VE280 2021FA Mid RC Part1

L2: Intro to Linux

Shell/Terminal

The program that interprets user commands and provides feedbacks is called a **shell**. Users interact with the computer through the shell. And **Terminal** provides an input and output environment for commands.

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`

Basic Commands

The following are some basic Linux commands.

- `man <command>`: display the manual for a certain command **(very useful!)**
    - Browse the manual using the same commands as for `less`
- `pwd`: print the path of current working directory.
- `cd <directory>`: change directory.
    - For example, `cd ../` brings you to your parent directory.
- `ls <directory>`: list the files under the directory.
    - Argument:
        - If no arguments are given, list the working directory (equivalent to `ls .`).
        - If the argument is a directory, list that directory (`ls <directory>`).
    - Optional arguments:
        - `-a` List hidden files as well. File name with leading dot means "hidden".
        - `-l` List files in long format.
- `mkdir <directory-name>`: make directory.
- `rmdir <directory>`: remove directory.
    - Only empty directories can be removed successfully.
- `touch <filename>` : create a new empty file.
- `rm <file>`: remove the file.
    - This is an extremely dangerous command. See the famous bumblebee accident.
    - Optional arguments:

    - `-i`: prompt user before removal, and put it into `~/.bashrc`
    - `-r` Deletes files/folders recursively. **Folders requires this option**, e. g. (`rm -r testDir/`)
    - `-f` Force remove. Ignores warnings.
- `cp <source> <destination>`: copy.
  - Takes 2 arguments: `source` and `destination`.
  - Be very careful if both source and destination are existing folders (if the destination doesn't exist, it will be created first).
  - `-r` Copy files/folders recursively. **Folders requires this option**.
  - Variations:
    - `cp -r <dir1> <dir2>`: If `dir2` does not exist, copy `dir1` as `dir2`. If `dir2` exists, copy `dir1` inside `dir2`.
    - `cp <file1> <file2>`: copy the content of file1 into file2 (the content of `file2` will be covered and replaced by the content of `file1`).
    - `cp <file1> <dir>`: copy file into a directory.
      - `cp <file1> <file2> <dir>` (copy two files into one directory)
      - `cp file* <dir>` (`*`: wildcard. Can represent any character string, even an empty string!), and this will copy all the files that begin with `file` into the directory.
- `mv <source> <dest>`: move.
  - Takes 2 arguments: `source` and `dest`.
  - Be very careful if both source and destination are existing folders.
  - Variations:
    - `mv <file1> <file2>`: **rename** `file1` as `file2`
    - `mv <file> <dir>`: move `file` into a directory
    - `mv <dir1> <dir2>`: If `dir2` does not exist, **rename** `dir1` as `dir2`. If `dir2` exists, move `dir1` inside `dir2`.
- `cat <file1> <file2> ...` : concatenate.
  - Takes one or multiple arguments, **concatenate** and print their complete content one by one to `stdout`.
  - `cat <file>` Basically print the file content to `stdout`.
- `less <file1> <file2> ...`: display the content of the files
  - `Less` is a program similar to more, 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.
  - 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
  - *multiple files: enter `:n` to view the next file, enter `:p` to view the previous one
- `diff`: compare the difference between 2 files.
  - If files are the same, no output.
  - If there are differences: lines after “<” are from the first file; lines after “>” are from the second file.

    - In a summary line: `c`: change; `a`: add; `d`: delete
    - `-y` Side by side view;
    - `-w` Ignore white spaces (space, tab).
- `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!`. See how do i exit vim.
- Auto completion: type a few characters; then press `Tab`
    - If there is a single match, Linux completes the remaining.
    - If there are multiple matches, hit the second time, Linux shows all the possible candidates.
- `sudo apt-get install <program>`: install a program.
    - `sudo` command: execute command as a superuser, and requires you to type your password.
- `sudo apt-get autoremove <program>`: remove a program.

IO Redirection

Most command line programs can accept inputs from standard input (keyboard by default) and display their results on the standard output (display by default).

- `executable < inputfile` Use inputfile as `stdin` of executable.
- `executable > outputfile` Write the `stdout` of `executable` into outputfile.
    - Note this command always truncates the file.
    - Outputfile will be created if it is not already there.
- `executable >> outputfile` **Append** the `stdout` of executable into outputfile.
    - Outputfile will be created if it is not already there.
- `exe1 | exe2` Pipe. Connects the `stdout` of exe1 to `stdin` of exe2.
    - e. g., `./add < squareofsum.in | ./square > squareofsum.out`

They can be used in one command line. Like `executable < inputfile > outputfile`.

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> <owner> <group> <file_size>(in bytes) <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 for permission syntax:

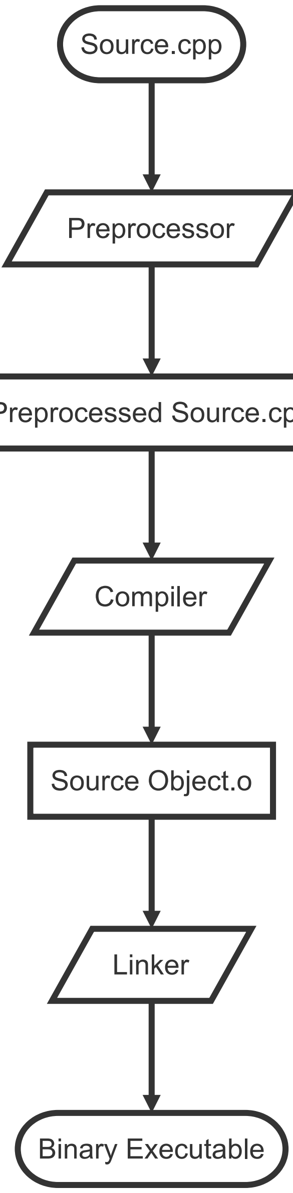
- char 1: Type. `-` for regular file and `d` for directory.
- char 2-4: Owner permission. `r` for read, `w` for write, `x` for execute.
- char 5-7: Group permission. `r` for read, `w` for write, `x` for execute.
- char 8-10: Permission for everyone else. `r` for read, `w` for write, `x` for execute.

L3: Compile a Program

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 370 / 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 source1.cpp source2.cpp`. (header files don't need to be included)

Run the program: `./program`

In steps:

- Compile: `g++ -c source1.cpp`;

  `g++ -c source2.cpp`;
- Link: `g++ -o program source1.o source2.o`.

Some options for g++:

- `-o <out>` Name the output file as . Outputs `a.out` if not present.
- `-std=` Specify C++ standard.
- `-g`: Put debugging information in the executable file
- `-Wall` Report all warnings. **Do turn `-Wall` on during tests**.
- `-c` Only compiles the file (Can not take multiple arguments).
- `-E` Only pre-processes the file (Can not take multiple arguments).

Note: Object code (*.o) is not equivalent to executable code. Object code is **a portion of machine code** that hasn't yet been linked into a complete program.

Multiple Source Files

Two types of files:

(1) header files (.h) : class definitions and function **declarations**

```
//add.h
#ifndef ADD_H
#define ADD_H
int add(int a, int b);
#endif
```

(2) C++ source files (.cpp) : member functions of classes and function **definitions** (or implementations)

```
//add.cpp
int add(int a, int b)
{
    return a+b;
}
```

add.h and add.cpp complete a function `add( )`

If we want to use this function `add( )` in another file (run_add.cpp), we should put `#include "add.h"`:

```
//run_add.cpp
#include "add.h"
int main()
{
    add(2,3);
    return 0;
}
```

Header Guard

```
//add.h
#ifnedf ADD_H // test whether ADD_H has not been defined before
#define ADD_H
int add(int a, int b);
#endif
```

Notes: If `ADD_H` has not been defined before, `#ifndef` succeeds and all lines up to #endif are processed. Otherwise, `#ifndef` fails and all lines between `#ifndef` and `#endif` are ignored.

What will happen for the following two header files, with/without header guard in add.h?

my_project1.h

```
#include "add.h"
...
```

my_project2.h

```
#include "add.h"
#include "my_project1.h"
```

Including of a header file more than once may cause multiple definitions of the classes and functions defined in the header file.

With a header guard, we guarantee that the definition in the header is just seen once.

Makefile

```
all:run_add

run_add:run_add.o add.o
    g++ -o run_add run_add.o add.o

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

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

clean:
    rm -f run_add *.o
```

- Rule:

```
Target:Dependency
<Tab>Command
```

Dependency: A list of files that the target depends on.

- Steps to use

1. Write the file and make the file name as **“Makefile”**.
2. Type “**make**” on command-line, then it does the work correspondingly.
3. The command will be issued when the dependency is more recent than target.

- All Target:
  - It is the default target.
  - Its dependency is program name.
  - It has no command.
- Clean Target:
  - Type “**make clean**”, and what will it do according to the makefile shown above?
  - It has no dependency!

L4: 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.

```
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;
```

References and Pointers

What is the output of the following example? (4 3 3)

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

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

What is the output of the following example? (6 4 6 6)

```
int main(){
    int a = 1;
    int b = 5;
    int *p1 = &a;
    int *p2 = &b;
    int &x = a;
    int &y = b;
    p2 = &a;
    x = b;  // b here is treated as r-value, which assign the value of b to x &
a.
    a++;
    b--;
    cout << x << " " << y << " " << *p1 << " " << *p2 << endl;
}
```

Non-constant references must be initialized by a **variable** of the same type (cannot be a constant), 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**. But you can change the object to which a pointer points.

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`;
  - Number of arguments: 2;
  - Types of arguments: `int`*2;
- Name of the function: `getArea`;
- Formal Parameter Names (*): `length` and `width`.

It is considered good coding style to keep the formal parameter names as meaningful ones, 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 1B, `int32` is 4B, `int64` is 8B, and a pointer is 8B (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?

Arrays and Pointer

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}; // Initialize an array
    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]` and `arr == &arr[0]`

Structures

Your familiarity of structures is assumed in this course. `struct` is in fact totally the same as `class`, instead the default is `public`. And for function call, it is better to use pointer to a struct as the argument, instead of passing by value.

```
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;
}
```

Reference

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

[2] Jiayao, Wu. VE280 Midterm Review Slides. 2021SU.

[3] Yu, Pan. VE280 Mid RC Part1. 2020FA.