Final Project - MPCS 51040

(Issued: November 28, 2022; Last modified: December 9, 2022)

Deadline: Sun Dec 11 2022, 11:59am CST

General Instructions

These are general instructions which typically apply to every assignment, where apprioriate, and **apply unless explicitly stated otherwise**.

If you have questions about this assignment, please ask for clarification on Ed Discussion. **Any instructor clarifications on Ed Discussion are assumed to be known, and apply to this assignment**.

Compiling

Your code must compile with `gcc -std=c11 -Wall -Werror -pedantic`. There should be **no warnings or errors** when compiling with gcc (as installed **on `linux.cs.uchicago.edu`**).

Your code will be tested on `linux.cs.uchicago.edu` and must compile and run there!

Handing in

To hand in your final, you need to commit all requested files (***with correct filenames***!) to your personal git repository. **Make a subdirectory called 'final' and place your files under that directory** (or a subdirectory thereof, as specified in the problem assignment). Don't forget to commit and push your files! You can check on the GitHub website to make sure all files were committed and pushed to the remote repository correctly.

The deadline for this final is Sun Dec 11 2022, 11:59am CST. Any changes made after the deadline are not taken into account.

Code samples

This document, and any file you might need to complete the final can be found in the git repository `https://github.com/mpcs51040-2022/class`.

Grading

The following applies unless specified otherwise in the problem assignment. Your code will generally be graded based on the following points (in order of descending importance):

- Correctness of the C code: **there should be no compiler errors or warnings** when compiling **on linux.cs.uchicago.edu**. There should be no memory leaks or other problems (such as those detected by valgrind).
- Be prepared to handle errors, for example when opening files.
- Correctness of the solution. Your code should implement the required functionality, as specified in this document.
- Code documentation. Properly documented code will help understand and grade your work.

- Code quality: your code should be easy to read and follow accepted good practices (avoid code duplication, use functions to structure your program, . . .). This includes writing portable code (which will work on both 32 bit and 64 bit systems for example).
- Your code should not have hardcoded limits (such as the maximum input set size or the maximum number of elements, . . .), *unless it is explicitly specified in the assignment that this is allowed.*
- Global variables or static local variables **are not allowed** unless explicitly mentioned otherwise.
- Efficiency: your code should not use more resources (time or space) than needed.

Some items might be marked as 'optional' or 'extra credit'. When correctly completing these tasks, the points obtained will go towards mistakes made elsewhere in the **same** assignment.

Assignments

1: Vararray

Goal of this assignment

The C language doesn't have a built-in variable-length array type – unlike for example C++ (`std::vector`) or python (`list`).

While it is possible to allocate a fixed-size array as a local variable, the lifetime of that array ends when the function returns. The alternative, making a global variable breaks good software development practices. And in either case, the programmer would be responsible for keeping track of the number of elements in the array, ensuring no out of bound accesses occur and that no attempt is made to exceed the (fixed) array capacity.

Given the fundamental nature of a convenient and safe to use array data structure in any non-trivial program, this assignment aims to rectify this omission.

We will be building a generic (non-type specific), safe and resizeable array data structure, with the goal of reusing this code elsewhere in more complex assignments.

vararray

Why `void *` for elements

As discussed during the lectures, using `void *` allows the use of our `vararray` with any kind of type. If the particular type we want to store in the array is smaller than a pointer, we can cast this type to and from a `void *` pointer and store the data directly in the array.

If the type does not fit, we can instead store a pointer to the type in our array.

Why a `vararray_destroy_func_t`?

If our `vararray` ends up storing **pointers** to data external to the array, and **if** that external data is allocated on the heap (i.e. using `malloc`) setting an appropriate destroy function in our `vararray` can simplify using the array since the implementation of the array itself can take care of cleaning up the external data belong to an element when this element is removed from the `vararray`.

For example, if our `vararray` stores pointers to C-style strings **allocated on the heap**, we could provide a destroy function as follows:

```
static void my_destroy_func(void * element, void * user)
{
    free(element);
}
```

This way, we can call `vararray_destroy` or `vararray_clear` without having to worry about first freeing the memory associated with each `vararray` element.

Note that his only works properly provided the user of the `vararray` provides a correct destroy function as well as using the array in a proper manner.

For example, if a pointer to the same c-style string would be stored twice in the `vararray`, upon `vararray_clear` the destroy function would be called twice with the same value (which would result in calling `free` for the same pointer twice in our example)!

Why `void * visitdata` and `void * userdata`?

This is a common pattern in the C language whenever a (typically library provided) function needs to call back into that function user's code.

For example, consider `qsort_r`:

```
 void qsort_r(void *base, size_t nmemb, size_t size,
       int (*compar)(const void *, const void *, void *),
       void *arg);
```

The goal of this parameter to allow the caller of the `qsort_r` function to pass information to the `compar` function provided to `qsort_r`.

For example, imaging that the `compar` function can support sorting in an ascending or descending manner. Without the `void * arg`, the `compar` function would have to resort to using a global variable to determine if it should be sorting in ascending order or not.

The reason the type of this extra argument is `void *` is the same as explained in section ; If more data than can fit in a pointer sized memory location needs to be transferred, this data can be passed to the `compar` function by simply pointing to that data and passing the pointer to the function.

Unit Test

You need to provide a unit test (using the `cunit` library) for your `vararray` code. You need to have a **separate test function for each function in `vararray.h`.** (Even though most of the functions cannot be tested in isolation, you still need a separate test for each function.)

This is a black-box test: you will not be using any knowledge of the internal structure or implementation of your `vararray` code, and can only use the functions from `varray.h` to design your tests. To put it another way: it should be possible to take your test and run it against somebody else's implementation of `vararray` and any conforming implementation (i.e. following the specification in this documentation and `vararray.h`) should pass your test.

Write your test code in `vararray_unit.c` and add a `vararray_unit` target to your `Makefile` which builds the unit test, as well as a `test` target which builds the unit test and runs the test `under valgrind`.

The test is not an afterthought – there is a significant amount of points on your test code.

Notes

- Make sure to study (and fully understand!) `vararray_example.c` before starting your implementation.
- You are not allowed to change any of the functions provided in `vararray.h`.
- You are not allowed to put the definition of `struct vararray_t` in `vararray.h`.
- You are allowed to **add** additional `vararray` functions to `vararray.h` as you see fit to simplify your usage of `vararray` in other projects. (For example, adding a `vararray_find` or a function allowing at positions other than the end of the array might be useful.) You are not required (but it is still recommended) to write unit tests for functions you add.
- If you modify `vararray.h` in any way, your `vararray.h` and `vararray.c` must continue to work with any other valid unit test for an unmodified `vararray.h`.

Handing in

The following files need to be committed for this assignment **in directory `final/vararray`**:

- `vararray.h` and `vararray.c`.
- `README.vararray` which describes any addition you made to `vararray.h`. If you did not make any changes to `vararray.h` then state this in your readme file.
- A `Makefile`:
  - capable of building `vararray_example` and `vararray_unit`;
  - with a `test` target which **builds and runs** `vararray_unit` **under valgrind**;

Your code **must** compile without warnings and running valgrind should not be able to detect any issues when running `vararray_example` or `vararray_unit`.

Using `vararray` as part of a larger project

If you're using your `vararray` as part of a larger project, keep the directory structure as described here. In the larger project, you can use an appropriate relative include directive to get access to your `vararray` implementation.

For example, if you have the following directory structure:

```
final/vararray
final/snake
```

Any `.c` file within the `snake` directory can include `vararray.h` using:

```
#include "../vararray/vararray.h"
```

Note that your program will also have to link with `../vararray/vararray.c` (or at least the resulting object file) in any project that uses `vararray.h`.

2: FIFO for coordinates

Goal of this assignment

This is a small helper data structure for a game such as snake. We want to keep a FIFO of coordinates (`struct coordinate_t`, provided by `coordinate.h`) so we can simplify 'moving' our snake across the screen.

If you consider the motion of the snake in the game as far as which positions (coordinates) the snake occupies on the screen, you can see that the movement can be described using a FIFO (first in first out) data structure: when the head moves to the next position, the tail moves up one position but all the other positions making up the snake remain in the same place.

This perfectly matches a FIFO structure, where the next position of the snake's head is pushed onto the FIFO and the the old tail position (the last segment of the snake) can be popped of the FIFO. In other words, if we push the new head position of the snake on the FIFO, and pop once (removing the old tail position), the FIFO exactly contains all of the positions occupied by the snake.

While it would be possible to implement this FIFO structure using a generic approach (i.e. `void *` data elements), we are directly using `struct coordinate_t` here instead. If ever needed, it should be straightforward to change the code to make it more generic by changing the data stored in the FIFO to `void *` *(and possibly adding a destroy callback function for convenience).*

Implementation: CFIFO

Take a look at `cfifo.h` (see class repository) for the functions you need to implement.

You can see how these functions are used by studying the unit test (`cfifo_example.c`).

You are free to decide on how to implement these functions except for the following:

- Your implementation should follow all specification in this document and `cfifo.h`. This means (but is not limited to) that it should work with any black-box unit test without any memory leaks or valgrind warnings.
- Your implementation **needs to use a linked list (any kind of linked list) internally**.
- You cannot modify any of the existing functions in `cfifo.h`.

Most functions are straightforward and self-describing. However, pay special attention to the comments provided in the header. In particular, if a certain return value is prescribed, ensure your implementation honors that description.

Note the `cfifo_visit` function; You can see how it is used in the example code (see the `demo_visit` function). A good way of testing your understanding of the visit functionality would be to implement a `cfifo_contains` function which returns `true` if the FIFO contains a coordinate matching a provided `struct coordinate_t`. You should be able to implement this function by internally calling `cfifo_visit` with an appropriate visit function.

Notes

- Make sure to study (and fully understand!) `cfifo_example.c` before starting your implementation.
- You are not allowed to change any of the functions provided in `cfifo.h`.

- You are not allowed to put the definition of `struct cfifo_impl` in `cfifo.h`.
- You are allowed to **add** additional `cfifo` functions to `cfifo.h` as you see fit to simplify your usage of `cfifo` in other projects. (For example, adding a `cfifo_find`.) You do not need to write tests for the functions you add (but are encouraged to do so).
- If you modify `cfifo.h` in any way, your `cfifo.h` and `cfifo.c` must continue to work with the any valid black-box unit test. (Which means you cannot remove or modify the behavior of any of the existing `cfifo.h` functions.)

Handing in

The following files need to be committed for this assignment **in directory `final/cfifo`**:

- `cfifo.h` and `cfifo.c`.
- `cfifo_unit.c`.
- `README.cfifo` which describes any addition you made to `cfifo.h`. If you did not make any changes to `cfifo.h` then state this in your readme file.
- A `Makefile`:
  - capable of building `cfifo_unit`;
  - with a `test` target which **builds and runs** `cfifo_unit` **under valgrind**;

Your code **must** compile without warnings and running valgrind should not be able to detect any issues when running `cfifo_unit`.

Using `cfifo` as part of a larger project

If you're using your `cfifo` as part of a larger project, keep the directory structure as described here. In the larger project, you can use an appropriate relative include directive to get access to your `cfifo` implementation.

For example, if you have the following directory structure:

```
final/cfifo
final/snake
```

Any `.c` file within the `snake` directory can include `cfifo.h` using:

```
#include "../cfifo/cfifo.h"
```

Note that your program will also have to link with `../cfifo/cfifo.c` (or at least the resulting object file) in any project that uses `cfifo.h`.

3: Snake

For this final, you will be implementing a text-based game.

Overview

The game you will be implementing is called ‘snake’. There are many variants; more information is available here: `https://en.wikipedia.org/wiki/Snake_%28video_game%29`.

The game consists out of a 2D playing field, on which ‘food’ can be found. The player controls the snake, which moves at a constant speed in a straight line, unless directed by the player to change direction. The goal of the game is to grow the snake by guiding it to the food. Every time the snake feeds, it increases in length. If the snake runs into anything else (the borders of the playing field, or obstacles on the playing field *or itself*, the game ends. Depending on the variant, as time progresses or as the snake grows larger, the snake moves faster as well.

The object of the game is to avoid running into obstacles. The player does this by controlling the direction of the snake, by pressing one of the array keys.

An example implementation which can be played in the browser can be found here: `https://playsnake.org/`.

Variants and Extra Credit

You can be as creative as you want. Here are some examples:

- You can make a level system in which a level ends after the snake reaches a certain length, and where each subsequent level is harder. You can make a level harder by:
  - having more obstacles on the playing field
  - increasing the movement speed and/or maximum snake length
  - ...
- You can place obstacles in the playing field.
- You can add ‘wormholes’ (directly connecting two far away parts of the playing field) – maybe a wormhole would be the only way to get in and out a certain area? Maybe wormholes could be one-way only?
- You can make the food move around after a certain time.
- You can add extra bonus items which could have different outcomes when the snake eats them; Some examples: slow down the snake, make the snake shorter again, add extra food, gain extra points, . . .
- . . .

You are not required to implement these suggestions. If If you implement at least three of the items from the list above, you will receive extra credit which can recover points lost elsewhere in the final. *Make sure to mention clearly in your `README.TXT` that you implemented optional features, and describe which features you implemented.* You are not limited to the examples above. Three or more non-trivial features (all features described above are considered non-trivial – use your own judgement for other features) will result in extra credit.

Doing so is optional; any implementation similar to the version described in the overview section, provided all the requirements described in this document are met, can result in a perfect score.

You are required to describe your game, and all of the features (for example if the snake will change speed or not, how the game decides what the next length of the snake will be, . . .) in a `README.TXT` file.

Implementation

A skeleton for the game is already provided. It consists out of the following files:

- `main.c`: The main program. It initializes the screen, creates the game object, handles keyboard input and controls the game timing. **Do not modify these files.**
- `util.h` and `util.c`: Some utility functions (mostly related to time). You will likely need those. **Do not modify these files.**
- `screen.h` and `screen.c`: These files implement screen manipulation. **You must not modify these files.**
- `game.h`: contains the interface `main.c` uses to interact with your game. There are (or will be) two implementations of this interface:
  - `demo_game.c` and `demo_game.h`: a fully working (albeit boring) example game. It demonstrates the interaction between `main.c` and your game.
    Do not modify these files.
  - `snake_game.c` and `snake_game.h`: You need to implement your version of the snake game here.

  You can control which of these two is used by changing the `game_selection.h` file.
- `game_object.h`: the interface implemented by each visual entity in your game. The following implementations exist (or need to be completed by you):
  - `square_wall_object.h` and `square_wall_object.c`: a fully implemented example (used by the demo game).
  - `snake_object`, `food_object`, `obstacle_object` and `border_object` need to be completed by you.

In additions to these files, you will need your `cfifo` and `vararray` implementations from the previous tasks.

curses

The skeleton code is using the *curses* library, which provides control of the terminal. Because of this, the code provided is less portable (a C11 compliant compiler alone is not sufficient; a curses-compatible library is required as well). The code should compile on most OS X and linux/unix derivatives. As always, we will be testing on `linux.cs.uchicago.edu` and it is your responsibility to ensure everything works on that system.

For more information about the ncurses library (default implementation of curses on linux and some other platforms), see
`http://invisible-island.net/ncurses/man/ncurses.3x.html` and `http://invisible-island.net/ncurses/NCURSES-Programming-HOWTO.html`.

You do not need to worry about these details; A simplified set of screen manipulation functions (implemented on top of the curses library) is provided for you by the `screen.h` header.

Note that if you want, you are allowed to directly use `ncurses` functions to manipulate the screen, for example if you want to be able to control colors. You can access the underlying ncurses window by using the `screen_get_mainwin` function of `screen.h`.

Linking

Depending on the system, the curses library itself might have other dependencies. To known which libraries you need to link with, use the following command:

```
pkg-config --libs ncurses
```

You can integrate this in your `Makefile` as follows:

```
CURSESLIBS=$(shell pkg-config --libs ncurses)

# Use $(CURSESLIBS) on your link command line
# i.e.
#   gcc -o main someobject.o someotherobject.o $(CURSESLIBS)
```

Valgrind & ncurses

Memory Leaks

For various reasons, most curses implementations do not free all memory when releasing resources (such as ending the screen). Although these are memory leaks in the strictest sense of the words, the amount of leaked data is constant (and so won't grow unbounded). (Regardless, even these kinds of leaks are considered bad style, as they can mask other more serious problems.) For these use cases, valgrind supports 'suppression files', a file listing known memory leaks that can be ignored under normal circumstances.

The repository contains a suppression file for the ncurses library on `linux.cs.uchicago.edu`. This file might not be complete, and is not guaranteed to be sufficient for other platforms (it is specific for the version of the library installed on the machine). If valgrind still reports memory leaks, please check if the memory was allocated in the curses library. If so, you can likely ignore the leak. *However, if this is the case, include that valgrind output in your `README.TXT`, and explain why you feel that this memory leak is due to the curses library and not a problem of your own code.*

To use valgrind with suppression files, use `--suppressions=filename` (for more information, see the valgrind manual page).

Valgrind output

Running a curses program with valgrind, or more in general, writing to `stdout` or `stderr` is a bit problematic since the curses library will assume full control over the terminal window. Any other code outputting to the terminal (valgrind or otherwise) will break that assumption, which will result (in the best case scenario) in that output being overwritten by the next curses screen update.

One possible workaround for this is to not write to the terminal while curses has control over it. For example, you could write to a file instead (using `fopen` and `fprintf` instead of `printf`).

For valgrind, you can use the `--log-file=` (see the manpage for valgrind) to make valgrind do the same. That way you can check the contents of the file after your (curses) application has finished running.

Alternatively, since valgrind only writes to `stderr` (and curse does not use `stderr`) you can also *redirect* `stderr` for your program, which has the same effect (and is probably easier since you do not have to modify the code). If you print to the screen for logging/debugging, you still have to make sure to use `fprintf` in combination with `stderr`!

So:

```
valgrind <usual valgrind options> ./snake 2>mylog
```

will redirect `stderr` for valgrind (and any program executed by it) to the file `mylog`.

You can view the contents of the file after your program/valgrind completes using `less mylog`.

Debugging

The fact that our game uses the whole terminal means that we can't directly debug the program via `gdb`. Otherwise, the output from the game and the gdb output would be mixed, making debugging very hard.

Instead, we'll be using the remote debug functionality of gdb. Instead of having gdb start the program directly, another program (`gdbserver`) will start the program and then wait for commands. We then use `gdb` as usual, but tell it to connect to the running `gdbserver` to control the program that we want to debug.

Steps:

1. Run `gdbserver --multi localhost:12345`
   NOTE: if somebody else is already using port 12345, pick another port between 1025 and 65535.
2. Now open another window (possibly via SSHing to the **same** linux server). You can tell from the prompt where you started `gdbserver` which server you should connect to (for example `linux1.cs.uchicago.edu`).
3. In the second window, start gdb using the `gdb` command (or `cgdb` if you prefer).
4. Now tell gdb that we will be connecting to the remote, that the remote is on the same system, and which program we want to debug.
   - `set sysroot`
   - `file game` (assuming game is the executable you want to debug and that it is in the current directory)
   - `target extended-remote :12345`
   - `set remote exec-file game`

Now you can debug as usual. For example, setting a breakpoint on the main function (`break main`) followed by the run command (`r`) will allow you to step through the program. You'll see how the window that is running `gdbserver` will be showing the game output. Note that while the game is in stopped state (because it hit a breakpoint for example) it will not be able to respond to keypressed. However, any input during the stopped state will be processed once the game runs again.

Requirements

You have a lot of freedom in how to implement and structure your game. However, the following rules **must** be followed:

- Do not modify files which are so marked (either in this document or in the header or implementation file itself).

- Nothing should be drawn on the screen directly from your `snake_game.c` file. **Anything** drawn on the screen must be done via the `game_object_draw` call made on an appropriate object (such as `snake_object` or `border_object`).
- Your game implementation **must** use `vararray` (in `snake_game.c`) to hold **all** of the objects visible on the screen.
- Your `snake_object` implementation **must** use `cfifo` to keep track of the positions occupied by the snake.
- Your `snake_ojbect` should use the visit functionality of the `cfifo` to implement the `draw` and `overlaps` methods.
- Your game should adjust to the actual size of the terminal window (as reported by the `screen_getmax[xy]` functions). You do not have to worry about resizing the window once the game starts. You are allowed to require a minimum playing field of 60 wide and 60 tall. **If you do so, your code should check when constructing the game object (in `snake_game.c`) and show an error message (using `screen_message`) and end the game**.
- You **must** use the destroy functionality of the vararray (i.e. `vararray_create_with_destroy` to automatically release any remaining game objects when the `vararray` itself is destroyed).
- You should prefer to have one `.c` file per object; if extra object-specific functionality needs to be made available you will have to put a declaration for those functions in the header for the object (so it can be included and accessed by other code).

Steps

The following approach is recommended to get you started. It is not an exhaustive description of everything that is needed. If you get stuck or have doubts, make sure to ask for help immediately!

Directory Structure

1. You should have completed the `cfifo` and `vararray` task (and all unit tests should pass without any valgrind warnings or leaks).
2. Make sure you placed those files in the required directories in your personal repository.
3. Create `final/snake`in your personal repository and copy the game files (everything except for the `cfifo` and `vararray` files to that directory.

Makefile

Create a `Makefile` capable of building the (unmodified) code provided to you. (See the 'Handing In' section for more details on what is expected of the `Makefile`.

Study the example game

1. Run the example game.
2. Review `demo_game.c`. Make sure you understand how the functions in that file end up being called from `main.c`. (Hint: it involves `game_selection.h`)

3. Review the draw function within `demo_game.c`. Can you see how this ends up calling the draw function within `square_wall_object.c`? (Even though this function isn't even mentioned in `square_wall_object.h`?)

Start your own game

Based on `demo_game.c`, start the implementation of your own game in `snake_game.c` and `snake_game.h` and update `game_selection.h` so your game code is called instead.

Don't try to do too much at this point:

- Your 'constructor' function (the one returning `game_t *`) should be allocating memory for the private game data (just like `demo_game.c` does) and should be storing the provided `screen_t *` for later usage.
- Your draw function can simply be clearing the screen (and the refresh needed to make the changes visible).
- Implement the `key` and `tick` function so the game ends as soon as a key is pressed.
- Add a start message (using `screen_message`) which is only shown once (from the `tick` function, similar to `demo_game.c`).

Adding to your object collection

1. Now add a `vararray` to your private game data structure and make sure it is properly destroyed when your game is destroyed.
2. Set up your `vararray` in such a way that it destroys all of the objects it holds when the `vararray` itself is destroyed (i.e. you **must** use `vararray_create_with_destroy`).
3. Update your `draw` function so it calls `game_object_draw` for each object in the `vararray`.
4. Do not forget to check for memory leaks (and invalid memory accesses) using valgrind!
5. Now create a function in `snake_game.c` which will set up the playing field (call it `populate_level`). Use a variable in your private game structure to keep track if the playing field has been initialized (and – if you implement multiple levels – what level). Check this variable from your `tick` function and call `populate_level` if needed. (Make sure to update the variable so you don't end up repopulating over and over again!)
6. In your `populate_level`, add the `square_wall` object as shown in `demo_game`. However, in your case, you will be adding it directly to the `vararray`.
7. Now, when you run your game, you should see the square wall appear on the screen.
8. Congratulate yourself! Now it is time to start implementing your own game objects.

Screen Borders

1. Implement a game object representing the borders on the top, bottom and left and right side of the game area. Take a look at `square_wall_object.c` and `square_wall_object.h`
2. **Do this in file `border_object.h` and `border_object.c`.**
3. Add the screen border object to your game object `vararray`.
4. Run your game, and ensure that you now have borders around the playing field.

5. Remember to run your game under valgrind to ensure you don't have any memory leaks.

Collision Detection

A number of tasks require the ability to either find a free spot on the playing field or, given a specific spot, to determine if an object – and if so which – occupies that position.

You will need to implement two functions in `snake_game.c` for this task.

1. A function which, given a coordinate, checks all objects in the `vararray` (using the `game_object_overlaps` function) to see if any object claims that position.
2. A function which randomly generates coordinates on the playing field until one is found to be free (using the previously described function).

The former will be useful to check if your snake is going to collide with anything, the latter will be useful to randomly place a food object on the playing field.

Handing In

You need to hand in the following files (in directory **`final/snake`**):

- `Makefile`: Provide a `Makefile` which has the following targets:
  - `clean`: Remove all files created by the `Makefile`.
  - default target: build `snake`.
  - `snake`: build (**not run**) the game.

  Make sure to use the correct compilation flags! **To receive full points, your makefile should be efficient and not recompile/relink any file unless needed**. (This means you should compile `.c` files individually to object files (using correct header dependencies) and then link the object files together into an executable.)
- **All files** needed to build your game, even files you did not modify.
- Note that the `cfifo` and `vararray` files are in their own directory (as specified in their respective tasks). **You should not need to copy those files into `final/snake`**.
- `README.TXT`. **Make sure to clearly indicate in your `README.TXT` what exactly you implemented, what works (and what doesn't work) and anything else you think could be relevant for us to evaluate your work.**