Project in Pairs. Implementation of an application that manages and manipulates files (20 points over 100 from the team work).

Diseño modular.

The design by the Senior engineer defined the following modules. You should follow the following order to implement the modules:

1. `programa.c` (***1 point***): the program itself. It is not a module, it is where the main function is defined. The main function should invoke the rest of the functions exposed by other modules. The error messages should be printed from the main program and never from a modules (you should use the errors returned by the functions).
   Go to the `Referencia` directory from your project repository and execute the file `programa` to learn how it behaves (you should invoke it as `./programa from your terminal`)
2. `senhal` (***1 point***): module devoted to the management and handling of signals.
3. `menu` (***3 points***): module devoted to the management of the user menus.
4. `input_program` (***3 points***): module devoted to the I/O from the user
5. `item` (***2 points***): module devoted to the management of the collection items
6. `util_files` (***1 point***): utilities module (it has a function to obtain info about a file)
7. `collection` (***3 points*** fixed size collection ***or 7 points*** if an implementation of a variable size collection is submitted): module devoted to the management of the collection items
8. `directorio` (***2 points***): module devoted to the reading of directories content

First of all, get familiar with the different functionalities offered by the different modules.
Read the different header files (.h) contained at the directory `Referencia`.
Remember: you are not suposed to modify any .h file. You should only implement the C code in the correspondant C file.
At the directory `Ejemplos_Uso` you will find different directories, each one containing a C file that shows the behaviour of a module. Within each directory you will also find a `README` file that contains instructions about compiling and executing the different programs.

Project versions

In order for the client to see a few versions of the program, we have defined 4 versions: each one encompassing different modules (and one depending on the version of the module `collection` implemented.
As an incentive to constancy, the company has decided to divid the payment between modules. The fee will be if the submitted version complies with the technical and functional requirements in the rubric and if the employee passes an individual exam.

Warning!

The company requires that both pair member work dedicate evenly to the project in order to get paid. In order to verify this, there will be an individual exam after the submission of the project.
The maximum grade at the individual lab exam will depend on how much it is implemented of the project. Over 10 points:

1. Version 1: up to 2 points
2. Version 2: up to 6 points
3. Version 3: up to 9 points
4. Final Version: up to 10 points

The individual payment of the project will depend on the individual performance on the laboratory exam. For each member of the pair who obtains N points over 20 in the project:

- If the mark of the exam laboratory for member A is 0, the project mark for the member A is 0.
- If the mark of the exam for member A is greater than 0, the project mark for member A will be N, no matter the mark for member B.

Version 1 (5 points).Modules: programa, senhal and menu

Important!

Where to submit work:

- This version must be submitted in directory ***version1*** within the pair working directory.

Considerations for grading:

- Submitted code that doesn't compile nor run, won't be considered as valid and their mark will be 0.
- If the submitted code has 1 segmentation fault in a module, the grade of the module will be 0. If the code presents 2 or more segmentation fault errors, the final mark of the ***whole project*** will be 0.
- The final mark of this part will depend on the individual project exam. If its mark is zero, it will invalidate the mark obtained in the project.

Main program (1 point)

First of all, read and understand the examples contained at `PRJ/Ejemplos_Uso/uso_menu` and `PRJ/Ejemplos_Uso/uso_collection`, they will serve you as a guide to start to implement the program.
Now, you should implement the main program:

1. Between steps, you should compile and execute your code to check if it works:

```
$gcc -Wall -g -o mi_programa programa.c *.o
$./mi_programa
```

2 Copy in the directory `version1` all the .o and .h files of the directory Referencia.
3 Create a file named `programa.c`, that will include the `main`
4 Write main function `int main(int argc, char **argv)` in `programa.c`. So that it only calls `print_menu()` and returns.
5 Before invoking `print_menu()` initialize the signal module with a call to `inicialize_signal_handlers()`. You should also initialize the collection invoking `initialize_collection`.
6 Ask a command from the user invoking `ask_comando`
7 Now, you should implement a loop that does not finish until the user wants (when `comando.comando` is equal to EXIT). Remember that you should clean the command using `clear_comando(comando)` in each loop iteration.
8 Before finishing the program (once the loop is finished) you should clean all the memory from the different used modules invoking the functions `clean_module_input_program` and `clean_module_collection`.

senhal Module (1 point)

Now, you should implement the code of the `senhal` module:

1 Create a file named `senhal.c`
2 Implement the function `signal_handlers` to behave as explained in the .h.
3 You should implement the function `inicialize_signal_handlers` that should change the default signal handlers of SIGINT y SIGTSTP to your function `signal_handlers`. In order to do so, search within the man how it is done (`man signal` at the terminal). In the previous activities of this session you will find an example of signal handling.

Delete from the directory `version1` the file `senhal.o`. Compile and execute your code in order to check that it works:

```
$gcc -Wall -g -o mi_programa programa.c senhal.c *.o
$./mi_programa
```

menu Module (3 points)

First of all, read and understand the example contained at `PRJ/Ejemplos_Uso/uso_input_program`, it will serve you as a guide to call to functions from the `input_program` module

1 Create a file named `menu.c`
2 Implement the function `print_menu`
3 In order to check that it works, create a file named `pruebo.c` with a main that invokes your function (this is valid for all the functions that you create, you should check them before the integration with the whole program).

```
$gcc -Wall -g -o pruebo pruebo.c menu.c
$./pruebo
```

4 Implement the function `ask_command`:
   - In this function it is allowed to show error messages.
   - You can implement auxiliary functions as the commented at the bottom of `menu.h`.

- This function (and/or its auxiliary functions) should use `get_number`, `get_error_msg_input` (to get the string describing an error) and `get_string`. So, you should include the library `input_program.h`.
- When the command has an additional string parameter, this function (or its auxiliaries) should copy the additional parameter using `strdup`, or `malloc` and `strcpy`.
- In order to check that it works, create a file named `pruebo.c` with a main that invokes your function:

```
$gcc -Wall -g -o pruebo pruebo.c menu.c
$./pruebo
```

5 Implement the function `clear_comando`. This function should free the memory that was allocated when copying the optional parameter (only done when the optional parameter was a string).

Delete from the directory version1 the file `menu.o`. Compile and execute your code to check that it works:

```
$gcc -Wall -g -o mi_programa programa.c senhal.c menu.c *.o
$./mi_programa
```

Don't forget to add created files to subversion and to do the needed commits.

Version 2 (6 + 3 points).Modules: input_program, item, util_files and collection (static size)

Important!

Where to submit work:

- This version must be submitted in directory ***version2*** within the pair working directory.
- Copy all the files from ***version1*** to ***version2***.

Considerations for grading:

- Submitted code that doesn't compile nor run, won't be considered as valid and their mark will be 0.
- If there are memory leaks (check this with `valgrind`), the mark won't be the maximum.
- If the submitted code has 1 segmentation fault in a module, the grade of the module will be 0. If the code presents 2 or more segmentation fault errors, the final mark of the ***whole project*** will be 0.
- The final mark of this part will depend on the individual project exam. If its mark is zero, it will invalidate the mark obtained in the project.

You can start implementing `util_files`. Then, you can continue with `input_program` or `item`.

`util_files` module (1 point)

Now, you should implement the `util_files` module:

1 Create a file named `util_files.c`. Advise: create also an auxiliary file with a main in order to test your different functions before integrating them within the whole project.

2 First of all, read and understand the examples contained at `PRJ/Ejemplos_Uso/uso_util_files` and `PRJ/Ejemplos_Uso/uso_item`, they will help you to understand the behaviour of the module.
3 The only function implemented by the module, `info_file`, receives a relative path (for example, "."), a filename, a double pointer to char (to store the absolute path) and a pointer to an structure of type `struct stat` (to store the information of the file). The different calls to the operating system will modify `errno`, thus you should take this into account.
4 This function will obtain the absolute path of the file using the functions`strcpy`, `strcat` and `realpath`.
5 Then, it will use the function `lstat` to obtaing the information of the file (it is important to use this function, as it does no resolve the symbolic links).
6 Finally, it will copy in dynamic memory the absolute path of the file.

Delete from the directory `version2` the file `util_files.o`. Compile and execute your code in order to check that it works:

```
$gcc -Wall -g -o mi_programa programa.c senhal.c util_files.c *.o
$./mi_programa
```

Check that your code has no memory leaks using valgrind:

```
$valgrind ./mi_programa
```

input_program Module (3 points)

Now, you should implement the input_program module:

1 Create a file named `input_program.c`. Advise: create also an auxiliary file with a main in order to test your different functions before integrating them within the whole project.
2 Copy within that file the variables of the module (commented out at the beginning of `input_program.h`)
3 Create at the beginning of the file the global variables (internal to the module) `char *cadena = NULL; size_t number_alloc=0;`
4 Implement just after the function `void clean_module_input_program()`. This function should check the value of the global variable `cadena` and if it different from NULL, it will free cadena and set cadena to NULL, and will set to zero `number_alloc`.
5 Implement the function `const char *get_error_msg_input(int error)` as indicated in the comment to the prototype with `input_program.h`
6 Implement the function `char *get_string(int *error)`. You should invoke once the function `getline` over `cadena` and `number_alloc`. If an error happens (`errno` different than 0) you should return NULL and copy the value of errno into `*error`. If the user has introduced CTRL+D, you should set `*error` to `ERR_CTRLD_INPUT` (and invoke before returning to `clean_module_input_program()`).
7 Implement the function `long get_number(int base, int *error)`. This function will invoke to `get_string` and if it returned with no error, it will call `strtol` to transform the string into a `long` (you can find an example of use at the manual, `man strtol` at the console). Remember to return the

appropriate error at *error if an error happens (or a zero if no error happens).

Delete from the directory `version2` the file `input_program.o`. Compile and execute your code in order to check that it works:

```
$gcc -Wall -g -o mi_programa programa.c senhal.c util_files.c input_program.c *.o
$./mi_programa
```

Check that your code has no memory leaks using valgrind:

```
$valgrind ./mi_programa
```

item Module (2 points)

First of all, read and understand the example contained at `PRJ/Ejemplos_Uso/uso_item`.

Now, you should implement the item module:

1. Create a file named `item.c`. Advise: create also an auxiliary file with a main in order to test your different functions before integrating them within the whole project.
2. Implement the function `create_item` that receives the name of the directory where the filename is stored, a filename, its absolute path, its information (stored in an structure of type `struct stat`) and an address to store the error, and returns a variable of type `struct item`.
   This function will copy into dynamic memory (and will allocate memory within it before that) the name of the file, the name of the directory and its absolute path. Additionally, it will store the information abaout the file in the proper files of the `struct item` that will be returned (the size of the file, its modification time, mode, owner, group of the owner, number of hard links and inode number).
   If the file is a symbolic link (check it with the macro `S_ISLNK`), you should obtain the name of its target, using the function `readlink`, store it at dynamic memory and at the proper field of the variable of type `struct item`. If it is not a symbolic link, you should set this field to `NULL`
3. Implement the fucntion `free_item` that frees all the allocated memory and leaves all the fields of the item to NULL. This function does not free the variable item.
4. Implement the function `print_item`.
   In order to print the modification time, you should use the field (that is not defined, but because of the backwards compability) `foo.data.st_mtime`, and, for example, the functions `localtime` and `strftime` (search them into the man)
   In order to check the permission bits you can use bitwise ands (using &) with the permission mode and the different predefined masks (more information here).
   In order to print the file type, use predefined macros (more information here).
5. Implement the comparison functions that will be used later with `qsort`. In the man page of `qsort` you will find an example

Delete from the directory `version2` the file `item.o`. Compile and execute your code in order to check that it works:

```
$gcc -Wall -g -o mi_programa programa.c senhal.c util_files.c input_program.c item.c collection.c *.o
$./mi_programa
```

Check that your code has no memory leaks using valgrind:

```
$valgrind ./mi_programa
```

collection Module (3 points)

The implementation of this module can be of fixed size (3 points) or dynamic size (version 3 of the project). If you implement the dynamic size version it is not mandatory to implement the other one, as the dynamic version is an upgraded version of the dynamic one

The static implementation only adds files if there are space within the array `my_collection.tabla`. When the total capacity (initialised to INI_SIZE) is reached, if a user whats to add a new file, the program will show her/him an error message.

First of all, read and understand the example contained at `PRJ/Ejemplos_Uso/uso_collection`, it will serve you as a guide to call to functions from the `collection` module

Now, you should implement the collection module:

1. Create a file named `collection.c`. Advise: create also an auxiliary file with a main in order to test your different functions before integrating them within the whole project.
2. Copy within that file the variables of the module (commented out at the beginning of `collection.h`)
3. Create at the beginning of the file the global variable (internal to the module) `struct collection my_collection;`
4. Implement just after the function `struct collection initialize_collection()` This function will initialize the value of total_capacity to INI_SIZE and the number of occupied positions to 0.
5. Implementa later the function `struct collection delete_collection(struct collection my_collection)` that frees all the allocated memory within the module, including the allocated strings of each item from the collection.
6. Implement just after the function `void clean_module_collection()` that calls to the former function.
7. Implement the function `const char *get_error_msg_collection(int error)` as indicated in the comment to the prototype with `collection.h`
8. Implement the rest of the functionality. Remember, before adding a file, you should use the function `info_file` that if the file does not exist returns the corresponding error.

Delete from the directory `version2` the file `collection.o`. Compile and execute your code in order to check that it works:

```
$gcc -Wall -g -o mi_programa programa.c senhal.c util_files.c input_program.c item.c collection.c *.o
$./mi_programa
```

Check that your code has no memory leaks using valgrind:

```
$valgrind ./mi_programa
```

Don't forget to add created files to subversion and to do the needed commits.

Version 3 (7 points).Module collection (dynamic size)

Important!

Where to submit work:

- This version must be submitted in directory ***version3*** within the pair working directory.
- Copy all the files from ***version2*** to ***version3***.

Considerations for grading:

- Submitted code that doesn't compile nor run, won't be considered as valid and their mark will be 0.
- If there are memory leaks (check this with `valgrind`), the mark won't be the maximum.
- If the submitted code has 1 segmentation fault in a module, the grade of the module will be 0. If the code presents 2 or more segmentation fault errors, the final mark of the ***whole project*** will be 0.
- The final mark of this part will depend on the individual project exam. If its mark is zero, it will invalidate the mark obtained in the project.

Remember that if you implement the dynamic size version it is not mandatory to implement the other one, as the dynamic version is an upgraded version of the dynamic one

The dynamic implementation of the collection, when the total capacity is reached, this capacity doubles. Thus, if the total capacity is `INI_SIZE` and there are already `INI_SIZE` elements, when the user wants to add one more, the program should double th capacity (the new total capacity would be `INI_SIZE*2`). Later, when there are `INI_SIZE*2` elements and the user wants to add one more, the program should double the capacity to `INI_SIZE*2*2` and so on.

In order to implement the dynamic version of the collection module:

1. Change the definition of `struct collection` to admit a variable `tabla` of dynamic size
2. Change the function `initialize_collection()` in order to create a dynamic table with an initial size of `INI_SIZE`. You can change the value of `INI_SIZE` to a lower one in order to be able of testing better this new version of the module.
3. You should change also the functions `add_file`, `delete_collection` and `clean_module_collection` (minimum).
   Notice that `delete_collection` leaves the collection as it was just initialized while `clean_module_collection` frees all the module memory. Thus, in this version, the behaviour of both functions is different.
4. Maybe you will need to define new errors at the .h file and new error messages.

Compile and execute your code in order to check that it works:

```
$gcc -Wall -g -o mi_programa programa.c senhal.c util_files.c input_program.c item.c collection.c *.o
$./mi_programa
```

Check that your code has no memory leaks using valgrind:

```
$valgrind ./mi_programa
```

Don't forget to add created files to subversion and to do the needed commits.

Final Version (2 points).Module directorio

Important!

Where to submit work:

- This version must be submitted in directory ***versionFinal*** within the pair working directory.
- Copy all the files from ***version3*** to ***versionFinal***.

Considerations for grading:

- Submitted code that doesn't compile nor run, won't be considered as valid and their mark will be 0.
- If there are memory leaks (check this with `valgrind`), the mark won't be the maximum.
- If the submitted code has 1 segmentation fault in a module, the grade of the module will be 0. If the code presents 2 or more segmentation fault errors, the final mark of the ***whole project*** will be 0.
- The final mark of this part will depend on the individual project exam. If its mark is zero, it will invalidate the mark obtained in the project.

First of all, read and understand the example contained at `PRJ/Ejemplos_Uso/uso_directorio`, it will help you to understand this module.
In order to implement directorio module:

1. Implement the function `free_all_filenames_array( char **array, int len)`. The function receives the address of an array of strings and its length. It frees all the strings within the array and, then, it frees all the array.
2. Implement the function `info_all_files_dir`:
   - The function returns the number of read files.
   - This function receives as input parameters: `path`, the name of the directory to read; `info_files`, the position where to store the address where the array of strings will be stored (that is a variable dynamically allocated by the function `scandir`); `error`, the address to store the error, if any.
   - The function should obtain the real path of the directory by doing `realpath`, where real_path is an static array of char of size PATH_MAX (defined at the system library limits.h).
   - The function will invoke `scandir` to obtain the information. At the man page of `scandir` you will find an usage example.

Compile and execute your code in order to check that it works:

```
$gcc -Wall -g -o mi_programa programa.c senhal.c util_files.c input_program.c item.c collection.c directorio.c
$./mi_programa
```

```
$valgrind ./mi_programa
```

Check that your code has no memory leaks using valgrind:
Don't forget to add created files to subversion and to do the needed commits.

2. How to compile the project

The project will be graded by first compiling the program with the following commands:

```
gcc -Wall -g -DDEBUG *.c
```

```
gcc -Wall -g *.c
```

Once finished, the project must have two executables: regular executable for the Intel platform and the executable with the debug option for the Intel platform (see company requirement 4). For the regular development phase, we recommend you use the version for the Intel platform with the debug option (that is, compile with option "**-DDEBUG**").
To avoid to type several times these lines, it is recommended to these two commands in two text files (the command must be in a single line) and execute them from the interpreter with the "**source**" command followed by the name of the file containing the command to execute.

3. Project Evaluation

The following table shows the guide used to evaluate the project submissions. Those categories that occupy two cells are evaluated with the highest score. If a project falls into a red cell, its score will be 0.

Aspect	Excellent (100%)	Good (75%)	Fair (50%)	Poor (0%)
1 Code compilation with the option `-Wall`	The code compiles with no warning no error.		The code compiles with two warnings.	The code does not compile, or it does with more than two errors.
2 Program tests	The program runs normally and according to the spec.	The program does not comply with the specification in one or two aspects.	The program terminates abruptly in one test or does not comply with the spec in more than two aspects.	The program terminates abruptly in two tests or does not comply with the spec.
3 Memory management	The application runs with no anomaly when analyzed with **Valgrind**.	Only a leak or an incorrect memory deallocation is detected.	**Valgrind** detects two anomalies when executing the application.	More than two anomalies are detected when executing the application with **Valgrind**.
4 Debugging messages	The main steps of the of application print debugging messages when the symbol `DEBUG` is defined.		The application prints few debugging messages.	The program does not contain any debugging messages.
5 Use of file templates and documentation in the header.	All the files follow the template, have all the mandatory fields filled and the documentation in the header is helpful.		One or two of the files do not follow the template, do not have all the mandatory information fields filed, or the documentation in the header is not enough.	More than two files do not follow the template, have some fields empty, or the header documentation is not sufficient.
6 Global variables, structures and function documentation	All the global variables, structures and functions are perfectly documented.	In a file there is a variable, a function or a structure definition with no comments.	There are two files with variables, functions or structure declarations with no documentation.	There are more than two files lacking documentation for functions, structure declaration or global variables.
7 Code Style	All files comply with the requirements of the style guide.		Some of the files do not comply with the requirements of the style guide. Or some of the criteria in the style guide are ignored completely.	The majority of the files are not indented or do not observe the 80 column width limit. Or most of the style guidelines are ignored.
8 The application is intuitive	The user understands with no problems the functions, the data to introduce and how to introduce them.		In general the messages are clear, but in some cases it is not clear what the user has to do.	The user is frequently stalled because the functions are not clearly explained.