ECE2800J RC1

1. Basic Linux Command

1.1 `cd`

Go to the directory `<pathname>`.

```
cd <pathname>
```

Special characters for directories:

- Root directory: `/`
- Home directory: `~`
- Current directory: `.`
- Parent directory: `..`

1.2 `ls`

List the files in `<directory>`.

```
ls <directory>
```

Argument options:

- `-l` : list in long format
- `-a` : list all files including the hidden files

Long Format of File Information:

```
-rw-r--r-- 1 jenny_wyj_o jenny_wyj_g 3331 Oct  1 19:36 README.md
drwxr-xr-x 2 jenny_wyj_o jenny_wyj_g 4096 Oct  9 10:50 p1
```

- `-rw-rw-r--` : permission
  - First character: `-` regular file; `d` directory
  - Next three: read, write, execution permission of **the owner**

    - Next three: read, write, execution permission of **the group**
    - Final three: read, write, execution permission of **everyone else**
- `jenny_wyj_o` : owner
- `jenny_wyj_g` : group
- `3331` & `4096`: file size in bytes
- `Oct 1 19:36` & `Oct 9 10:50` : modification time
- `README.md` : file name
- `p1` : directory name

1.3 `mkdir`

Create `<directory>` under `<pathname>`.

```
mkdir <pathname/directory>
```

1.4 `rmdir`

Remove an **empty** `<directory>` under `<pathname>`.

```
rmdir <pathname/directory>
```

1.5 `touch`

Create an **empty** `<file>` under `<pathname>`.

```
touch <pathname/file>
```

1.6 `cp`

Copy files or directories.

```
cp <file1> <file2>                      # copy file1 into file2
cp <file1> <file2> ... <fileN> <dir>    # copy files into directory
cp <ST>*<ED> <directory>                # wildcard copy
cp -r <dir1> <dir2>                     # If dir2 exists, copy dir1 into
```

```
dir2.
                                        # Otherwise, copy dir1 as dir2.
```

Note: `*` is a wildcard that represents any character string (even empty).

1.7 `mv`

Move or rename files and directories.

```
mv <file1> <file2>                    # rename file1 as file2
mv <file> <directory>                 # move file into directory
mv <dir1> <dir2>                      # If dir2 exists, move dir1 inside dir2.
                                      # Otherwise, rename dir1 as dir2.
```

Argument options:

- `-i` : prompt before every removal

1.8 Edit `file`

```
nano <file>                           # "Ctrl+s": save, "Ctrl+x": exit
gedit <file>
```

1.9 Show `file`

```
cat <file>
less <file>                           # Exit with "q"
```

1.10 I/O Redirection

```
>  # redirect from standard output to a file
<  # redirect standard input from a file instead of keyboard
```

1.11 `diff`

Compare two files.

```
diff <file1> <file2>
```

- If `<file1>` and `<file2>` are the same → no output.
- If different → lines after `<` are from `<file1>`, lines after `>` are from `<file2>`.
- In summary line: `c` : change `a` : add `d` : delete

Argument options:

- `-w` : ignore white spaces

2. Developing and Compiling Programs on Linux

2.1 Compile the Program

```
g++ -o <programName> <source.cpp>
```

Argument options:

- `-o` : name of the output file (by defalut, `a.out`)
- `-g` : put debugging information in the exeutable file
- `-Wall` : turn on all warnings
- It is highly recommended to run `g++ -g -Wall -o <programName> <source.cpp>`.

2.2 Run the Program

Run `<programName>` under the `<pathname>`.

```
<pathname>/<programName>
g++ <source.cpp>
./a.out                  # run default .out file under current directory
```

2.3 Compilation Process

```
g++ -c <source.cpp>                 # compile
g++ -o <programName> <source.o>     # link
```

Process Overview:

```
source code (*.cpp) --compile--> object code (*.o) --link--> executable
program
```

2.4 Developing Program on Linux

Multiple Source Files:

- Header files (`.h`): contain **function declarations** and **class definitions**.
- C++ source files (`.cpp`): contain **function definitions** and **class implementations**.

Example:

- `add.h`

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

- `add.cpp`

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

- `run_add.cpp`

```
#include "add.h"
#include <iostream>

int main() {
    std::cout << add(2, 3) << "\n";
    return 0;
}
```

In C++, the preprocessor replaces each `#include` by the contents of the specified file.

Header Guards:

```
#ifndef VAR
#define VAR
...
#endif
```

Used to prevent **multiple definitions** and avoid **reprocessing** when including the same header multiple times.

2.5 Makefile

Make automates compilation.

Example:

```
all: my_executable

my_executable: my_program.o my_class.o
    g++ -std=c++17 -Wall -g -o my_executable my_program.o my_class.o

my_program.o: my_program.cpp
    g++ -std=c++17 -Wall -g -c my_program.cpp

my_class.o: my_class.cpp
    g++ -std=c++17 -Wall -g -c my_class.cpp

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

```
CC := g++
CFLAGS := -Wall -Wextra -O2
DBGFLAGS := -g -O0
TARGET := run_add
SRCS := add.cpp run_add.cpp
OBJS := $(SRCS:.cpp=.o)

.PHONY: all run debug clean

all: $(TARGET)

$(TARGET): $(OBJS)
        $(CC) $(CFLAGS) $^ -o $@
```

```
%.o: %.cpp
	$(CC) $(CFLAGS) -c $< -o $@

run: all
	./$(TARGET)

debug: CFLAGS := $(DBGFLAGS)
debug: clean $(TARGET)

clean:
	rm -f $(OBJS) $(TARGET) a.out out.out
```

Notes:

- Indentation **must be a tab**, not spaces.
- Run `make` in the directory with the Makefile.
- Run `make clean` to remove compiled files.

3. Review of C++ Basics

3.1 lvalue and rvalue

- **lvalue**: can appear on both sides of `=`. (e.g., variables)
- **rvalue**: can appear only on the right side of `=`. (e.g., constants)

```
int a = 1;              // a is lvalue, 1 is rvalue
int b = a;              // b is lvalue, a is lvalue
const int c = a + b;    // c and a+b are rvalues
int *p = &a;            // p is lvalue, &a is rvalue
```

3.2 Function Declaration and Definition

Declaration in `.h` (before function call):

```
// return_type function_name(parameter_list);
int add(int a, int b);
void print(string s);
```

Definition in `.cpp`: (before or after function call)

```
/*
return_type function_name(parameter_list) {
function_body;
}
*/

int add(int a, int b) {
    return a + b;
}

void print(string s) {
    cout << s << endl;
}
```

3.3 Pointers, References, and Arrays

Pointers store the address of another variable.

```
int a = 1;
int *p = &a;         // p is a pointer to a, &a is the address of a
cout << *p << endl; // *p is the value of a
*p = 2;              // a is now 2
```

- `*`: the dereference operator, which returns the value of the variable that the pointer points to.
- `&`: the address of operator, which returns the address of a variable.

References are alias of variables.

```
int a = 1;
int &r = a;          // & is the reference operator, r is the reference to a
r = 2;               // a is now 2
```

- A reference must be initialized when it is declared.
- After initialization, a reference cannot be changed to refer to another variable.

Arrays are collections of same-type variables.

```cpp
int a[5] = {1, 2, 3, 4, 5}; // a is now an array of 5 integers
cout << a[0] << endl;       // 1
```

- The size of an array must be known at compile time.
- You can access the elements of an array using `[]` operator by pointers.

3.4 Function Call Mechanism

- **Pass by value:** copy of the variable. Modifying the parameter inside the function doesn't affect the original variable.
- **Pass by reference:** Modifying the parameter inside the function affects the original variable.
    - With pointers: you can change the value of the variable that the pointer points to. This sometimes avoid copying instances of large objects.
    - With references: readable and intuitive.

In C++, arrays are passed by reference.

```cpp
void f(int a[]) {
    a[0] = 1;     // pass by reference, a[0] is modified
}
```

3.5 Structs

Structs are user-defined data types that can contain multiple variables of different types. The members of a stryct can be common data types or other structs.

```cpp
struct Student {
    string name;
    int id;
    double gpa;
};  // caution the semicolon here

// initialization way 1
Student s1;
s1.name = "Alice";
s1.id = 52337091;
s1.gpa = 4.0;

// initialization way 2
Student s2 = {"Bob", 52337092, 3.9};
Student *s3 = &s1;  // s3 is a pointer to s1
s3->name = "Bob";   // use pointer to access members of structs
```

4. Exercises

4.1 Median of Two Sorted Arrays

Given two **sorted** arrays `nums1` and `nums2` of size `m` and `n` respectively, return the **median** of the two sorted arrays.

Example 1:

- **Input**: `nums1 = [1,3]`, `nums2 = [2]`
- **Output**: 2.00000
- **Explanation**: merged array = [1,2,3] and median is 2.

Example 2:

- **Input**: `nums1 = [1,2]`, `nums2 = [3,4]`
- **Output**: 2.50000
- **Explanation**: merged array = [1,2,3,4] and median is (2+3) / 2=2.5.

Constraints:

- `nums1.length == m`
- `nums2.length == n`
- `0 ≤ m ≤ 1000`
- `0 ≤ n ≤ 1000`
- `1 ≤ m+n ≤ 2000`
- `-10^6 ≤ nums1[i], nums2[i] ≤ 10^6`

Code:

```
double findMedianSortedArrays(int* nums1, int nums1Size, int* nums2, int
nums2Size) {
    // To Do...
}
```

4.2 First Missing Positive

Given an unsorted integer array `nums`. Return the **smallest positive integer** that is not present in `nums`.

Example 1:

- **Input**: `nums = [1,2,0]`
- **Output**: 3
- **Explanation**: The numbers in the range [1,2] are all in the array.

Example 2:

- **Input**: `nums = [3,4,-1,1]`
- **Output**: 2
- **Explanation**: 1 is in the array but 2 is missing.

Example 3:

- **Input**: `nums = [7,8,9,11,12]`
- **Output**: 1
- **Explanation**: The smallest positive integer 1 is missing.

Constraints:

- `1 ≤ nums.length ≤ 10^5`
- `-2^31 ≤ nums[i] ≤ 2^31-1`

Code:

```c
int firstMissingPositive (int* nums, int numsSize) {
    // To Do...
}
```