Outlab 8: CMake + GDB

We hope this problem statement will help you dive a bit deeper into the world of build tools and debugging! We have tried to introduce quite a few interesting problems, so enjoy :)

General Instructions

- This outlab carries **100 points**.
- The resources for this outlab can be found under the Resources tab of Piazza as **`outlab8-resources.tar.gz`**
- Any changes to this problem statement will be highlighted in **`yellow.`**
- **NOTE:** This outlab will be **auto-graded**. You **must** use **Docker** for this Outlab and ensure that your output format follows the same standards as the sample outputs and specifications.
- Please make sure you understand what you've written. You might be asked to explain your code at a later point in time.
- **Note: Standard plagiarism rules hold and you must cite any major references you used in the `references.txt` as usual.**
- **The deadline for submitting this outlab is 7th November 23:59. Standard late penalties apply. You'll get a 2 hour leeway beyond 7th November 23:59 with a 1% penalty and no submissions will be accepted after that.**

GDB: The GNU Project Debugger 

Q1: Prettier Debug Outputs [45 Points]

You have already been introduced to GDB in your initial Outlabs, and you must know, we can `print` intermediate data structures/variables in GDB using **`print v`** or **`p v`**(in short).

In the first 2 tasks, we will introduce the Python API for pretty-printing debug structures in GDB. Once you are able to retrieve the structures, you can get the best of both worlds!

Task1: Pretty 2D Matrices

```
Breakpoint 1 at 0x14b0: file q1.cpp, line 28.

Breakpoint 1, main (argc=203, argv=0x7fffffffdd78) at q1.cpp:28
28      GDBBREAKPOINT:
$1 = {{10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26,
8}, {29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 4
, {48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65,
{67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 8
6, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 21, 22, 23, 24, 25}
 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44},
6, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63}, {6
 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82}, {83,
5, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 69, 70}, {71, 7
 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89}}
```

(a) Stock GDB before any pretty printing. Looks unreadable :(

```
Breakpoint 1 at 0x14b0: file q1.cpp, line 28.

Breakpoint 1, main (argc=203, argv=0x7fffffffdd78) at q1.cpp:28
28      GDBBREAKPOINT:
$1 =
ROWS: 10
COLUMNS: 19
10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28
29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47
48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66
67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85
86 87 88 89 90 91 92 93 94 95 96 97 98 99 21 22 23 24 25
26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44
45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63
64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82
83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 69 70
71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89
```

(b) Using a pretty printer we will make. Much better! :)

In this task, we will write a small extension for the stock GDB to pretty print 2D integer Matrices in any interactive GDB session (basically from the figure **(a)** to figure **(b)**). You are given the following resources for this task:

```
Task1/
`-- makefile
`-- t1.gdb
`-- pretmat_skeleton.py
`-- matrix.cpp
```

To run your code: **`make gdb.`** Our goal is to print an array of type **`int[m][n]`**. You can assume and hardcode the type of the 2D matrix for this Task. You must not change the **`matrix.cpp`**. It is

a pretty simple C++ code that takes in a 2D matrix through the command line arguments. It's specified as:

`./matrix m n e11 e12 ... e1n e21 ... e2n ... em1 ... emn`

where **`eij`** are the array elements. You don't need to do anything about this, you can change these arguments specified in the **`makefile,`** to try out other 2D matrices.

Your task: You are given a skeleton pretty printer file **`pretmat_skeleton.py.`** (It is completely optional to use this skeleton file.) You need to use the passed **`gdb.Value`** object and return a well-formatted string in the **`to_string()`** function. You will submit the **`pretmat.py`** for this Task.

[15 Points]

The **`t1.gdb`** file contains a list of commands which are auto-executed by gdb when you do **`make gdb`**. Here in the first command, we **`source`** the **`pretmat.py`** file to activate our pretty-printer (you can remove this line to check the default output).

Required Output Specification: (as specified in the figure**(b)**):

```
\n
ROWS: <m>
COLUMNS: <n>
e11 e12 e13 ... e1n
.
.
.
em1 em2 em3 ... emn
```

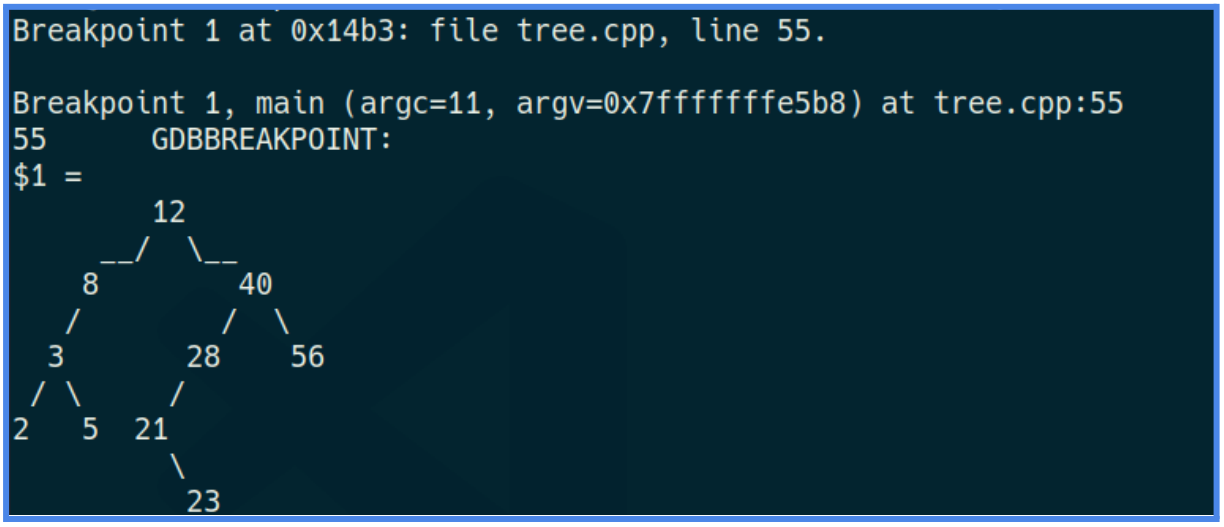
Task2: Pretree printers ;)

In this task, we will write yet another pretty printer, but this time we'll rather transform a useless looking pointer into a meaningful data structure. Say we have a data structure sharing some semantics with a linked list, that is a struct with self-referencing. If we print the root of such a custom made data structure in stock GDB, we'll inadvertently end up with a pointer that doesn't explain anything.

```
Breakpoint 1 at 0x14b3: file tree.cpp, line 55.

Breakpoint 1, main (argc=11, argv=0x7fffffffe5b8) at tree.cpp:55
55      GDBBREAKPOINT:
$1 = {root = 0x55555556aeb0}
```

(a) Pretty useless address pointer, pointing to the `root` of the `tree`.



(b) Same code with our pretty printer on :)

You are given the following resources for this task:

```
Task2/
`-- makefile
`-- t2.gdb
`-- pretree_skeleton.py
`-- tree.cpp
`-- tree.hpp
`-- utils.py
```

To run your code: **`make gdb.`** Our goal is to print a binary tree, specified as a class object in C++. You can assume and hardcode the type of the binary tree, i.e **`tree`** for this Task. You must not change the **`tree.cpp and tree.hpp`**. tree.cpp takes in command-line arguments and constructs a binary tree out of it.

`./tree e1 e2 ... en`

where **`ei`** are the tree node data. You don't need to know about how this tree is created. What's important is that these **`ei`** somehow build up the **`tree b`** in **`line 51`** of **`tree.cpp`** and you can change the arguments specified to **`./tree`** in the **`makefile,`** to try out other examples.

Your task: You are given a skeleton pretty printer file **`pretree_skeleton.py.`** In this problem, the skeleton should be pretty useful, since the printing facility is already built-in for you (the skeleton should be self-explanatory, just take a look!). You need to use the passed **`gdb.Value`** object and return a well-formatted string in the **`to_string()`** function. You will submit the **`pretree.py`** for this Task.

[15 Points]

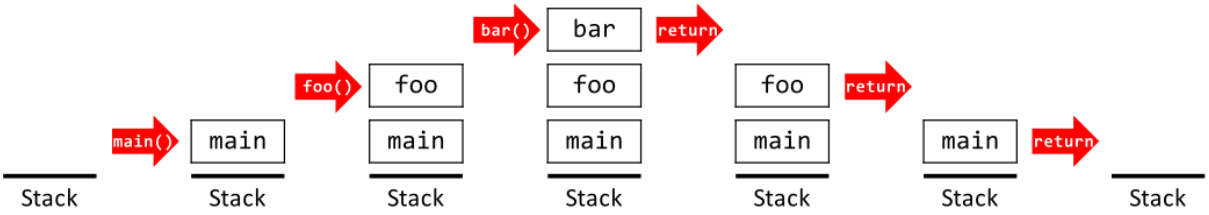
Required Output Specification: (as specified in figure**(b)** and already set up in the skeleton):

You'll need to use the **`printBTree()`** function of **`utils.py`** along with the **`TreeNode class`** of the skeleton to reach the exact output. Don't change the **`utils.py`**.

Task3: Stack visualizer

Although you must have programmed only in high-level languages like C and C++ by now, you certainly must have heard the phrase “local variables are allocated on the stack during a function call and get deallocated when the function returns”. Surprisingly, this notion of scoping of local variables in function calls is built into the hardware of a computer. This task intends to give us a closer look into what happens under the hood by visualizing this process in action using gdb.

Every running program on a computer owns a piece of memory called the “stack”. The “stack” is a special area of a program’s memory that stores temporary variables created by a function. In the stack, variables are declared, stored and initialized during runtime.



A computer’s processor also has “registers” which are just quickly accessible places that hold values. Processors that follow the x86-64 instruction set have 2 special registers named “**rbp**” and “**rsp**”.

- **rbp** (base pointer) stores the address of the base (bottom) of the stack frame of the function currently being executed, while,
- **rsp** (stack pointer) stores the address of the top of the stack frame of the function currently being executed.

[NOTE] As depicted by the above diagram, the stack grows upwards. This means that the addresses of values in the stack frame of the `bar` function will be **lower** than that in the `foo` function. It also implies that **rbp ≥ rsp** almost always.

You may think of the memory of a program as an array of bytes, then, the addresses are just the index or offsets into this array. Shown below is an example of how the stack frame may look like in the memory array :

```
[  . . .   0x23  0x9f  0xa0  0x55  0x6b  0xff  . . .  ]
                       ↑                 ↑
                      rsp               rbp
                   (0x4000)          (0x4003)
```

It is possible to read the registers values and to inspect values stored at any memory location (even the stack) programmatically using gdb-python.

Your Task: You are given a skeleton pretty printer file **`stack.py`**. (Again, it is completely optional to use this file.) Implement a function that gets executed every time the program’s execution is stopped and prints out the stack memory in the format specified below along with the points where **rbp** and **rsp** point to.

[15 Points]

This is what your pretty printer should output when ~~stopped at line 9 in~~ one of the following sets of commands are run in gdb with compiled version (with -g flag) of **`t3_program.c`** (provided in the resources) :

[Added on 26 October] I accidentally missed including t3_program.c in the resources, here is a **link** to the required file : **t3_program.c**

`source stack.py`		`source stack.py`
`break main`		`break main`
`break 9`	**OR**	`step`
`run`		`step`
`continue`		`step`

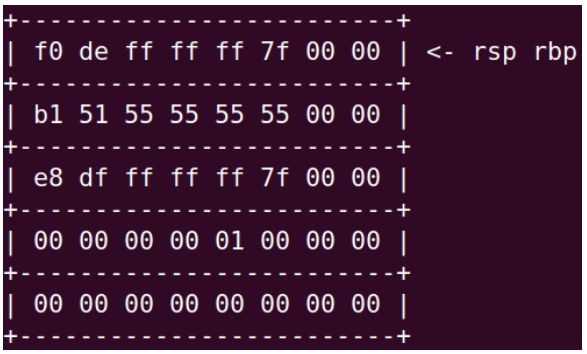
> **Note** that the individual bytes may differ in value, but the overall structure of the output should be similar to what is given below.

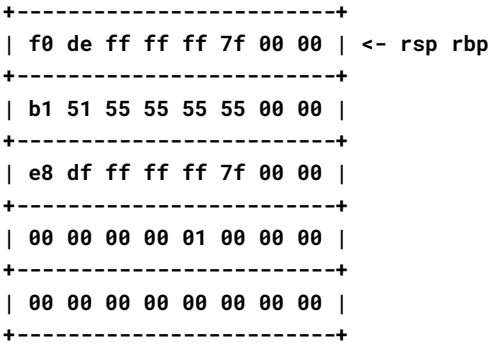
```
+-------------------------+
| b6 e5 ff ff 03 00 00 00 | <- rsp
+-------------------------+
| 02 00 00 00 01 00 00 00 |
+-------------------------+
| c8 df fb f7 ff 7f 00 00 |
+-------------------------+
| c0 51 55 55 55 55 00 00 |
+-------------------------+
| d0 e5 ff ff ff 7f 00 00 | <- rbp
+-------------------------+
| b1 51 55 55 55 55 00 00 |
+-------------------------+
| c8 e6 ff ff ff 7f 00 00 |
+-------------------------+
| 00 00 00 00 01 00 00 00 |
+-------------------------+
| 00 00 00 00 00 00 00 00 |
+-------------------------+
```

```
+-------------------------+
| b6 e5 ff ff ff 7f 00 00 | <- rsp
+-------------------------+
| 0d 52 55 55 55 55 00 00 |
+-------------------------+
| c8 df fb f7 ff 7f 00 00 |
+-------------------------+
| c0 51 55 55 55 55 00 00 |
+-------------------------+
| d0 e5 ff ff ff 7f 00 00 | <- rbp
+-------------------------+
| b1 51 55 55 55 55 00 00 |
+-------------------------+
| c8 e6 ff ff ff 7f 00 00 |
+-------------------------+
| 00 00 00 00 01 00 00 00 |
+-------------------------+
| 00 00 00 00 00 00 00 00 |
+-------------------------+
```

Output Format :

- The first byte of the stack that is printed is the one pointed to by **rsp** (`0xb6` in the above example).
- Other bytes are then sequentially printed, 8 bytes in one block (**rsp+8** points to the first byte in the second block which is `0x02` here).
- The separating line between blocks has 25 hyphens (-) and 2 plus symbols (+).
- All the whitespaces are 1 <space> character wide.
- If **rbp** points to any byte in a particular block "`<- rbp`" is printed after that block. Same rule applies for **rsp**.
- We stop printing at the byte pointed to by the *first valid value* of **rbp** (i.e. when **rbp ≥ rsp**) *among the points where the program was stopped*. If this byte is not the last one in a block, we fill in the block with the next bytes in the sequence.
- When **rbp** and **rsp** point to bytes in the same block, we print it as such **:**





Q2: GG (GDB Games) [25 Points]

This task is divided up into 8 subparts. Each part is supposed to teach you a different way of interacting with the program being debugged with the help of gdb commands.
A nice tutorial to get you started: GDB Tutorial.

You are given the following resources for the task :

```
q2/
|-- init
|-- init.h
|-- makefile
|-- t1.cpp
|-- t2.cpp
|-- t3.cpp
|-- t4.cpp
|-- t5.cpp
|-- t6.cpp
|-- t7.cpp
`-- t8.cpp
```

To work on a subpart / task '**i**' : **`make ti`** (`make t1`, `make t2` ...). This will produce an executable **`ti.out`** (`t1.out`, `t2.out` ...). This executable is what you will use with `gdb`.

`init.h` contains function definitions that are used in various other files. Each function in **`init.h`** (implemented in object file **`init`**) loads a unique "answer" for each subpart in a different way. You may read the function signatures and comments in **`init.h`** to better understand this.
Each **`ti.cpp`** file uses functions defined in **`init.h`** to get a hold of the "answer" for subpart '**i**'. These programs (**`ti.out`**s) do not output anything on the screen when run on the terminal but do get a handle to the "answer" during their execution.

Your task is to use `gdb` with the **`ti.out`** and print the "answer" values in any possible way (you can even change values of variables, program execution flow etc. using `gdb` !!) for each subpart. After you print the "answer", immediately use the command **`quit`** or **`q`** to end the gdb session. For submission, note down the commands you used to print the "answer" and the **`quit`** command in a file named **`ti.gdb`** (`t1.gdb`, `t2.gdb` ...)

[NOTE] You are **not** allowed to call functions from `gdb` in subparts **other than** 7 and 8 (**`t7`**, **`t8`**).

Additional notes :

- To run **`ti.out`** provide your group member's roll numbers (in lexicographic order separated by a hyphen '-') as the first command line argument.
- The "answer" for subpart "**`i`**" is of the form **`parti_<arbitrary number>`** (for example : `part1_65756`, `part2_28222` ...)

Your task: For each subpart / task '**`i`**', create a corresponding file **`ti.gdb`** which when run as follows : "**`gdb ti.out -x ti.gdb`**", eventually prints the "answer" for that subpart and then immediately quits the gdb session.

[25 Points]

Submission format :

```
q2/
|-- t1.gdb   (3 points)
|-- t2.gdb   (3 points)
|-- t3.gdb   (3 points)
|-- t4.gdb   (3 points)
|-- t5.gdb   (3 points)
|-- t6.gdb   (3 points)
|-- t7.gdb   (3 points)
`-- t8.gdb   (4 points)
```

[NOTE] As long as the "answer" is printed somewhere on executing "**`gdb ti.out -x ti.gdb`**" the autograder will give you the appropriate marks. Do not worry about all the other stuff that gets printed too.

CMake 

Q3: Make and CMake walkthrough [30 points]

C was one of the first human-readable programming languages designed for people in all domains (business and scientific community). C is an imperative procedural language

created by Dennis Ritchie. Unix and Linux kernels are built in C (and assembly) language. Due to this C code runs faster than any other language. But still, C was a very low-level language, meaning that bigger **abstractions** were not readily available in it.

To deal with this Bjarne Stroustrup created C++ with objectives of speed and abstraction in mind. The core of his creation was something called a **class.** Thus C++ became one of the first object-oriented languages.

C has a simple architecture and few paradigms which makes it simple to use. Remember the zen of python (*There should be one—and preferably only one—obvious way to do it.*) But due to the sheer flexibility of C++ (with several paradigms it possesses), it became very difficult to maintain standards in bigger C++ projects.

One of the biggest misconceptions in this area is that C and C++ are the same languages. C is not object-oriented while C++ is. Acknowledge the difference.

Today we will learn about compiling & linking C/C++ code in various ways using raw commands, Makefile and CMake. This is one of the ways to gain insights into how exactly things work in the C/C++ world.

Create a Makefile named **rawmake** (can be run using **make -f rawmake**) and fill it up as instructed in the below tasks. Source and Header files are provided in outlab8-resources/q3 directory. Assume that the **rawmake** file will be placed directly under q3/ directory. The dir structure after creating the **rawmake** file will be as follows

```
q3/
├── helloworld.cpp
├── myengine/
│   ├── myengine.cpp
│   └── myengine.hpp
├── mygame/
│   └── mygame.cpp
├── rawmake
└── usespthread.cpp
```

Task 1 [1 point]
Add a rule to compile **helloworld.cpp** to form **helloworld** executable.

Command run while checking: **make -f rawmake helloworld**

Task 2 [2 points]

Add a rule to compile **usespthread.cpp** to form **usespthread** executable. This file uses the pthread library used to create and manage threads. You will be using this a lot in your Operating Systems course.
In order to properly compile this file, you need to special use a flag for g++.

Command run while checking: **make -f rawmake usespthread**

Task 3 **[4 points]**
In this task, we build a library using files **myengine.hpp** and **myengine.cpp.** There are actually two types of libraries categorized based on the way the library is linked to the main file. Dynamic and Static libraries. This and This are good reads on compiling static and dynamic libraries. This is a good read which distinguishes one from the other.

Use the given resources to add rules to make dynamic and static libraries. Dynamic library created should have name **libMyEngineDynamic.so** and similarly static **libMyEngineStatic.a**

Command run while checking: **make -f rawmake libMyEngineDynamic.so** **[2 points]** Command run while checking: **make -f rawmake libMyEngineStatic.a** **[2 points]**

Task 4 **[4 points]**
In this task, we try to install the above-built libraries into the system (requires **sudo** in general, but you're **root** by default inside docker. See the note below.).

Add a PHONY rule named **installdynamic** which installs dynamic version (.so file) to **/usr/local/lib/** and corresponding headers to **/usr/local/include/**

Do the same for the static version (**.a** file) by creating a rule **installstatic**

Command run while checking: **make -f rawmake installstatic** **[2 points]** Command run while checking: **make -f rawmake installdynamic** **[2 points]**

These rules should have previously created libraries as dependencies so that they are built first (if not already built) and then installed.

Task 5 **[2 points]**
Now we use the installed libraries in **mygame.cpp** Add a rule to compile **mygame.cpp** with the static library installed in the previous task and produce binary name **mygamestatic**.

Command run while checking: **make -f rawmake mygamestatic**

Task 6 **[2 points]**
Add a rule to compile **mygame.cpp** with the dynamic library installed in the previous task and produce binary name **mygamedynamic**.

Command run while checking: **make -f rawmake mygamedynamic**

Task 7 [2 points]
Also, add a PHONY rule to **clean** all the generated intermediates and binaries (**.o .a .so** and other binaries[i.e. including the final executables])

Command run while checking: **make -f rawmake clean**

Task 8 [9 points]
Now your job is to do the same tasks using **CMake**.

To do this you need to create a **CMakeLists.txt** file (directly under **q3/** directory) which generates a Makefile to build and install targets in the same manner as above **tasks 1 - 4**

1. **helloworld** binary from **helloworld.cpp** [1 point]

2. **usespthread** binary from **usespthread.cpp** [2 points]

3. **libMyEngineDynamic.so** and **libMyEngineStatic.a** libraries from **myengine.hpp** and **myengine.cpp** [2 + 2 points]

4. Instead of installing two types of libraries separately, it should now install both of the libraries and the corresponding header file **myengine.hpp** to **/usr/local/lib/** and corresponding headers to **/usr/local/include/** respectively. The command used will be **sudo make install** [2 points]

Observe that the makefile that **CMakeLists.txt** creates already contains **PHONY** rule clean which removes all the targets and intermediates generated!

Command run while checking: (from **q3/** dir)

1. **mkdir build**

2. **cd build**

3. **cmake ..**

4. **make** (should create helloworld, usespthread, libMyEngineStatic.a, libMyEngineDynamic.so)

5. **sudo make install** (should install libMyEngineStatic.a, libMyEngineDynamic.so and header into specified paths appropriately)

Note: Ignore the **sudo** for all problems**.** Just make sure it works on docker with **make install.** Don't worry if you haven't noticed this note. It's a pretty late clarification, so we'll take care of the grading if your code works with **sudo make install** but not **make install.**

Task 9 [4 points]
Now we want to do **tasks 5** and **6** using CMake.

Create a **CMakeLists.txt** file in **q3/mygame** directory.

This file should compile **mygame.cpp** with the static library installed in the previous task and produce binary name **mygamestatic** and similarly compile **mygame.cpp** with the dynamic library installed in the previous task and produce binary name **mygamedynamic**

[2 + 2 points]

Command run while checking: (from **mygame/** dir)

1. **mkdir build**
2. **cd build**
3. **cmake ..**
4. **make** (should create **mygamestatic** and **mygamedynamic**)

The dir structure after creating **rawmake** file and **two CMakeLists.txt** files will be as follows

```
q3/
├── CMakeLists.txt                    (For task 8)
├── helloworld.cpp
├── myengine/
│   ├── myengine.cpp
│   └── myengine.hpp
├── mygame/
│   ├── CMakeLists.txt                (For task 9)
│   └── mygame.cpp
├── rawmake                           (For tasks 1-7)
└── usespthread.cpp
```

Make sure the directory structure like shown above and **submit the entire q3/ directory** during submission.

Submission Instructions

- **For this outlab, you'll need to submit the compressed directory:**
  - The final directory structure will look something like this :

```
rollno1-rollno2/
|
`-- q1/
|     |
|     `-- task1/
|     |     `-- pretmat.py
|     |
```

```
|     `-- task2/
|     |     `-- pretree.py
|     |
|     `-- task3/
|           `-- stack.py
`-- q2/
|     |
|     `-- t1.gdb
|     `-- t2.gdb
|     `-- t3.gdb
|     `-- t4.gdb
|     `-- t5.gdb
|     `-- t6.gdb
|     `-- t7.gdb
|     `-- t8.gdb
`-- q3/
      `-- CMakeLists.txt            (For task 8)
      `-- helloworld.cpp
      `-- myengine/
      |   `-- myengine.cpp
      |   `-- myengine.hpp
      `-- mygame/
      |   `-- CMakeLists.txt        (For task 9)
      |   `-- mygame.cpp
      `-- rawmake                   (For tasks 1-7)
      `-- usespthread.cpp
```

Compress your top-level directory **<rollno1>-<rollno2>** using the following command :

tar -czvf <rollno1>-<rollno2>.tar.gz <rollno1>-<rollno2>/

Submit the **<rollno1>-<rollno2>.tar.gz** file generated from the above command on Moodle from the account of the lexicographically smaller roll number **before 7th November 23:59.**