CS 246 - Object-Oriented Software Development

Cameron Roopnarine

Last updated: January 28, 2020

Contents

1 2020-01-07

1.1 Linux Shell

Shell: interface to an OS

Graphical shell: click/touch, intuitive

Command line shell: type commands, not intuitive, more powerful

Stephen Bourne (70s): original UNIX shell

History: C shell (`csh`) → Turbo C shell (`tcsh`) → KornShell (`ksh`) → Bourne Again Shell (`bash`)

Check what command line shell: `echo $0`

Go into bash: `bash`

1.2 Linux File System

Directories: files that contain files (called folders in Windows), e.g. `usr` , `share` , `dict` are all directories

Root (literally a backslash) `/` : top directory

Path: location of a file in the file system, e.g. `/usr/share/dict/words`

Absolute path: path that starts at the root directory

Relative path: path relative to a directory

The path `dict/words` relative to `/user/share` is `/usr/share/dict/words`

1.3 Commands

- NAME: `ls` → list directory contents.

  SYNOPSIS: `ls [OPTION]... [FILE]`

  DESCRIPTION: List information about the non-hidden `FILE` s (current directory by default). Hidden files start with a `.`

  `ls -a` or `ls -all` do not ignore entries starting with a `.` ; the `-a` is an argument

- NAME: `pwd` → print name of current/working directory.

  SYNOPSIS: `pwd [OPTION]...`

  DESCRIPTION: Print the full filename of the current working directory.

- NAME: `cd` → change the shell working directory.

  `..` → parent directory

  `.` → current directory

  `-` → previous directory

  `~` → home directory

  `~userid` → `userid` 's directory

2 2020-01-09

It's strongly recommend that you **do not** memorize these commands presented, you should try them out on your own to see what the output is.

- `CTRL + C` → send kill signal
- `CTRL + D` → send EOF (end-of-file)
- NAME: `cat` → concatenate files and print on the standard output

  SYNOPSIS: `cat [OPTION]... [FILE]`

  DESCRIPTION: Concatenate FILE(s) to standard output. With no `FILE`, or when `FILE` is `-`, read standard input.
- `>` → output redirection, overwrites files

  `cat > out.txt` → redirects output produced by `cat` to the file `out.txt`

  `cat t1.txt > t2.txt` → redirects all text from `t1.txt` to `t2.txt`
- `>>` → appends at the end of the file instead of overwriting like `>`
- `<` → input redirection

  `cat < sample.txt` → input redirection, the shell handles this

  `cat sample.txt` → `cat` handles this

  `cat -n < in > out` → `-n` numbers all output lines. Input redirect from file `in` to `cat`, then output redirect with numbered lines to file `out`.

2.1 Linux Streams

- 1. Standard input (stdin)

  keyboard

  use `<` to change to file
- 2. Standard output (stdout)

  terminal

  use `1>` to change to file; the `1` before the `>` is optional

  buffered
- 3. Standard error (stderr)

  terminal

  use `2>` to change to file

  non-buffered

We use the non-buffered stream when we immediately want to output an error so that it does not take extra CPU cycles (extra material).

Within the stream,

`stdin` → program → 1. `stdout` and 2. `stderr`

`prog arg1 < in > out 2>&1` → `&1` is the location of `stdout`, so any errors will be redirected to `stdout`.

2.2 Wildcard Matching

`ls *.txt` → match anything that ends with `.txt`. The shell performs this operation.

globbing pattern

Using single/double quotes will suppress globbing patterns.

`\` is the escape character

Example

Count the number of words in the first 15 lines of `sample.txt`.

Solution.

- `wc -w` → print number of words in entire text
- `head -15 sample.txt` → get only the first 15 lines of `sample.txt`
- `head -15 sample.txt > temp.txt wc -w temp.txt` → doing both, with output in a `temp.txt` file.

What if we didn't want `temp.txt` to be produced? We use Linux pipes.

2.3 Linux Pipe

Connect `stdout` of prog1 to `stdin` of prog2.

`head -15 sample.txt | wc -w` → the first program, `head` runs with `sample.txt`, then the output is fed into the second program, `wc`.

Example

Suppose `words*.txt` contains one word per line. Produce a list of words sorted, with no duplicates from `words*.txt`.

Solution.

`cat words*.txt | sort -u` OR `cat words*.txt | sort | uniq` → `sort -u` will sort and remove any duplicate words. `uniq` removes duplicates.

`echo Today is $(date)` → `$(date)` is embedding a command date

Double quotes: does not supresses embedded commands

Single quotes: supresses embedded commands

3 2020-01-14

3.1 Searching Text

- NAME: `grep`

  SYNOPSIS: `grep [OPTIONS] PATTERN [FILE...]`

  DESCRIPTION: `grep` searches for `PATTERN` in each `FILE`. A `FILE` of "-" stands for standard input. If no `FILE` is given, recursive searches examine the working directory, and non-recursive searches read standard input. By default, `grep` prints the matching lines. In addition, the variant programs `egrep`, `fgrep` and `rgrep` are the same as `grep -E`, `grep -F`, and `grep -r`, respectively. These variants are deprecated, but are provided for backward compatibility.

`PATTERN` examples:

- outputs on `stdout` lines that contain a match for the pattern
- case sensitive
- $\underbrace{|}_{\text{“choice”}}$ → OR

  `"cs246|CS246"` → `cs246` or `CS246` or possibly both
- `\` → “escape” special characters
- factor stuff

  `"cs246|CS246"` ⟺ `"(cs|CS)246"`
- `"a|b|c|d"` ⟺ `"[abcd]"` → choose 1 character from this set
- `^` → negation

  `"[^abcd]"` → 1 character *not* from this set.

  `"CS24[^6]"` → anything character except the `6` after `CS24`
- within square brackets, characters don't have their typical meanings
- `?` → 0 or 1 occurrences of the proceeding subexpression

  `"CS ?246"` → `CS246` or `CS 246`

  `"(CS)?246"` → `CS` is optional
- `*` → 0 or more of the proceeding subexpression

  `"CS *246"` → `CS246`, $\texttt{CS}\underbrace{\quad}_{n}\texttt{246}$, $n \geq 0$
- `+` → 1 or more occurrences

  `"(CS)+246"` → $\underbrace{\texttt{CS}}_{n}\texttt{246}$, $n \geq 1$
- `.` → any 1 character
- `.*` → any number of any character

  `"CS.*246"` → lines that contain substrings that contain `CS` and end with `246`
- `^` → match beginning of line

  `"^CS246"` lines that start with `CS246`
- `$` → match ending of line

  `CS246$` → lines that end with `CS246`

  `^CS246$` → lines that *only* contain `CS246`
- words in `dict` that begin with `e` and have length `5`

  ```
  egrep "^e(.){4}" /usr/share/dict/words
  ```
- words in `dict` that have even length

  ```
  egrep "^(..)*$" /usr/share/dict/words
  ```

- files in current directory that have exactly one `a` in their name

```
ls | egrep "^[^a]*a[^a]*$"
```

3.2 File Permissions

- `ls -l` → long listing
- `ls -la` → long listing with hidden files

When above commands are run, in the first column there will be a sequence of 10 characters.

d	rwx	r-x	r–

- `d` → directory
- `r` → read
- `w` → write
- `x` → execute
- Box 2: `usr` bits, owner permissions
- Box 3: group bits
- Box 4: other bits

The owner can change perms with `chmod`.

`chmod MODE FILEs`

`MODE` has three subcategories:

1. ownership
   - `u` → `usr`
   - `g` → `group`
   - `o` → `other`
   - `a` → `all`
2. operator
   - `+` → add permission(s)
   - `=` → set exact permission(s)
   - `-` → remove permission(s)
3. permissions
   - `r` → read
   - `w` → write
   - `x` → execute

Examples of `chmod`:

- `chmod g-x 1201`
- `chmod a=rx file` → set all read, execute access, take away write; there is a implicit `-w` here

- `chmod u+x shellscript`

  shortcut: `chmod 744`, in binary they are corresponding to the box[2,4] above: `111 100 100`

`umask` → default permissions of a file

3.3 Shell Variables

`x=5` → sets variable `x` to `5`; can't have spaces

`echo ${x}` → prints out value of `x`; curly braces are good

Shell variables hold strings.

`dir=$(pwd)` → `dir` holds `pwd`'s value now

`$PATH` → special variable; to append stuff to `PATH` we can do `PATH=newpath:$PATH`

3.4 Shell Scripts

Text file containing Linux commands executed as a program. See `1201/lectures/shell/scripts` for some examples of shell scripts.

File: `basic`

```
#!/bin/bash
date
whoami
pwd
```

- `#!` → Shebang
- `chmod a+x basic` → gives permission to execute `basic`
- `./basic` → executes basic

3.5 Summary of Files

Files covered in this lecture found in `1201/lectures/shell/scripts`:

- `basic`

4 2020-01-16

4.1 Shell Scripts

Shell scripts: text files containing Linux commands executed as a program. Information to a program: **arguments**, `stdin`

We can provide arguments to a script. Arguments are available in special variables named `$1`, `$2`,...

File: `isItAWord`

```
#!/bin/bash
egrep "^$1$" /usr/share/dict/words
```

- `./isItAWord hello` → finds `hello` in `/usr/share/dict/words`

Every process sets a status code: 0 a success, non-zero for failure. `$?` → last status code

Run: `[ 1 -eq 2]` `echo $?` → returns 0 because $1 \neq 2$

File: `goodPassword`

```
#!/bin/bash
# Answers whether a word is in the dictionary (and therefore not a good
#  password)

egrep "^$1$" /usr/share/dict/words > /dev/null

if [ $? -eq 0 ]; then
    echo Not a good password
else
    echo Maybe a good password
fi
```

- `/dev/null` → equivalent to discarding output

`if` statement:

```
if [  ]; then
    ...
elif [  ]; then
    ...
else
    ...
fi
```

`while` loop:

```
while [  ]; do
    ...
done
```

File: `goodPasswordCheck` → same as `goodPassword`, but checks for the correct number of arguments by adding the following (exits with a non-zero code if incorrect number of arguments are supplied):

```
if [ ${#} -ne 1 ]; then
    echo "Usage: $0 password" >&2
    exit 1
fi
```

- `${#}` → number of arguments to the script

File: `count`

```
#!/bin/bash
# count limit ---counts the numbers from 1 to limit

x=1
while [ $x -le $1 ]; do
    echo $x
    x=$((x + 1))
done
```

- `./count 10` → prints out numbers 1 to 10, each on a new line
- `$((x+1))` is proper addition for `int` data type

Run: `x=1`

`echo $((x+1))` → outputs `2`

`echo $x+1` → outputs `1+1`

File: `renameC`

```
#!/bin/bash
# Renames all .C files to .cc

for name in *.C; do
    mv ${name} ${name%C}cc
done
```

- given a `file`, `mv ${file} ${file%C}cc` → renames `file.C` to `file.cc` → removes `C`, adds `cc`; that is, anything after `%` is removed

Files: `countWords`, `payday`

4.2 Summary of Files

Files covered in this lecture found in `1201/lectures/shell/scripts`:

- `basic`
- `isItAWord`
- `goodPassword`
- `goodPasswordCheck`
- `renameC`
- `count`
- `countWords`
- `payday`

5 2020-01-21

5.1 Testing

A pizza shop allows users to order pizza online and earn 10 points for each pizza ordered.

Ordering: A user types `O` followed by a number N to order N pizzas. e.g. `O` 2 orders 2 pizzas

The system allows ordering between 1 to 10 pizzas. If `N` is outside this range, the system prints "Illegal order".

On a successful order, the system display "2 pizzas ordered" followed by the total number of points they have.

Redeeming: At any time, users can type `R` to try to redeem free pizza. If the user has enough points(50), "Free Pizza!" is printed. If the user does not have enough points, "No pizza for you!" is printed followed by the number of points the user has.

Points: At any time, users can type `P` to print their current points balance.

Write exhaustive tests for this system.

Solution.

- `O 1` → 1 pizza ordered
- `O 10` → 10 pizzas ordered
- `O 0` → Illegal order
- `O 11` → Illegal order
- `1` → X
- `O 1 1` → X
- `O 5` → 5 pizzas ordered
  - `P` → 50
  - `R` → Free pizza!
  - `P` → 0
- `O 7` → 7 pizzas ordered
  - `P` → 70
  - `R` → Free pizza!
  - `R` → No free pizza for you! 20

5.2 C++ Introduction

Simula 64 → first OO language (has classes)

C with classes → C++

History: C++99 → C++03 → C++11 → C++14

In C,

```c
# include <stdio.h>
int main(void) {
    printf("Hello world\n");
    return 0;
}
```

File: `hello.cc`

```c
# include <iostream>
using namespace std;

int main() {
    cout << "Hello world" << endl;
    return 0;
}
```

5.3 iostream header

`stdio.h`, `printf`, `scanf`, `read` → not allowed in C++ (although valid)

Instead use, `std::cout << data1 << data2;`

By placing `using namespace std;`, we can say

- `cout` instead of `std::cout`
- `endl` instead of `std::endl`

5.4 Compile C++

Since we created an alias for `g++` in assignment 0, we can instead compile with simply `g++14 hello.cc`. To rename the output file we can specify the `-o` parameter.

- `g++ -std=c++14 hello.cc` → creates `a.out`
- `g++14 hello.cc -o prog.out` → creates `prog.out`

5.5 Run C++

- `./a.out` → runs `a.out`

5.6 C++ I/O

`cout << "Hello" << "World" << endl;`

When we create `iostream`, we get access to three stream variables.

1. `stdin`

   `std::in`

   type: `istream`

   e.g. `cin << x;`

2. `stdout`

   `std::out`

   type: `ostream`

   e.g. `cout << "Hello";`

3. `stderr`

   `std::err`

   e.g. `cerr << "Error";`

File: `add.cc`

```
#include <iostream>
using namespace std;

int main() {
  int x, y;
  cin >> x >> y;
  cout << x + y << endl;
```

```
}
```

- If a read fails, the expression `cin.fail()` is true
- If a read fails due to EOF, then expressions `cin.fail()` and `cin.eof()` are both true

File: `readInts.cc`

```
#include <iostream>
using namespace std;

int main() {
  int i;
  while (true) {
    cin >> i;
    if (cin.fail()) break;
    cout << i << endl;
  }
}
```

- Read as many `int`s from `stdin` and output to `stdout`
- Stop if a read fails
- C++: an automatic conversion from iostream variables to bool.
- `cin` is true if `cin.fail()` is false
- `cin` is false if `cin.fail()` is true

5.7 Summary of Files

Files covered in this lecture found in `1201/lectures/c++`:

- `hello.cc`
- `add.cc`
- `readInts.cc`

6 2020-01-23

6.1 C++ I/O

`cin >> x >> y;` → `cin >> y;` → `cin;`

If a read fails, all subsequent attempts to read fail, unless you acknowledge that failure.

Read and print all ints from `stdin`. Terminate on EOF. Ignore "bad input" (non-int).

File: `readInts5.cc`

```
#include <iostream>
using namespace std;

int main () {
    int i;
```

```
        while (true) {
            if (!(cin >> i)) {
                if (cin.eof()) {
                    break; // nothing left to read
                } else { // read failed, infinite loop without below
                    cin.clear(); // set flag to false
                    cin.ignore(); // ignore next character
                } else {
                    cout << i << endl;
                }
            }
        }
    }
```

6.2 C++ Strings

In C, we used null-terminated character arrays. In C++, we have a string type, header: `<string>`

`string str = "hello";` → creates a null terminated string in both C and C++

C++ strings automatically resize.

	C	C++
comparisons	`strcmp(s1,s2)`	`s1 == s2`
length	`strlen(s)`	`s.length`
concatenation	`strcat(s1,s2)`	`s1 + s2`

When we do `s1 + s2`, we create a new string. If we wanted to append s1 to s2, we can do `s1 = s1 + s2`.

File: `readStrings.cc`

- reads until first whitespace (ignore all whitespace until first character)

`getline(cin,s)` → reads until a new line

In C, we used `%d`, `%x`, etc. for `printf`. In C++, we can use the `<iomanip>` header as follows:

```
    int x = 24;
    cout << x; // prints 24 in decimal
    cout << hex; // switch cout to hexadecimal
    cout << x; // prints 24 in hexadecimal
    cout << dec; // switch cout to decimal
```

`<fstream>` → input file stream

`<ofstream>` → output file stream

File: `fileInput.cc`

```
    #include <iostream>
    #include <fstream>
    using namespace std;

    int main () {
        ifstream file{"suite.txt"};
        string s;
```

```
    while (file >> s) {
        cout << s << endl;
    }
}
```

`ifstream file {"suite.txt"}` → uniform initialization syntax, only in ≥ C++11

(`ifstream`: type; `file`: variable; `{"suite.txt"}`: initialization of variable)

Other examples of uniform initialization syntax:

- `int x{5};`
- `string s{"hello"};`

`<sstream>`

- `istringstream`
- `ostringstream`

File: `buildString.cc`, `getNum.cc`

6.3 Summary of Files

Files covered in this lecture found in `1201/lectures/c++/2-io`:

- `readInts5.cc`
- `readStrings.cc`
- `fileInput.cc`
- `buildString.cc`
- `getNum.cc`

7 2020-01-28

7.1 Short C++ Topics

File: `readInts5.cc`

- Read as many inputs from stdin, int output, bad int ignore
- Terminate when done receiving input

File: `readIntsSS.cc`

```
#include <iostream>
#include <sstream>
using namespace std;

int main () {
  string s;
  while (cin >> s) {
    istringstream ss{s};
    int n;
    if (ss >> n) cout << n << endl;
```

```
    }
}
```

- Difference: ignores entire string if read fails, e.g. `he1lo` would ignore `helo`, but `readIntsSS.cc` discards entire string

7.2 Default Arguments

```
void print (string name = "suite.txt" ) { // set default value
    string s;
    ifstream f{ name };
    while (file >> s) {
        cout << s << endl;
    }
    print("filename");
    print(); // default value
}
```

- Default arguments must come last, e.g. `void test(int x = 0, string str);` → illegal because `string` has no default value but comes last
- `void test(int x = 0, string str = "hello");` → legal, can be called in three different ways as denoted "legal" below.
- When calling the function, we can use default values for the last N parameters, e.g. →

  `test();` → legal;

  `test(s);` → legal;

  `test(s, "bla");` → legal;

  `test( ,"bla")` → illegal;

  `test("bla");` → illegal;

- We cannot implement `void test();`, or `void test(int);`, or `void test(int, string)`
- In C, function names must be unique. In C++, we can have functions with the same name, but they must differ in the number of types of parameters, called **function overloading**.
- **Signature**: name of function, types and number of parameters.
- The return type of a function is not part of the signature.
- A new function: `void test(int, int);` → legal, does not conflict with `test(int, string);`
- A new function: `void test(string);` → legal

```
int a = 21; // 10101
int b = 3;
a = a << b; // 10101000
```

- left shift operator (bit shift): multiply by 2^b;
- right sift operator: divide by 2^b

- `<<` is a overloaded operator. Overloading is a type of function Overloading.
- C++ allows us to define the meaning of operators for user-define types (all types not built-in); `istream` → `iostream` header.

```
int x;
cout << x;
string s{"hello"};
cout << s;
```

- these operators are **overloaded**, because `<<` can work differently for `string` and `int`
- example of function overloading since `int x` and `string s` call different functions

```
int a, b;
a + b;
string c, d;
c + d;
```

- operators are overloaded as `+` differs depending on the type

7.3 Structs

In C,

```
struct Node {
    int data;
    struct Node *next;
};
struct Node n = {3, NULL};
```

- In C++, you don't have to write `struct` before `Node` after defining `Node` as a struct
- Can remove the `=` before {} for uniform initialization
- Discouraged to use `NULL` constant, use `nullptr` instead

7.4 Constants

`const int 17AX = 10;`

- Constants must always be initialized
- `const Node n{3, nullptr}` → `3` data, `nullptr` next
- `n.data` or `n.next` → illegal

```
int n = 5;
const int *p = &n;
```

- `p` is a pointer to an `int` which is `const`; pointer is not `constant`

  `p = &m;` → legal

  `*p = 10;` → illegal, but can still do `n = 10;`

- `int *const q = &n;` → `q` is a `const` ptr to an `int`

  `q = &m;` → illegal

  `*q = 10;` → legal

- `const int *const r = & n;` → r is a `const int` to a ptr that is `const`
- `const` applies to the "thing" on the left, unless there is nothing to the left, in which case it applies to the right

7.5 Parameter Passing

```
void inc(int x) { // copied passed by value
    x = x + 1
}
int x = 5;
inc(x); // does not actually modify x
cout << x; // prints 5
```

Passing pointers:

```
void inc(int *p) {
    *p = *p + 1;
}
int x = 5;
inc(&x);
cout << x; // prints 6
```

`scanf("%d, &x");`	`cin >> x;`
	`operator >> (cin, x);` pass by reference

7.6 Lvalue References

Informally, an **lvalue** is anything that can appear on the left hand side of an assignment.

- `x = 5;` → `x` is an lvalue
- `5 = 7` → not an lvalue
- `x + y = 5` → not an lvalue
- `str[i] = '3';` → `str[i]` lvalue

Formally, an **lvalue** is a storage location, something whose addresses we can obtain.

```
int y = 10;
int &z = y;
```

- `z` is an lvalue reference to `y`
- `z` acts as a constant pointer to `y` with **automatic dereferencing**
- `z = 20;` → automatically dereferencing, don't need `*z` (actually a compile error); changes `y` to `20`

- `z` becomes an alias to `y`
- `int *p = &z` → gets `y`'s address
- `z` behaves like `y`
- `int &` is a type, not an address

References must be initialized to lvalues.

- `int &z;` → illegal
- `int &z = 3;` → illegal
- `int & = a + b;` → illegal
- Cannot create a pointer to a reference.
- Cannot create a pointer to an array of reference.
- Cannot create a pointer to a reference to a reference.

```
void inc(int &n) { // n is an alias for x
    n = n + 1; // n acts like a constant pointer to x with automatic dereferencing
}
int x = 5;
inc(x); // x is passed by reference
cout << x; // prints 6
```

`cin << x` → `x` is passed by reference