Question #9
What will be the output of this program? (on a standard 64 bits, Linux machine)

```
#include <stdio.h>
#include <stdlib.h>

#define int char

int main(void)
{
int i;

i = 5;
printf ("sizeof(i) = %lu", sizeof(i));
return (EXIT_SUCCESS);
}
```

sizeof(i) = 1

Segmentation Fault

sizeof(i) = 4

sizeof(i) = 5

sizeof(i) = 8

It does not compile

The C program provided has a preprocessor directive that redefines the keyword `int` as `char`:

```
#define int char
```

This means that wherever the `int` keyword is subsequently used, it is replaced with `char` by the preprocessor before the compilation process truly begins.

Therefore, when the program declares `i` as an `int`, it's actually being declared as a `char`:

```
int i;
```

So, in reality, this is what the compiler sees:

```
char i;
```

The `sizeof` operator returns the size in bytes of the type of the operand. Since `i` is a `char`, and the size of a `char` is always 1 byte in C, the `sizeof(i)` will be `1`.

Thus, the `printf` function will print `sizeof(i) = 1`. The output of the program will be:

```
sizeof(i) = 1
```

The choice "sizeof(i) = 1" is correct.

What will be the last 5 lines of the output of the command gcc -E on this code?

#include <stdlib.h>

```
int main(void)
{
NULL;
return (EXIT_SUCCESS);
}

int main(void)
{
0;
return (0);
}

int main(void)
{
'\0';
return (0);
}

int main(void)
{
((void *)0);
return (0);
}

int main()
{
0;
return (0);
}
```

The `gcc -E` command in C is used to run the preprocessor stage of the C compilation process. The preprocessor resolves all the `#include` directives by copying the content of the included files into the source code, and it also processes the macro definitions and expansions.

In the code you've provided, there are no macro definitions or conditional preprocessor directives that would affect the output, except for the inclusion of `stdlib.h` and the use of `NULL` and `EXIT_SUCCESS`.

The `stdlib.h` header file defines `NULL` and `EXIT_SUCCESS`, and those will be replaced in the preprocessed output:

- `NULL` is typically defined as `((void *)0)`.
- `EXIT_SUCCESS` is typically defined as `0` in standard C.

Since `gcc -E` only runs the preprocessor, the function definitions themselves will not be altered by the preprocessor. It's the macro replacements within those functions that will change.

The `NULL;` line in the first `main` function will be replaced with its definition, and the `EXIT_SUCCESS` in the return statement will also be replaced with `0`. All the other `main` functions do not have any macros or preprocessor directives that need to be expanded, so they will remain largely the same, except for the one with `NULL`, which will be expanded as it was in the first instance.

Considering this, the last 5 lines will be from the expansion of the last `main` function defined in the source code, because `gcc -E` will not remove duplicate definitions and will not report them as errors (that would occur in the compilation step, which is not run with the `-E` flag).

However, the preprocessed code would still contain a lot of lines resulting from the inclusion of `stdlib.h`, and these lines can be very numerous. Therefore, the exact last 5 lines cannot be definitively stated without actually running the command, as they will be from the end of the content of `stdlib.h` or the boilerplate code added by the preprocessor, not from the code snippet provided.

If we only consider the snippet provided, disregarding the contents of the included `stdlib.h`, the last five lines would look something like this, with `NULL` and `EXIT_SUCCESS` replaced, and assuming that nothing else in `stdlib.h` affects the last part of the code:

```
int main(void)
{
 ((void *)0);
 return (0);
}

int main()
{
 0;
 return (0);
}
```

But remember, this is a simplification, as the actual output will include the content at the end of the `stdlib.h` file, which can vary depending on the specific implementation of the C standard library and the compiler's preprocessor specifics.

Why should we use include guards in our header files?

Because we said so, and we should never ask why.

To avoid the problem of double inclusion when dealing with the include directive.

The correct and professional reason to use include guards in header files is:

To avoid the problem of double inclusion when dealing with the include directive.

Include guards prevent the same header file from being included multiple times in a single translation unit, which can lead to errors such as multiple definition errors, longer compilation times, and other unintended behaviors. They are a crucial part of writing maintainable and error-free C and C++ code, especially in large projects with multiple files and complex interdependencies.

In C and C++, include guards are usually implemented with preprocessor directives:

```
#ifndef HEADER_FILE_NAME
#define HEADER_FILE_NAME

// content of the header file

#endif // HEADER_FILE_NAME
```

These ensure that the compiler only includes the content of the header file once per translation unit, even if the `#include` directive appears multiple times.

The macro __FILE__ expands to the name of the current input file, in the form of a C string constant.

True

False

True

The macro `__FILE__` in C and C++ is predefined by the preprocessor and expands to the name of the current input file as a string literal. This can be particularly useful for debugging purposes, logging, or generating informative error messages, where you want to include the file name in the output.

#ifndef PI
#define PI 3.14159265359
#endif
in this code why we use
#ifndef PI
#endif

The use of `#ifndef PI` followed by `#define PI 3.14159265359` and `#endif` in C or C++ is a common pattern for creating an "include guard" specifically for a macro definition. This pattern ensures that the macro `PI` is defined only once, even if the header file containing these lines is included multiple times.

Here's what each part does:

- `#ifndef PI` checks if `PI` has not been defined yet. `#ifndef` stands for "if not defined".

- `#define PI 3.14159265359` defines `PI` with the value `3.14159265359` if it has not been defined previously.
- `#endif` ends the conditional preprocessor block.

This technique prevents multiple definitions of the same macro, which could cause compilation errors. It is especially useful in large projects with many files including the same headers to ensure that the compiler knows `PI` should only be defined once. It's a form of ensuring idempotence for header file processing, meaning no matter how many times you include the header, the outcome is the same as if it were included just once.

```
#ifndef MAIN_H
#define MAIN_H

int _putchar(char c);
char *_memset(char *s, char b, unsigned int n);
char *_memcpy(char *dest, char *src, unsigned int n);
char *_strchr(char *s, char c);
unsigned int _strspn(char *s, char *accept);
char *_strstr(char *haystack, char *needle);
void print_chessboard(char (*a)[8]);
void print_diagsums(int *a, int size);
void set_string(char **s, char *to);

#endif
```

and in this code

```
#ifndef MAIN_H
#define MAIN_H


#endif
```

In both code snippets you've provided, `#ifndef MAIN_H`, `#define MAIN_H`, and `#endif` form a set of include guards. Include guards are a common technique in C and C++ to prevent multiple inclusions of the same header file, which can lead to errors due to redefinition of functions, variables, or types.

The purpose of include guards is to check if a unique value (in this case, `MAIN_H`) has already been defined. If not, it defines this value and includes the code between `#define MAIN_H` and `#endif`. If the value has already been defined, it skips including the code, thus preventing duplicate definitions.

Here is a breakdown of the process:

1. `#ifndef MAIN_H` checks if `MAIN_H` has not been defined.
2. `#define MAIN_H` defines `MAIN_H` if it was not defined already.
3. The code between `#define MAIN_H` and `#endif` is included only once, thanks to the include guard.

In the first snippet:

```
#ifndef MAIN_H
#define MAIN_H

// Function declarations here

#endif /* MAIN_H */
```

You have a set of function declarations that are protected by include guards. This means that these function declarations will only be included once, no matter how many times the header file is included in the project.

In the second snippet:

```
#ifndef MAIN_H
#define MAIN_H

// No code here

#endif /* MAIN_H */
```

You have an include guard with no actual content between `#define MAIN_H` and `#endif`. This might be a template for a header file where the function declarations or other definitions have yet to be added. It serves the same purpose as the first snippet in preventing multiple inclusions but currently does not define any functions or other items.

why in the perives we define only pi and now we define all file of main

The inclusion of a guard like `#ifndef PI` around the definition of `PI` serves a different purpose than wrapping an entire header file with `#ifndef MAIN_H`. Let me explain both:

1. **Defining `PI`:** When you define `PI` using:

```
#ifndef PI
#define PI 3.14159265359
#endif
```

The intention is to ensure that `PI` is defined only once across your entire project, to prevent a redefinition error if multiple files include the header file where `PI` is defined. If `PI` has already been defined in one header, another header that attempts to define it again will not do so because of the `#ifndef` guard.

2. **Guarding an entire header file (`MAIN_H`):** In the case of `MAIN_H`, this is a common practice to guard the contents of an entire header file:

```
#ifndef MAIN_H
#define MAIN_H

// Entire header file content...

#endif /* MAIN_H */
```

This prevents multiple inclusions of the same header file from causing errors. If the code inside the header file is included more than once, it could result in redefinitions of structures, functions, and other tokens which is illegal in C/C++. The guard ensures that the header file's contents are processed only once per translation unit, no matter how many times it is included.

The difference between defining `PI` and guarding the `main.h` file is based on their use cases:

- `PI` is a macro that should only be defined once to represent the mathematical constant π. It's a single token, and it's common to prevent its multiple definitions with a guard.
- `MAIN_H` is an include guard pattern that protects an entire header file from being processed multiple times within the same translation unit. This pattern is not about a single token but about the entire collection of declarations and definitions in a header file.

Using these guards is a crucial part of C/C++ programming to avoid compilation errors related to multiple inclusions.

Write a program that prints the name of the file it was compiled from, followed by a new line.

You are allowed to use the standard library
julien@ubuntu:~/0x0c. macro, structures$ gcc -Wall -pedantic -Werror -Wextra -std=gnu89 2-main.c -o c
julien@ubuntu:~/0x0c. macro, structures$ ./c
2-main.c
julien@ubuntu:~/0x0c. macro, structures$ cp 2-main.c 02-main.c
julien@ubuntu:~/0x0c. macro, structures$ gcc -Wall -pedantic -Werror -Wextra -std=gnu89 02-main.c -o cc
julien@ubuntu:~/0x0c. macro, structures$ ./cc
02-main.c
julien@ubuntu:~/0x0c. macro, structures$

explain this

The program you're asking to write should display the name of the file from which it was compiled. In C, you can use the `__FILE__` macro for this purpose. The `__FILE__` macro expands to the name of the current input file, as a string literal. This name is the path by which the preprocessor opened the file, not the short name specified in `#include` or as the input file name argument. When you compile a file, the `__FILE__` macro is replaced with the name of the file being compiled.

Here is a simple C program that accomplishes this:

```c
#include <stdio.h>

int main(void) {
    printf("%s\n", __FILE__);
    return 0;
}
```

When you compile this program with `gcc` and then run it, it prints the name of the file it was compiled from, followed by a new line.

The commands you've provided show the process of compiling the program `2-main.c` into an executable named `c`, running it, then copying `2-main.c` to `02-main.c`, compiling this new file into an executable named `cc`, and running it.

Here's what each command does:

1. `gcc -Wall -pedantic -Werror -Wextra -std=gnu89 2-main.c -o c`
    2. This command compiles `2-main.c` with standard `gnu89`, with all warnings enabled (`-Wall`), treating warnings as errors (`-Werror`), and some extra warnings (`-Wextra`), into an executable named `c`.
3. `./c`
    4. This runs the executable `c`, which prints `2-main.c` because that's the file from which it was compiled.
5. `cp 2-main.c 02-main.c`
    6. This copies `2-main.c` to a new file named `02-main.c`.
7. `gcc -Wall -pedantic -Werror -Wextra -std=gnu89 02-main.c -o cc`
    8. This compiles the newly copied file `02-main.c` into an executable named `cc`, with the same compiler flags as before.
9. `./cc`
    10. Running the executable `cc` now prints `02-main.c`, because that's the file from which this executable was compiled.

Each time you compile a C program, the `__FILE__` macro within the code is replaced by the name of the file being compiled. This is why when you copy the source file and compile it under a new name, the output changes accordingly.