

## 1 Before You Start

- It is mandatory to submit all of the assignments in pairs. It is recommended to find a partner as soon as possible and create a submission group in the submission system. Once the submission deadline has passed, it will not be possible to create submission groups even if you have an approved extension.
- Read the assignment together with your partner and make sure you understand the tasks. Please do not ask questions before you have read the whole assignment.
- Skeleton classes will be provided on the assignment page, you must use these classes as a basis for your project and implement (at least) all the functions that are declared in them.

### KEEP IN MIND

While you are free to develop your project on whatever environment you want, your project will be tested and graded ONLY on a CS LAB UNIX machine. It is your own responsibility to deliver a code that compiles, links and runs on it. Failure to do so will result in a grade 0 to your assignment.

Therefore, it is mandatory that you compile, link and run your assignment on a lab unix machine before submitting it.

We will reject, upfront, any appeal regarding this matter!!

We do not care if it runs on any other Unix/Windows machine.

Please remember, it is unpleasant for us, at least as it is for you, to fail your assignments, just do it the right way.

## 2 Assignment Goals

The objective of this assignment is to design an object-oriented system and gain implementation experience in C++ while using classes, standard data structures and unique C++ properties such as the “Rule of 5”. You will learn how to handle memory in C++ and avoid memory leaks. The resulting program must be as efficient as possible.

## 3 Assignment Definition

In this assignment you will write a C++ program that simulates a naïve hierarchical computer file system.

To communicate with the file system, you will implement a command-line/terminal that enables the user to create/delete files/folders, print the contents of a directory etc.

The file system consists of a root directory (“/”) and optionally other files and directories underneath.

The program has a global variable called `verbose` to control its output. See the `verbose` command for details.

Your program will be executed in the following way: `“./bin/fs”`. Your program must not receive any external arguments.

### 3.1 Important terms

- ❖ **Root directory** – The first or top-most directory in the file hierarchy (“/”).
- ❖ **Working directory** – Current directory, see relative path for its purpose.
- ❖ **Absolute path** - The path to a specific directory/file starting from the root directory (“/”). E.g. `“/dir1/dir2/file1”`
- ❖ **Relative path** – The path to a specific directory/file starting from working directory. E.g. if the working directory is `“/dir1/dir2”`, the relative path `“dir3”` is the absolute path `“/dir1/dir2/dir3”`.
- ❖ `..` – The parent directory of the working directory. This is a relative path by definition. E.g. `../.. /dir1` means to look for `dir1` under the parent of the parent of the working directory.

- ❖ **Path** – Can be either absolute path or relative path.

### 3.2 General instructions

- ❖ Class with resources must implement the rule of five.
- ❖ The names of files/directories consist of letters (a-z,A-Z) and digits only (0-9), in particular no spaces are allowed.
- ❖ File/Directory names are unique within the containing directory.
- ❖ You may not add constructors nor global variables nor any public/protected data members. See section 4 for more details.
- ❖ You may not add any global variables.
- ❖ It is highly recommended that you first implement the objects of Files.h before implementing any of the commands.h objects.

### 3.3 Classes

**BaseFile** – This is an abstract class for File and Directory.

**File** – Inherits from BaseFile. This class represents a single file in the system. Each file has a name, a size and a parent (the containing directory).

**Directory** – Inherits from BaseFile. This class represents a single folder in the system. Each directory has a name and a parent and contains a list of files and directories. The parent of the root directory is NULL. The size of a directory is the sum of the sizes of all its files and directories recursively.

**BaseCommand** – This is an abstract class for the different command classes

**Commands implementation** - each of the commands below must be implemented as a class that inherits from BaseCommand. The name of each command's class must be the name of the command (first letter uppercase) followed by "Command".

For example: `"class HistoryCommand: public BaseCommand {...};"`

**ErrorCommand** – This is a special command class that represents an unknown command, that is, a command that is not listed in the commands list below. It also inherits from BaseCommand.

When executed it prints: "<*the-input-command*>: Unknown command"

Example: If user enters "find myfile.txt" it will print "find: Unknown command"

**FileSystem** – Holds the root directory and the working directory.

**Environment** - Holds the file-system and the entire history of executed commands including duplicates, ErrorCommands and the HistoryCommands (see **history** in the command section below)

### 3.4 Commands

Below is the list of the commands you are required to support.

Please be aware for the following denotation:

[..] - These brackets denote that the argument is optional. Example: [-s]

<..> - These brackets denote the expectation of an argument.

Example: <path> should be replaced by any legal path (example: \dir1\dir2).

| - "|" denotes OR. Example: on|off, meaning either on or off.

#### The commands:

- **pwd**: print working directory path.
- **cd**: Change the current directory.  
Syntax: cd <path> - Change current directory to be <path>  
If <path> doesn't exist print out "The system cannot find the path specified"  
Example: cd .. - Change to the parent directory
- **ls**: Display the list of files and subdirectories of a directory. **The list has to be sorted alphabetically by default.** For each file/directory print out in a new line its type ("FILE"/"DIR"), its name and its size (See class Directory for directory size) separated by a tab

(\t).

For example:

DIR	dir1	750
FILE	file1	1000

Syntax: `ls [-s] <path>` - Display the list of files and subdirectories in *<path>*

`ls [-s]` - Display the list of files and subdirectories in the working directory

`-s` – Sort by size, from smaller to larger. If two or more files have the same size, sort them alphabetically.

If *<path>* doesn't exist print out "The system cannot find the path specified"

- **mkdir** – Create a new directory. If needed, create intermediate directories in the path.

Syntax: `mkdir <path>`

If *<path>* exists print out "The directory already exists"

Examples:

"`mkdir /newDir`" - If newDir doesn't exist in the root directory, creates it in the root directory.

"`mkdir newDir`" - If newDir doesn't exist in the working directory, creates it in the working directory.

"`mkdir /d1/d2/d3`" - assume d1 exists and d2 does not exist in d1 directory then it is the same as:

`cd /d1`

`mkdir d2`

`cd d2`

`mkdir d3`

`cd /`

- **mkfile** – Create a new file. The path of the file must exist

Syntax: `mkfile <path/filename> <size>`

If *<path>* doesn't exist print out "The system cannot find the path specified"

If *<path/filename>* exists print out "File already exists"

Example: “mkfile /dir1/dir2/mynewfile 1000” will create mynewfile in /dir1/dir2, the size of the file will be 1000.

- **cp** – Copy a file or directory to a destination.

Syntax: cp <source-path> <destination-path>

*source-path* may be either a file or a directory, *destination-path* is a directory.

If either file/directory/destination doesn't exist print out “No such file or directory”

Example: “cp dir1/dir2/dir3/file1 dir4/dir5” will copy file1 to dir4/dir5

Example: “cp dir4 dir6” will copy dir4 (recursively) under dir6.

- **mv** - Move a file or directory to a new destination.

Syntax: mv <source-path/file-name> <destination-path>

*source-path* may be either a file or a directory, *destination-path* is a directory.

If either source-path/file-name/destination-path doesn't exist print out “No such file or directory”.

Working-directory nor its parents nor the root-directory can't be moved. In such a case print out “Can't move directory”.

Example: “mv dir1/dir2/dir3/file1 dir4/dir5” will move file1 to dir4/dir5

Example: “mv dir4 dir6” will move dir4 to be under dir6.

- **rename** – Rename a file or a directory.

Syntax: rename <path/old-name> <new-name>

If either path/old-name doesn't exist print out “No such file or directory”.

If old-name is the working-directory print out “Can't rename the working directory” and do not rename.

Example: “rename dir1/dir2/file1 file2” will result in “dir1/dir2/file2”.

- **rm** – Remove (delete) a file or a directory. If the argument is a directory remove it recursively.

Syntax: rm <path>

If <path> doesn't exist print out “No such file or directory”.

Working-directory nor the root-directory can't be removed. In such a case print out “Can't remove directory”.

- **history** – Print out the entire list of the executed commands sorted from the oldest to the newest, excluding current history command. It must print all the commands entered by the user including duplicates, ErrorCommands and previous HistoryCommands. Each command will be numbered (0 for the oldest command) and printed in a single row. It will be printed out in the following format: “<index>tab<the command> (as it was entered originally including its full arguments list)”. If the list is empty print nothing.

Syntax: history

Example output:

```
0  ls
1  pwd
```

- **verbose** – Set the verbose variable as follows:

0 – verbose off (i.e. do not print 1, 2 or 3 below).

1 – Print a message in a new line each time entering a rule-of-five function. The message must be the signature of the function in the format:

“return-type Class-Name:: Function-Name(full-argument-list)”

Example: “MyClass &MyClass::operator=(const MyClass &mc)”

2 – Echo/Print the full input command (with its arguments) to the screen followed by a new line.

3 – Execute 1 and 2.

Syntax: verbose <0/1/2/3>

If the argument is different from either 0, 1, 2 or 3, print out: “Wrong verbose input”.

- **exec** – Executes a command from history.

Syntax: exec <command-number>

If <command-number> doesn’t exist print out “Command not found”.

Example: “exec 24” will execute command number 24 as numbered in the history command

### 3.5 The Program flow

Once the program starts, it prints out a prompt - the working directory followed by a “> ” (No new line). Then it waits for the user to enter a command and executes it. After each executed

command it prints again the prompt and waits for the next command in a loop. The program ends when the user enters “exit” at the command line.

You may assume that the number and the type of arguments for each command is legal.

## 4 Provided files

The following files will be provided for you on the assignment homepage:

Commands.h

Environment.h

Files.h

FileSystem.h

GlobalVariables.h

GlobalVariables.cpp

Main.cpp

You are required to implement the supplied functions and to add the Rule-of-five functions as needed.

All the functions that are declared in the provided headers must be implemented correctly, i.e. they should perform their appropriate purpose according to their name and their signature.

**Keep in mind that if a class has resources, ALL 5 rules have to be implemented even if you don't use them in your code. Do not add unnecessary Rule-of-five functions to classes that do not have resources.**

You are **NOT ALLOWED** to modify the signature (the declaration) of any of the supplied functions. We will use these functions to test your code, therefore any attempt to change their declaration might result in a compilation error and a major deduction of your grade. You are also **NOT ALLOWED** to add constructors, nor public/protected data members.



You also **must not** add any global variables to the program. The only allowed global variable is `verbose` which is already defined.

It is strongly recommended to test the correctness of all the public functions using your own external program (another “`main(..)`” function). Our test will include such a testing.

## 5 Examples

See assignment page for input and output examples.

In order to run the examples with your program use the following syntax:

```
./bin/fs < input-file.txt
```

## 6 Submission

- Your submission should be in a single zip file called “`student1ID-student2ID.zip`”. The files in the zip should be set in the following structure:

- `src/`
- `include/`
- `bin/`
- `makefile`

**src/** directory includes all `.cpp` files that are used in the assignment.

**Include/** directory includes the header (`.h` or `*.hpp`) files that are used in the assignment.

**bin/** directory should be empty, no need to submit binary files. It will be used to place the compiled file when checking your work.

- The `makefile` should compile the `cpp` files into the `bin/` folder and create an executable named “`fs`” and place it also in the `bin/` folder.
- Your submission will be build (compile+link) by running the following command: “`make`”.
- Your submission will be tested by running your program with different scenarios.
- Your submission must compile without warnings or errors on the department computers.
- We will test your program using `VALGRIND` in order to ensure no memory leaks have occurred. We will use the following `valgrind` command:  

```
valgrind --leak-check=full --show-reachable=yes ./bin/fs
```

You may also use an input file combined with valgrind like this:

```
valgrind --leak-check=full --show-reachable=yes ./bin/fs < input-file.txt
```

The expected valgrind output is:

definitely lost: 0 bytes in 0 blocks

indirectly lost: 0 bytes in 0 blocks

possibly lost: 0 bytes in 0 blocks

suppressed: 0 bytes in 0 blocks

ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)

We will ignore the following error only:

still reachable: 72,704 bytes in 1 blocks (known issue with std).

**We will not ignore** "still reachable" with different values than **72,704** bytes in **1** blocks

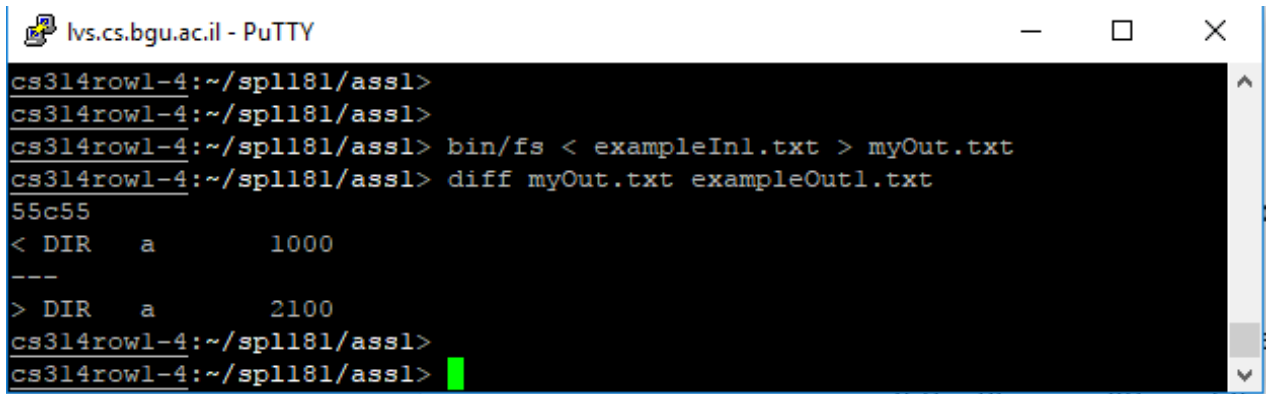
- Compiler commands must include the following flags:

```
-g -Wall -Werror -std=c++11.
```

## 7 Recommendations

1. Be sure to implement the rule-of-five as needed. We will check your code for correctness and performance.
2. It is highly recommended that you compare your program's output with the output examples that we provide for this assignment. In order to do so follow the following steps:
  - a. Open a unix terminal and change directory to your project's root directory.
  - b. Save the example files provided with the assignment to that directory.
  - c. From the prompt run: "bin/fs < in-example-file > your-output-file"  
Example: bin/fs < exampleIn1.txt > myOut1.txt  
(You may want to read about redirecting input and output in the shell)
  - d. From the prompt run the diff command: "diff your-output-file out-example-file"  
(type "man diff" for help)  
Example: diff myOut1.txt exampleOut1.txt

e. If the files are different it will show the difference on the screen



```
lvs.cs.bgu.ac.il - PuTTY
cs314row1-4:~/spl181/ass1>
cs314row1-4:~/spl181/ass1>
cs314row1-4:~/spl181/ass1> bin/fs < exampleIn1.txt > myOut.txt
cs314row1-4:~/spl181/ass1> diff myOut.txt exampleOut1.txt
55c55
< DIR    a          1000
---
> DIR    a          2100
cs314row1-4:~/spl181/ass1>
cs314row1-4:~/spl181/ass1>
```

3. After you submit your file to the submission system, re-download the file that you have just submitted, extract the files and check that it compiles on the university labs. Failure to properly compile or run on the departments' computers will result in a zero grade for the assignment.

בהצלחה