "HTTP/1.0 200 OK\n";
    if ($extension && $mime_type{$extension}) {
        print $c "Content-Type: $mime_type{$extension}\n";
    }
    print $c <my $f>;
}

```

Multi-processing

Our previous web server scripts serve only one request at a time. They cannot handle a high volume of requests. A slow client can deny access for others to the web server e.g. let's make our previous web client sleep for 1 hour

```

$url =~ /http:\/\/([^\|/]+)(:(\d+))?(.*)/ or die;
$c = IO::Socket::INET->new(PeerAddr => $1, PeerPort => $2 || 80) or die;
sleep 3600;
print $c "GET $4 HTTP/1.0\n\n";

```

A simple solution is to process each request in a separate process. The Perl subroutine fork duplicates a running program. It returns 0 in a new process (the child) and process id of a child in the original process (the parent).

We can easily add this to our previous web server:

```

while ($c = $server->accept()) {
    if (fork() != 0) {
        # parent process loops to wait for next request
        close($c);
        next;
    }
    # child processes request
    my $request = <$c>;
    ...
    close $c;
    # child must terminate here otherwise
    # it would compete with parent for requests
    exit 0;
}

```

Simple CGI

Web servers allow dynamic content to be generated via CGI (and other ways). Typically they can be configured to execute programs for certain URIs. For example cs2041's .htaccess file indicates files with suffix .cgi should be executed.

```

<Files *.cgi>
SetHandler application/x-setuid-cgi
</Files>

```

We can add this to our simple web server:

```

if ($url =~ /^(.*\.cgi)(\?(.*))?$/) {
    my $cgi_script = "/home/cs2041/public_html/$1";
    $ENV{SCRIPT_URI} = $1;
    $ENV{QUERY_STRING} = $3 || '';
    $ENV{REQUEST_METHOD} = "GET";
    $ENV{REQUEST_URI} = $url;
}

```

```
print $c "HTTP/1.0 200 OK\n";  
print $c '$cgi_script' if -x $cgi_script;  
close F;  
}
```

See `webserver-cgi.pl` for a fuller CGI implementation of GET and POST requests.

Make

Friday, 2 August 2019 4:02 PM

There are many tools available to control build of software systems. Tools popular with developers often change and are specific to platform/language.

make is a classic and widely used tool. Alternatives include **cmake** and **ninja**.

make

make allows you to

- Document intra-module dependences
- Automatically keep track of changes

make works from a file called **Makefile** or **makefile**.

A Makefile contains a sequence of rules like:

```
target : source1 source2
  commands to create target from source
```

Beware: each command is preceded by a single tab char. Take care when using cut and paste with Makefiles

Dependencies

The make command is based on the notion of **dependencies**. Each rule in a Makefile describes:

- Dependencies between each target and its sources
- Commands to build the target from its sources

make decides that a target needs to be rebuilt if it is older than any of its sources. This is based on file modification times.

Example of a multi-module C program:



Building with incremental compilation:

```
$ gcc -c -g -Wall world.c
$ gcc -c -g -Wall graphics.c
$ gcc -c -g -Wall main.c
$ gcc -Wall -o game main.o world.o graphics.o
```

For systems like the Linux kernel with 50,000 files, building is either:

- Inefficient (recompile everything after a change)
- Error-prone (recompile just what's changed + dependents)
 - Module relationships are easy to overlook (e.g. `graphics.c` depends on a typedef in `world.h`)

- You may not know when a module changes (e.g. you work on graphics.c and others work on world.c_

Example Makefile for the program above:

```
game : main.o graphics.o world.o
    gcc -Wall -o game main.o graphics.o world.o

main.o : main.c graphics.h world.h
    gcc -c main.c

graphics.o : graphics.c world.h
    gcc -c -g -Wall graphics.c

world.o : world.c
    gcc -c -g -Wall world.c
```

This can be easily parsed in Perl:

```
open my $makefile, '<', $file or die;
while (<$makefile>) {
    my ($target, $depends) = /(\S+)\s*:\s*(.*)/
    or next;
    $first_target = $target if !defined $first_target;
    $depends{$target} = $depends;
    while (<$makefile>) {
        last if !/^t/;
        $build_cmd{$target} .= $_;
    }
}
```

How make Works

The make command behaves as:

make(target, sources, command):

```
# Stage 1
FOR each S in sources DO
    rebuild S if it needs rebuilding
END
# Stage 2
IF (no sources OR any source is newer than target) THEN
    run command to rebuild target
END
```

An implementation of make in Perl:

```

my ($target) = @_ ;
my $build_cmd = $build_cmd{$target};
die "*** No rule to make target $target\n" if
    !$build_cmd && !-e $target;
return if !$build_cmd;
my $target_build_needed = ! -e $target;
foreach $dep (split /\s+/, $depends{$target}) {
    build $dep;
    $target_build_needed ||= -M $target > -M $dep;
}
return if !$target_build_needed;
print $build_cmd;
system $build_cmd;

```

Additional Functionalities of Makefiles

```

# string-valued variables/macros
CC = gcc
CFLAGS = -g
LDFLAGS = -lm
BINS = main.o graphics.o world.o

# implicit commands, determined by suffix
main.o      : main.c graphics.h world.h
graphics.o  : graphics.c world.h
world.o     : world.c

# pseduo-targets
clean :
    rm -f game main.o graphics.o world.o
    # or ... rm -f game $(BINS)

```

```

# multiple targets with same sources
stats1 stats2 : data1 data2 data3
    perl analyse1.pl data1 data2 data3 > stats1
    perl analyse2.pl data1 data2 data3 > stats2

# creating subsystems via make
parser:
    cd parser && $(MAKE)
    # assumes parser directory has own Makefile

```

Parsing Variables and Comments in Perl

```

open MAKEFILE, $file or die;
while (<MAKEFILE>) {
    s/#.*//;
    s/\$((\w+)\)/$variable{$1}||''/eg;
    if (/^\s*(\w+)\s*=\s*(.*)$/) {
        $variable{$1} = $2;
        next;
    }
    my ($target, $depends) = /(\S+)\s*:\s*(.*)/ or next;
    $first_target = $target if !defined $first_target;
    $depends{$target} = $depends;
    while (<MAKEFILE>) {
        s/\$((\w+)\)/$variable{$1}||''/eg;
        last if !/^t/;
        $build_cmd{$target} .= $_;
    }
}

```

Command-line Arguments

If make arguments are target, it builds just those targets

```

$ make world.o
$ make clean

```

If no arguments are supplied, it builds the first target in the Makefile.

The -n option instructs make to tell what it would do to create the targets, but would not execute any of the commands

Implementation in Perl

```

$makefile_name = "Makefile";
if (@ARGV >= 2 && $ARGV[0] eq "-f") {
    shift @ARGV;
    $makefile_name = shift @ARGV;
}
parse_makefile $makefile_name;
push @ARGV, $first_target if !@ARGV;
build $_ foreach @ARGV;

```

Example of a Makefile #2: a sample Makefile for a simple compiler

```

CC      = gcc
CFLAGS = -Wall -g
OBJS    = main.o lex.o parse.o codegen.o

mycc : $(OBJS)
       $(CC) -o mycc $(OBJS)

main.o : main.c mycc.h lex.h parse.h codegen.h
       $(CC) $(CFLAGS) -c main.c

lex.o : lex.c mycc.h lex.h
       $(CC) $(CFLAGS) -c lex.c

parse.o : parse.c mycc.h parse.h lex.h
codegen.o : codegen.h mycc.h codegen.h parse.h

clean :
       rm -f mycc $(OBJS) core

```

Abbreviations

To simplify writing rules, make provides default abbreviations:

| | |
|-----|--------------------------------|
| \$@ | Full name of target |
| \$* | Name of target, without suffix |
| \$< | Full name of first source |
| \$? | Full name of all newer sources |

```

# one of above rules, re-written
lex.o : lex.c mycc.h lex.h
       $(CC) $(CFLAGS) -c $*.c -o $@
       # or ... $(CC) $(CFLAGS) -c $<< -o $@

# update a library archive
lib.a: foo.o bar.o lose.o win.o
       ar r lib.a $?

```

Implementation in Perl:

```

my %builtin;
$builtin{'@'} = $target;
($builtin{'*'} = $target) =~ s/\.[^\.]*$//;
$builtin{'^'} = $depends{$target};
($builtin{'<'} = $depends{$target}) =~ s/\s.*//;
$build_command =~ s/\$([@*^<])/$builtin{$1}||''/eg;

```

Performance

Saturday, 3 August 2019 11:23 AM

Strategy for developing efficient programs:

1. Design the program well
2. Implement the program well (see "Programming Pearls", "Practice of Programming", etc.)
3. Test the program well
4. Only after you're sure it's working, *measure* performance
5. If (and only if) performance is inadequate, *find* the "hot spots"
6. *Tune* the code to fix these
7. Repeat **measure-analyse-tune** cycle until performance is ok

Rapid development of a prototype may be the best way to discover/asses performance issues.

Two C functions which intialise an array

```
void test0(int x, int y, int a[x][y]) {
    fprintf(stderr, "writing to array i-j order\n");
    for (int i = 0; i < x; i++)
        for (int j = 0; j < y; j++)
            a[i][j] = i+j;
}

void test1(int x, int y, int a[x][y]) {
    fprintf(stderr, "writing to array j-i order\n");
    for (int j = 0; j < y; j++)
        for (int i = 0; i < x; i++)
            a[i][j] = i+j;
}
```

Performance Improvement Example - Caching

Although the looks are almost identical, the first loop runs 20 times faster on a large array

```
$ time ./cachegrind_example 0 32000 32000
allocating a 32000x32000 array = 4096000000 bytes
writing to array i-j order
real    0m0.893s
user    0m0.364s
sys     0m0.524s
$ time ./cachegrind_example 1 32000 32000
allocating a 32000x32000 array = 4096000000 bytes
writing to array j-i order
real    0m15.189s
user    0m14.633s
sys     0m0.528s
```

The tool valgrind is used to detect accesses to uninitialized variables at runtime and it can also give memory caching information.

The memory subsystem is beyond the scope of this course but you can see valgrind explain the performance difference between these loops.

For the first loop D1 miss rate = 24.8%

For the second loop D1 miss rate = 99.9%

Due to the C array memory layout the first loop produces much better caching performance.

Tuning caching performance is important for some application and valgrind makes this much easier


```

$ valgrind '--tool=cachegrind' ./cachegrind_example 0 10000 10000
allocating a 10000x10000 array = 400000000 bytes
writing to array i-j order
==7025==
==7025== I    refs:      225,642,966
==7025== I1  misses:      882
==7025== LLi misses:      875
==7025== I1  miss rate:    0.00%
==7025== LLi miss rate:    0.00%
==7025==
==7025== D    refs:      25,156,289 (93,484 rd + 25,062,805 wr)
==7025== D1  misses:      6,262,957 ( 2,406 rd + 6,260,551 wr)
==7025== LLd misses:      6,252,482 ( 1,982 rd + 6,250,500 wr)
==7025== D1  miss rate:    24.8% ( 2.5% + 24.9% )
==7025== LLd miss rate:    24.8% ( 2.1% + 24.9% )
==7025==
==7025== LL refs:      6,263,839 ( 3,288 rd + 6,260,551 wr)
==7025== LL misses:      6,253,357 ( 2,857 rd + 6,250,500 wr)
==7025== LL miss rate:    2.4% ( 0.0% + 24.9% )

```

Where is execution time being spent?

Typically programs spend most of their execution time in a small part of their code.

This is often quoted as the 90/10 rule (or 80/20 rule or ...):

"90% of the execution time is spent in 10% of the code"

This means that

- most of the code has little impact on overall performance
- small parts of the code account for most execution time

We should clearly concentrate efforts at improving execution speed in the 10% of code which accounts for most of the execution time.

clang -p/gprof

Given the -p flag clang instruments a C program to collect profile information.

When the program executes this data is left in the file gmon.out.

The program gprof analyses this data and produces:

- number of times each function was called
- % of total execution time spent in the function
- average execution time per call to that function
- execution time for this function and its children

Arranged in order from most expensive function down.

It also gives a *call graph*, a list for each function:

- which functions called this function
- which functions were called by this function

Performance Improvement Example - Word Count

The program is slow on large inputs e.g.

```

$ clang -O3 word_frequency0.c -o word_frequency0
$ time word_frequency0 <WarAndPeace.txt >/dev/null
real    0m52.726s
user    0m52.643s
sys     0m0.020s

```

We can instrument the program to collect profiling information and examine it with clang.

```

$ clang -p -g word_frequency0.c -o word_frequency0_profile
$ head -10000 WarAndPeace.txt|word_frequency0_profile >/dev/null
$ gprof word_frequency0_profile
....
%    cumulative    self           self   total
time  seconds  seconds   calls  ms/call  ms/call  name
88.90    0.79    0.79    88335    0.01    0.01   get
 7.88    0.86    0.07     7531    0.01    0.01   put
 2.25    0.88    0.02    80805    0.00    0.00   get_word
 1.13    0.89    0.01         1   10.02   823.90  read_words
 0.00    0.89    0.00         2    0.00    0.00   size
 0.00    0.89    0.00         1    0.00    0.00  create_map
 0.00    0.89    0.00         1    0.00    0.00   keys
 0.00    0.89    0.00         1    0.00    0.00  sort_words
....

```

Examine *get* and we find it traverses a linked list. So replace it with a binary tree and the program runs 200 times faster on War and Peace.

```

$ clang -O3 word_frequency1.c -o word_frequency1
$ time word_frequency1 <WarAndPeace.txt >/dev/null
real    0m0.277s
user    0m0.268s
sys     0m0.008s

```

C was probably not the best choice for the word count program

Shell, Perl and Python are slower, but require a lot less code. So they are faster to write, have less bugs to find and easier to maintain and modify.

```

$ time word_frequency1 <WarAndPeace.txt >/dev/null
real    0m0.277s
user    0m0.268s
sys     0m0.008s
$ time word_frequency.sh <WarAndPeace.txt >/dev/null
real    0m0.564s
user    0m0.584s
sys     0m0.036s
$ time word_frequency.pl <WarAndPeace.txt >/dev/null
real    0m0.643s
user    0m0.632s
sys     0m0.012s
$ time word_frequency.py <WarAndPeace.txt >/dev/null
real    0m1.046s
user    0m0.836s
sys     0m0.024s
$ wc word_frequency*.
286  759 5912 word_frequency1.c
  8   19   82 word_frequency.sh
 11   38  325 word_frequency.py
 14   43  301 word_frequency.pl

```

Performance Improvement Example - cp - read/write

Here is a cp implementation in C using low-level calls to read/write

```

while (1) {
    char c[1];
    int bytes_read = read(in_fd, c, 1);
    if (bytes_read < 0) {
        perror("cp: ");
        exit(1);
    }
    if (bytes_read == 0)
        return;
}

```

```

    int bytes_written = write(out_fd, c, bytes_read);
    if (bytes_written <= 0) {
        perror("cp: ");
        exit(1);
    }
}

```

It is surprisingly slow compared to /bin/cp

```

$ time /bin/cp input_file /dev/null
real    0m0.006s
user    0m0.000s
sys     0m0.004s
$ clang cp0.c -o cp0
$ time ./cp0 input_file /dev/null
real    0m6.683s
user    0m0.932s
sys     0m5.740s
$ clang -O3 cp0.c -o cp0
$ time ./cp0 input_file /dev/null
real    0m6.688s
user    0m0.900s
sys     0m5.776s

```

Most of the C program's execution time is spent executing system calls. As a consequence -O3 is no help.

Two system calls for every byte copied is a huge overhead. If we modify the code to buffer its I/O it makes two system calls for every 8192 bytes copied, it should run much faster.

```

while (1) {
    char c[8192];
    int bytes_read = read(in_fd, c, sizeof c);
    if (bytes_read < 0) {
        perror("cp: ");
        exit(1);
    }
    if (bytes_read <= 0)
        return;
    int bytes_written = write(out_fd, c, bytes_read);
    if (bytes_written <= 0) {
        perror("cp: ");
        exit(1);
    }
}

```

```

$ time ./cp1 input_file /dev/null
real    0m0.005s
user    0m0.000s
sys     0m0.008s

```

Using portable stdio library a byte-by-byte loop runs quite fast, because stdio buffers the I/O behind the scenes.

```

while (1) {
    int ch = fgetc(in);
    if (ch == EOF)
        break;
    if (fputc(ch, out) == EOF) {
        perror("");
    }
}

```

```

        exit(1);
    }
}

$ time ./cp2 input_file /dev/null
real    0m0.456s
user    0m0.448s
sys     0m0.008s

```

And with a little more complex code, we get reasonable speed with portability:

```

while (1) {
    if(fgets(input, sizeof input, in) == NULL) {
        break;
    }
    if (fprintf(out, "%s", input) == EOF) {
        fprintf(stderr, "cp:");
        perror("");
        exit(1);
    }
}

$ clang -O3 cp3.c -o cp3
$ time ./cp3 input_file /dev/null
real    0m0.095s
user    0m0.084s
sys     0m0.012s

```

For comparison Perl code which does a copy via an array of lines:

```

die "Usage: cp <src> <dest>\n" if @ARGV != 2;
$in = shift @ARGV;
$out = shift @ARGV;
open IN, '<', $in or die "Cannot open $in: $!\n";
open OUT, '>', $out or die "Cannot open $out: $!\n";
print OUT <IN>;

$ time ./cp4.pl input_file /dev/null
real    0m0.248s
user    0m0.168s
sys     0m0.032s
\end{verbatim}

```

And Perl code which unsets Perl's line terminator variable so a single read return the whole file:

```

die "Usage: cp <infile> <outfile>\n" if @ARGV != 2;
$infile = shift @ARGV;
$outfile = shift @ARGV;
open IN, '<', $infile or die "Cannot open $infile: $!\n";
open OUT, '>', $outfile or die "Cannot open $outfile: $!\n";
undef $/;
print OUT <IN>;

$ time ./cp5.pl input_file /dev/null
real    0m0.029s
user    0m0.008s
sys     0m0.020s

```

Performance Improvement Example - Fibonacci

Here is a simple Perl program to calculate the n-th Fibonacci number:

```

sub fib {
    my ($n) = @_;
    return 1 if $n < 3;
    return fib($n-1) + fib($n-2);
}
printf "fib(%d) = %d\n", $_, fib($_) foreach @ARGV;

```

It becomes slow near n=35.

```

$ time fib0.pl 35
fib(35) = 9227465
real    0m10.776s
user    0m10.729s
sys     0m0.016s

```

We can rewrite the program in C:

```

#include <stdio.h>
int fib(int n) {
    if (n < 3) return 1;
    return fib(n-1) + fib(n-2);
}

int main(int argc, char *argv[]) {
    for (int i = 1; i < argc; i++) {
        int n = atoi(argv[i]);
        printf("fib(%d) = %d\n", n, fib(n));
    }
}

```

It's faster but the program's complexity does not change:

```

$ clang -O3 -o fib0 fib0.c
$ time fib0 45
fib(45) = 1134903170
real    0m4.994s
user    0m4.976s
sys     0m0.004s

```

It is very easy to change already computed results in a Perl hash. This changes the program's complexity from exponential to linear.

```

#!/usr/bin/perl -w
sub fib {
    my ($n) = @_;
    return 1 if $n < 3;
    ${f}{$n} = fib($n-1) + fib($n-2) if !defined ${f}{$n};
    return ${f}{$n};
}
printf "fib(%d) = %d\n", $_, fib($_) foreach @ARGV;

$ time fib1.pl 45
fib(45) = 1134903170
real    0m0.004s
user    0m0.004s
sys     0m0.000s

```

Now for Fibonacci we could also easily change the program to an iterative form which would be linear too. But memorisation is a general technique which can be employed in a variety of situations to improve performance.