VE280 2021SU Midterm Review Part1

Intro to Linux

Shell

The program that interprets user commands and provides feedbacks is called a *shell*. Users interact with the computer through the *shell*.

The general syntax for shell is `executable_file arg1 arg2 arg3 ...`.

- Arguments begin with `-` are called ”switches” or ”options”;
    - one dash `-` switches are called short switches, e. g. `-l`, `-a`. Short switch always uses a single letter and case matters. Multiple short switches can often be specified at once. e. g. `-al` = `-a -l`.
    - Two dashes `--` switches are called long switches, e. g. `--all`, `--block-size=M`. Long switches use whole words other than acronyms.
    - For many programs, long switches have its equivalent short form, e. g. `--help = -h`
    - If you are interested in switches design for your own program, feel free to take a look up `getopt()` and `getopt_long()` in documentation (this will be covered in VE482, EECS 281).

Basic Commands

The following are some basic Linux commands.

Those commands start with a '*' are not required, but maybe useful for you if you decide go on with programming in the future.

- `man <command>`: display the manual for a certain command **(very useful!)**
- `pwd`: print working directory.
- `cd <directory>`: change directory.
    - For example, `cd ../` brings you to your parent directory.
- `ls <directory/file>`: list files and folders under a directory.
    - Argument:
        - If no arguments are given, list working directory (equivalent to `ls .`).
        - If the argument is a directory, list that directory (`ls ~`).
        - If the argument is a file, show information of that specific file (`ls p1.cpp`).
    - Optional arguments:
        - `-a` List hidden files as well. Leading dot means “hidden”.
        - `-l` List files in long format.
- `mkdir <directory-name>`: make directory.
- `rm <file>`: remove.
    - This is an extremely dangerous command. See the famous bumblebee accident.
    - Optional arguments:
        - `-i`: prompt user before removal.

        - `-r` Deletes files/folders recursively. Folders requires this option, e. g. (`rm -r testDir/`)
        - `-f` Force remove. Ignores warnings.
- `rmdir <directory>`: remove directory.
    - Only empty directories can be removed successfully.
- `touch <filename>` : create a new empty file.
- `cp <source> <dest>`: copy.
    - Takes 2 arguments: `source` and `dest`.
    - Be very careful if both source and destination are existing folders.
    - `-r` Copy files/folders recursively. Folders requires this option.
- `mv <source> <dest>`: move.
    - Takes 2 arguments: `source` and `dest`.
    - Be very careful if both source and destination are existing folders.
    - Can be used for **rename** by making `source` = `dest`.
- `cat <file1> <file2> ...` : concatenate.
    - Takes one or multiple arguments, **concatenate** and print their complete content one by one to `stdout`.
- `less <file1> <file2> ...`: display the content of the files
    - "Less is a program similar to more (1), but it has many more features. Less does not have to read the entire input file before starting, so with large input files it starts up faster than text editors like vi (1)."
    - quit `less`: press `q`
    - go to the end of the file: press `G` (`shift+g`)
    - go to the beginning: press `g`
    - search: press `/`, then enter the thing to be searched, press `n` for the next match, press `N` for the previous match
- `diff`: compare the difference between 2 files.
    - `-y` Side by side view;
    - `-w` Ignore white spaces.
- `nano` and `gedit`: basic command line file editor.
    - Advanced editors like `vim` and `emacs` can be used also.
    - If you try `vim`: just in case you get stuck in this beginner-unfriendly editor...the way to exit `vim` is to press `ESC` and type `:q!`.
- *`grep`: global search regular expression and print out the line. Often used together with pipeline `|`. Try `CTRL+F` or `COMMAND+F` to search for `Linux` within this doc in text editor. And try `cat VE280_2021SU_MID_PART1.md | grep Linux`. It will be further discussed in `VE482`, `VE472`.
- *`git`: a free and open source distributed version control system. You can use it to handle your VE280 code or work in teams later. Used in `VE482`,`VE477`,`VE472`.
- *`top`, `free`,`iostat`: monitor the cpu usage, memory usage, disk usage of your Linux system. Quite same as Windows Task Manager.

IO Redirection

Most command line programs can accept inputs from standard input and display their results on the standard output.

- `executable < input` Use input as `stdin` of executable.

- `executable > output` Write the `stdout` of executable into output.
    - Note this command always truncates the file.
    - File will be created if it is not already there.
- `executable >> output` Append the `stdout` of executable into output.
    - File will be created if it is not already there.
- `exe1 | exe2` Pipe. Use the `stdout` of exe1 as the `stdin` of exe2.

They can be used in one command line.

Q: Consider `executable < input > output`. What is this line for?

Linux Filesystem

Directories in Linux are organized as a tree. Consider the following example:

```
/                        //root
├── home/                //users's files
    ├── username1
    ├── username2
    ├── username3
    └── ...
├── usr/                 //Unix system resources
    ├── lib
    └── ...
├── dev/                 //devices
├── bin/                 //binaries
├── etc/                 //configuration files
├── var/
└── ...
```

There are some special characters for directories.

- root directory: `/`
    - The top most directory in Linux filesystem.
- home directory: `~`
    - Linux is multi-user. The home directory is where you can store all your personal information, files, login scripts.
    - In Linux, it is equivalent to `/home/<username>`.
- current directory: `.`
- parent directory: `..`

File Permissions

The general syntax for long format is `<permission> <link> <user> <group> <file_size> <modified_time> <file_name>`.

```
dr-xr-xr-x 1 jess jess    4096 Mar 19  2019  diagnostics
---------- 1 jess jess   11433 Jun  9 00:25  diagwrn.xml
dr-xr-xr-x 1 jess jess    4096 Jun  9 00:12  en-US
-r-xr-xr-x 2 jess jess 4625184 Aug 14 01:17  explorer.exe
-r-xr-xr-x 2 jess jess   18432 Mar 19  2019  hh.exe
-r-xr-xr-x 2 jess jess   43131 Mar 19  2019  mib.bin
-r-xr-xr-x 3 jess jess  181248 Jun  9 00:15  notepad.exe
```

```
-r-xr-xr-x 2 jess jess  358400 Mar 19  2019  regedit.exe
dr-xr-xr-x 1 jess jess    4096 Mar 19  2019  rescache
dr-xr-xr-x 1 jess jess    4096 Mar 19  2019  schemas
dr-xr-xr-x 1 jess jess    4096 Mar 19  2019  security
dr-xr-xr-x 1 jess jess    4096 Aug 14 01:51  servicing
-r-xr-xr-x 1 jess jess    1333 Sep  4 13:11  setupact.log
-r-xr-xr-x 1 jess jess       0 Jun  9 00:21  setuperr.log
-r-xr-xr-x 2 jess jess  165376 Jul 16 06:16  splwow64.exe
-r-xr-xr-x 1 jess jess     219 Sep 15  2018  system.ini
drwxrwxrwx 1 jess jess    4096 Mar 19  2019  tracing
dr-xr-xr-x 1 jess jess    4096 Jun  9 00:17  twain_32
```

In total, 10 characters:

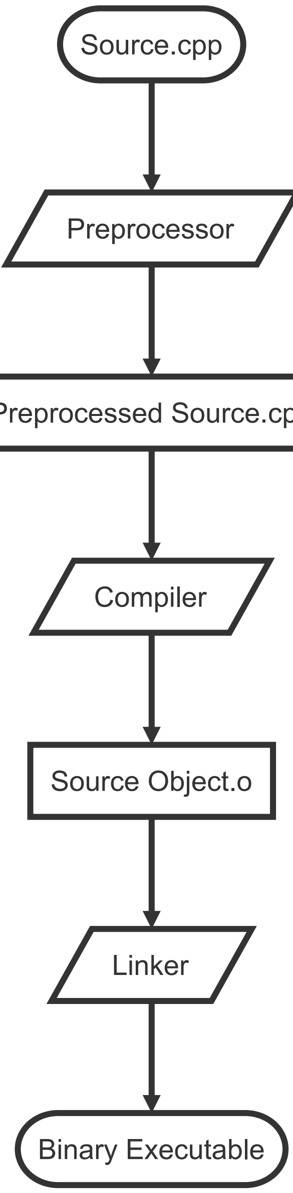
- char 1: Type. `-` for regular file and `d` for directory.
- char 2-4: Owner permission. `r` for read, `w` for write, `x` for execute.
- char 5-7: Group permission. `r` for read, `w` for write, `x` for execute.
- char 8-10: Permission for everyone else. `r` for read, `w` for write, `x` for execute.

Give executable permission to a file: `chmod +x <filename>`.

Compilation Process

For now just have a boarder picture of what's going on. Details will be discussed in the upper level courses.

- **Preprocessing** in `g++` is purely textual.
  - `#include` simply copy the content
  - Conditional compilation (`#ifdef`, `#ifndef`, `#else`, ...) directives simply deletes unused branch.
- **Compiler**: Compiles the `.c`/`.cpp` file into object code.
  - Details of this part will be discussed in UM EECS 483, Compiler Structure. Many CE students with research interest in this field also take EECS 583, Advanced Compiler.
- **Linker**: Links object files into an executable.
  - Details of this part will be discussed in JI VE 482 / UM EECS 370, Computer Organization.



`g++`

Preprocessor, compiler and linker used to be separate. Now `g++` combines them into one, thus is an all-in-one tool. By default, `g++` takes source files and generate executable. You can perform individual steps with options.

Compile in one command: `g++ -o program source.cpp`.

In steps:

- Compile: `g++ -c source.cpp`;
- Link: `g++ -o program source.o`.

Some options for g++:

- `-o <out>` Name the output file as out. Outputs a.out if not present.
- `-std=` Specify C++ standard.
- `-Wall` Report all warnings. Do turn `-Wall` on during tests. **Warnings are bugs.**
- `-O{0123}` Optimization level.
- `-c` Only compiles the file (Can not take multiple arguments).
- `-E` Only pre-processes the file (Can not take multiple arguments).

Header Guard

Everything in C++ should be defined for at most once during compilation.

What will happen during preprocessing if `point.h` is included in both `square.cpp` and `circle.cpp`?

```
// point.h

struct Point {
    double x, y;
};
```

That’s why we need a *header guard*.

```
// point.h
#ifndef POINT_H
#define POINT_H

struct Point {
    double x, y;
};

#endif
```

Note: Be careful when naming you header. Double check if an included library shares the same name with your header file. This may lead to unexpected errors when using header guards, which are extremely hard to detect.

Build

Why do we need a build system?

- Build process is complicated, avoid type every command.
- Project have dependence, need to manage dependence
- Compile minimum amount of code possible upon update.
- Many other reasons, abstract out actual compiler, compile for different platform / target.

GNU Make and Makefile

Makefile is made up of *targets*. A *target* can depend on other *target*, or some files.

```
target: prerequisites
    recipe # <- actual tab character, not spaces!```
```

Here's an example.

```
all: main

main: main.o add.o minus.o
    g++ -o main main.o add.o minus.o

main.o: main.cpp
    g++ -c main.cpp

```

```
add.o: add.cpp
	g++ -c add.cpp

minus.o: minus.cpp
	g++ -c minus.cpp

clean:
	rm -f main *.o

.PHONY: all clean
```

Try to run with `make`, `make all` and `make clean` on your own system and observe what's going on in your working directory.

Or...this below share almost the same output. **But this is only for you to understand it**. **Not recommended** in your project/code.

```
all: main

main: main.cpp add.cpp minus.cpp
	g++ -o main main.cpp add.cpp minus.cpp
```

A reference website in Chinese: https://seisman.github.io/how-to-write-makefile/introduction.html

C++ Basics

Value Category

L/R-value refers to memory location which identifies an object. A simplified definition for beginners are as follows.

- L-value may appear as **either left hand or right hand** side of an assignment operator(=).
    - In memory point of view, an l-value, also called a *locator value*, represents an object that occupies some identifiable location in memory (i.e. has an address).
    - Any non-constant variable is lval.
- R-value may appear as **only right hand** side of an assignment operator(=).
    - Exclusively defined against L-values.
    - Any constant is an rval.

Consider the following code.

```cpp
int main(){
    int arr[5] = {0, 1, 2, 3, 4};
    int *ptr1 = &arr[0];
    int *ptr2 = &(arr[0]+arr[1]);
    cout << ptr1 << endl;
    cout << ptr2 << endl;
    arr[0] = 5;
    arr[0] + arr[1] = 5;
}
```

The compiler will raise errors.

```
...\main.cpp:7:32: error: lvalue required as unary '&' operand
     int *ptr2 = &(arr[0]+arr[1]);
                                 ^
...\main.cpp:11:21: error: lvalue required as left operand of assignment
     arr[0] + arr[1] = 5;
```

Function Declaration and Definition

Consider the following codes.

```
// Function Declaration
int getArea(int length, int width);

int main()
{
    cout << getArea(2, 5) << endl;
}

// Function Definition
int getArea(int length, int width)
{
    return length*width;
}
```

Function Declaration

Function declaration (prototype) shows how the function is called. It must appear in the code before function can be called.

A Function declaration `int getArea(int length, int width);` tells you about:

- Type signature:
    - Return type: `int`;
    - # arguments: 2;
    - Types of arguments: `int`*2;
- Name of the function: `getArea`;
- Formal Parameter Names (*): `length` and `width`.

However, formal parameter names are not necessary. Try to replace:

```
int getArea(int length, int width);
```

with:

```
int getArea(int l, int w);
```

or even:

```
int getArea(int, int);
```

The program still works. Yet, it is considered good coding style to keep the formal parameter names, so that your potential collaborators can understand what the function is for.

Function Definition

Function definition describes how a function performs its tasks. It can appear in the code before or after function can be called.

```
int getArea(int length, int width)  // ----> Function Header
{                                   // -+
    return length*width;            //  |--> Function Body
}                                   // -+
```

Argument Passing Mechanism

Consider the following example.

```
void pass_by_val(int x){
    x = 2;
}

void pass_by_ref(int &x){
    x = 2;
}

void mixed(int x, int &y){
    x = 3;
    y = 3;
}

int main(){
    int y = 1;
    int z = 2;
    pass_by_val(y);
    pass_by_ref(z);
    cout << y << " " << z << endl;
    mixed(y, z);
    cout << y << " " << z << endl;
}
```

The output of the above code is

```
1 2
1 3
```

This example demonstrates the 2 argument passing mechanism in C++:

- Pass by Value;
- Pass by Reference.

The difference of above mechanisms can be interpreted from the following aspects:

- Language point of view: reference parameter allows the function to change the input parameter.
- Memory point of view:
  - Pass-by-reference introduce an extra layer of indirect access to the original memory object. In fact, many compilers implement references with pointers.

    - Pass-by-value needs to copy the argument.
    - Can both expensive, in terms of memory and time.

Choosing argument passing methods wisely:

- Pass atomic types by value (`int`, `float`, `char*`...).
    - `char` is 1 byte, `int32` is 4 bytes, `int64` is 8 bytes, and a pointer is 8 bytes (x64 System).
- Pass large compound objects by reference (`struct`, `class`...).
    - For `std` containers, passing by value costs 3 pointers, but passing by reference costs only 1;
    - What about structures/classes you created?
- Imagine this condition where you load a `4GB` movie into one of your structures and you need to modify/read information on this movie. If you pass the argument by value rather than by reference or by pointer, every time you will have to copy this `4GB` things. Terrible.

Arrays and Pointer

Your familiarity of arrays and pointers is assumed in this course. Test yourself with the following examples.

Pointers

What is the output of the following example?

```
int main(){
    int x = 1;
    int y = 2;
    int *p = &x;
    *p = ++y;
    p = &y;
    y = x++;
    cout << x << " " << y << " " << *p << endl;
}
```

Arrays

What is the output of the following example?

```
void increment(int arr[], int size){
    for (int i = 0; i < size; ++i){
        (*(arr + i))++; // correct
        //*(arr + i)++; // wrong
    }
}

int main(){
    int arr[5] = {0, 1, 2, 3, 4};
    increment(arr, 5);
    for (int i = 0; i < 5; ++i){
        cout << *(arr + i);
    }
    cout << endl;
}
```

Two important things to keep in mind here:

- Arrays are naturally passed by reference;
- Conversion formula between arrays and pointers: `*(arr + i) = arr[i]`

References and Pointers

L-values always corresponds to a fixed memory region. This gives rises to a special construct called references.

Your familiarity of **non-constant references** is assumed in this course. Test yourself with the following example. What is the output?

```cpp
int main(){
    int a = 1;
    int b = 5;
    int *p1 = &a;
    int *p2 = &b;
    int &x = a;
    int &y = b;
    p2 = &a;
    x = b;
    a++;
    b--;
    cout << x << " " << y << " " << *p1 << " " << *p2 << endl;
}
```

Non-constant references must be initialized by a variable of the same type, and cannot be rebounded. Try resist the temptation to think reference as an alias of variables, but remember they are alias for the memory region. References must be bind to a memory region when created: there is no way to re-bind of an existing reference.

Structures

Your familiarity of structures is assumed in this course.

```cpp
struct Student{
    // represents a JI student.
    string name;
    string major;
    long long stud_id;
    bool graduated;
};

int main(){
    // Initialize a structrue
    struct Student s = {"martin", "undeclared", 517370910114, false};
    struct Student *ptr = &s;

    // Use . and -> notation to access and update
    cout << s.name << " " << ptr->stud_id << endl;
    s.major = "ece";
    ptr->graduated = true;
}
```

Two facts to take away here:

- `struct` is in fact totally the same as `class`, instead the default is `public`.

Reference

[1] Paul, Weng, Weikang, Qian. VE280 Lecture 1-4.

[2] Ziqiao, Ma. VE280 Midterm Review Slides. 2020SU.

[3] Chujie, Ni. VE280 Midterm Review SLides. 2020FA.