

# **Project report**

<b>Introduction</b>	<b>2</b>
<b>Cards: how are they implemented?</b>	<b>2</b>
<b>The code structure</b>	<b>2</b>
<b>About object-oriented programming</b>	<b>4</b>
<b>Using GitHub</b>	<b>4</b>
<b>The display</b>	<b>5</b>
<b>Thoughts on the display code</b>	<b>6</b>
<b>Unit tests</b>	<b>6</b>
<b>Profiling data</b>	<b>7</b>
<b>Als</b>	<b>9</b>
<b>Leaderboard</b>	<b>9</b>
<b>The sorting problem</b>	<b>10</b>
<b>About global variables</b>	<b>10</b>
<b>Project documentation with Doxygen</b>	<b>11</b>
<b>What's next?</b>	<b>13</b>
<b>Conclusion</b>	<b>13</b>

## Introduction

After learning a programming language, it is essential to put everything learned into practice. For this reason, after our C class, we were given a team project: make the “Coinche” french game into a terminal-based application, where a single user can play with and against AIs.

Even if we did not find the project topic to be the most exciting one, the goal was clear: give our best until the deadline, and learn a maximum of things along the way (this is why we looked into unit tests, for example). This document will tell you about issues we had, what means we used for this project, and our thoughts on various subjects.

## Cards: how are they implemented?

At the very beginning of the project, we did a brainstorming session to decide on the base structure of the program. The first obvious thing to consider were the cards. At first, we thought about storing cards as an array of two elements: a value and a color (“color” means “suit” here, but that early on in the project we didn’t pay attention to this vocabulary issue, and now we are too used to it to change).

We quickly discarded the idea of storing a card as an array, to avoid having to deal with two-dimensional arrays, and also because we thought of a much better solution: type definition. Structs and enums, two powerful ways of storing data, were soon at the core of the project. Thus, a card is a structure composed of a color and a value, both of which are represented as enumerations (later on, we added a third element to the cards, a boolean saying whether or not the card can be played). The result of this is that card assertions can often be read as if they were english sentences, which is great.

## The code structure

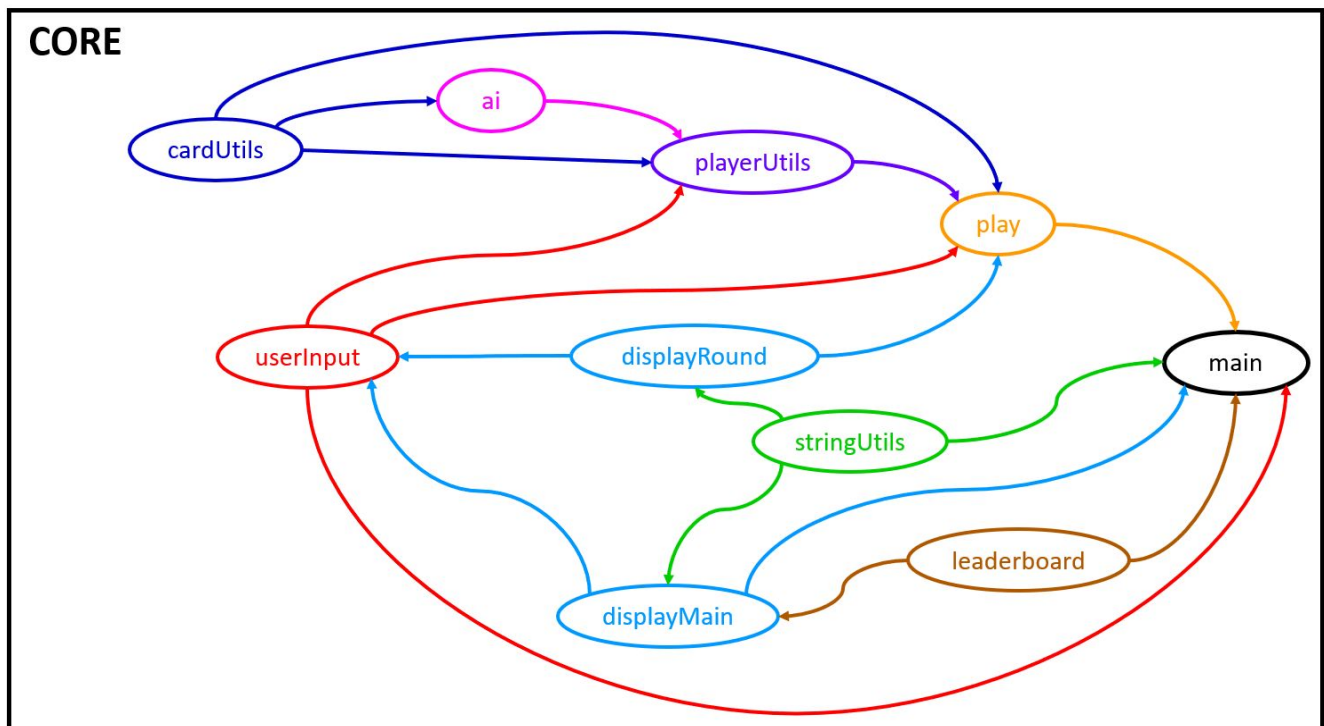
To make for a more structured code, the various functions are distributed over different files. Here is a quick list of them (each file has both a .c and a .h version):

- cardUtils, playerUtils and stringUtils: these files contain functions relative to their respective objects. If this project had been coded in an object-oriented language, most of these functions would have been suited as class or instance methods. More on object-oriented programming later.
- ai: each decision-making AI is in there (for cards and contracts). There will be a dedicated section to AIs later.
- play: this file has all functions to handle a coinche game. It uses a lot of functions from cardUtils.

- **leaderboard**: the functions in this file are opening the leaderboard.txt file and writing/reading data to it, to store the player wins.
- **displayMain** and **displayRound**: only user-related display functions, with boring code. More thoughts on the display code later.
- **userInput**: this file contains user input related functions, from simple string inputs to asking the user to input a card or make a contract.
- **main**: a few functions related to setting up the program and some dedicated to the main menu. There is also the `main()` function in here, which contains five lines of code.
- **core**: this essential file has all type definitions (structs and enums)

Here is a graph showing the full files structure:

An arrow means “this .h file is included in this .c file”. Note that the core header file is



included in every other header file. This is the only header included in other headers. Also, every header file is included in the corresponding C file (so the core.h file is included as well).

### About object-oriented programming

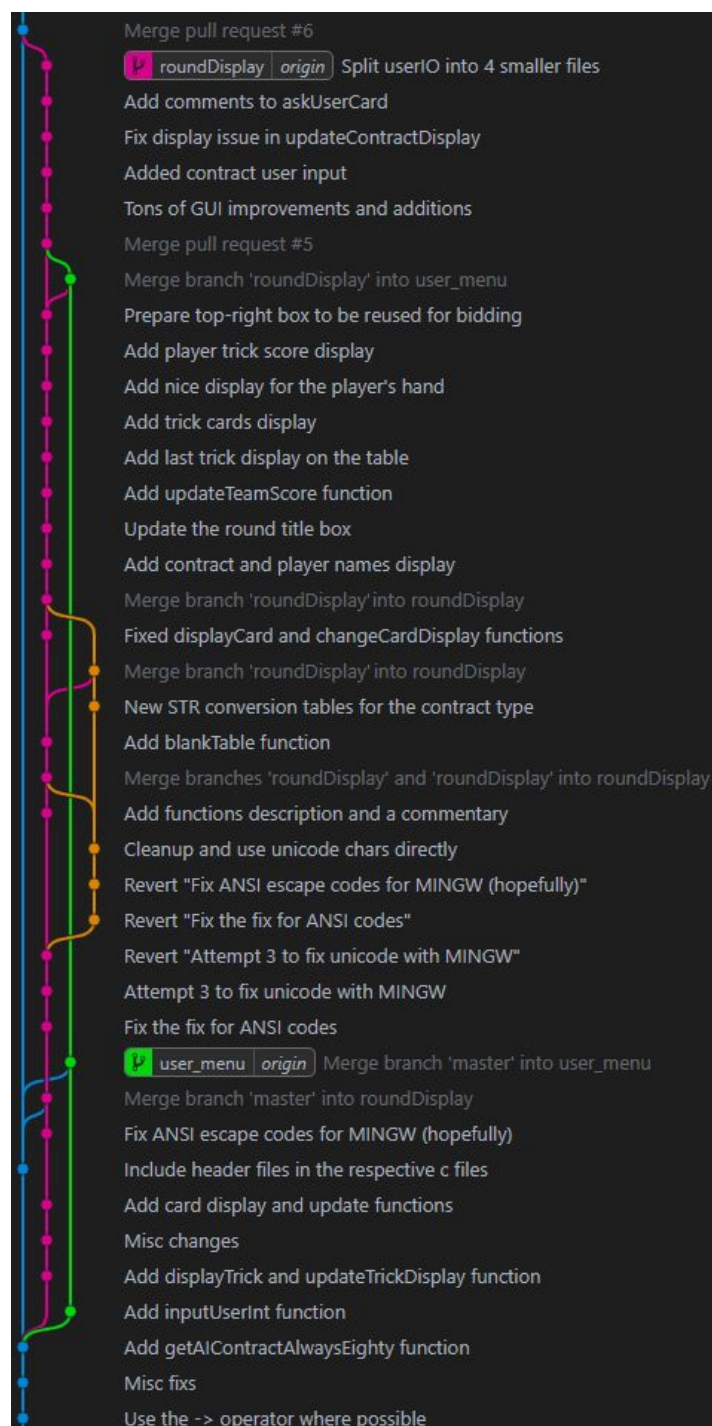
This project would have been really well suited to [OOP](#). Indeed, cards and players would have been classes, and would each have methods to work with them. With enough time to think about it, a lot more things could have been classes (a game class, a terminal/display frame class...).

## Using GitHub

Being a group project, we absolutely needed a way to both write code at the same time, and not having to go through the tedious task of merging code ourselves. The obvious solution to this problem is [Git](#), the version control system, and we decided to host the project on [GitHub](#), for its intuitive interface.

At first, we were both using the software GitHub desktop, but since we were using [Visual Studio Code](#) to code the game, we eventually switched to using the embedded Git support inside the editor itself, which proved to be both convenient and efficient. For a future project, this is definitely something that we'll try to use again.

The image below is an extract of the commit network from the project (this is the display code implementation):



## The display

From the very beginning, we knew that dealing with an user would be one of the most tedious part of this project. For this reason, we delayed the display and input part of the program as much as we could. At some point, the program was capable of running full AI-only games, without a single user prompt or display message. Sadly, we were required to add a user support, so we had to work on it eventually.

As expected, it turned out to be very tedious right from the start. We had a tough decision to make, because of two features we really wanted to use:

- Unicode characters. Displaying a card color as a letter is really not user-friendly, and would make reading cards quite hard, and Unicode would solve this issue.
- Ansi escape codes. When starting to code game display features, we realised that if we had to redisplay the full table every time the user entered a wrong input, the function structure would be a nightmare (because we didn't want to transmit any information through global variables). So, what was the solution ? Keep the same table displayed, and to change a specific information, move the cursor to it and write over the previous text. Ansi escape codes and sequences can move the cursor, and that's why we needed them.

The issue with these two features is that they were really hard to setup for Windows. Until that point, the project would compile under both MinGW and Cygwin, which was really neat. But neither Ansi escape codes nor Unicode worked with an executable compiled with MinGW. We lost too much time trying to make it work, and even though we eventually got Ansi escape codes to work (by using the windows.h library and changing an environment variable at runtime), we were stuck with Unicode. On the other hand, with the executable compiled with Cygwin, "it just worked".

This is the tough decision mentioned above: we had to choose between cross-platform and two key features (Unicode and Ansi escape codes). After weighting the pros and the cons of both paths, we decided to drop support for MinGW. This means that to run the executable compiled with Cygwin, we have to distribute a "cygwin1.dll" file along with it.

Legally, the cygwin1.dll file is covered by a GNU General Public License. Here is a quote from their [website](#): "To cover the GNU GPL requirements, the basic rule is if you give out any binaries, you must also make the source available." This is perfectly fine for us, as our source code is available publicly on GitHub.

Another notable thing we used Unicode for is the box-drawing character set:

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
2500	—	—			—	—			—	—						
2510	┐	┐	┐	┐	┐	┐	┐	┐	┐	┐	┐	┐	┐	┐	┐	┐
2520	└	└	└	└	└	└	└	└	└	└	└	└	└	└	└	└
2530	┌	┌	┌	┌	┌	┌	┌	┌	┌	┌	┌	┌	┌	┌	┌	┌
2540	┐	┐	┐	┐	┐	┐	┐	┐	┐	┐	┐	┐	┐	┐	┐	┐
2550	=		┌	┐	┌	┐	┌	┐	┌	┐	┌	┐	┌	┐	┌	┐
2560	┌	┐	┌	┐	┌	┐	┌	┐	┌	┐	┌	┐	┌	┐	┌	┐
2570	┌	/	\	X	-		-		-		-		-		-	

These were of tremendous help to design various UI parts, for example cards or the main frame.

Let's talk about what is probably the main issue of this project: the user can break the display at will. Indeed, the user can resize the window, or move the cursor during inputs, which will then break the display and in most cases will make the game unplayable. We didn't find a reasonable way to fix this. We did find a solution to prevent the user from scrolling, as we can enable the [alternate screen buffer](#) with Ansi escape codes. Besides, only the display will break: we didn't find a way for the user to crash the program (yet), as all user input is being carefully parsed before use.

### Thoughts on the display code

For various reasons, the display code is the worst part of this project. Indeed, the ANSI escape codes are making for unappealing code, even though we did write a lot of comments explaining each of them. We heard too late of [ncurses](#), a library that would have done all the heavy lifting for us, and would have made the application terminal-independant. Maybe the display code would even have worked with MinGW.

Moreover, most of the display functions are looking redundant, with just enough difference that grouping them would be impractical. The string utils functions are also contrasting with the overall quality of the game code. Even though the interface may look nice to the user, we are really unhappy with the code below it.

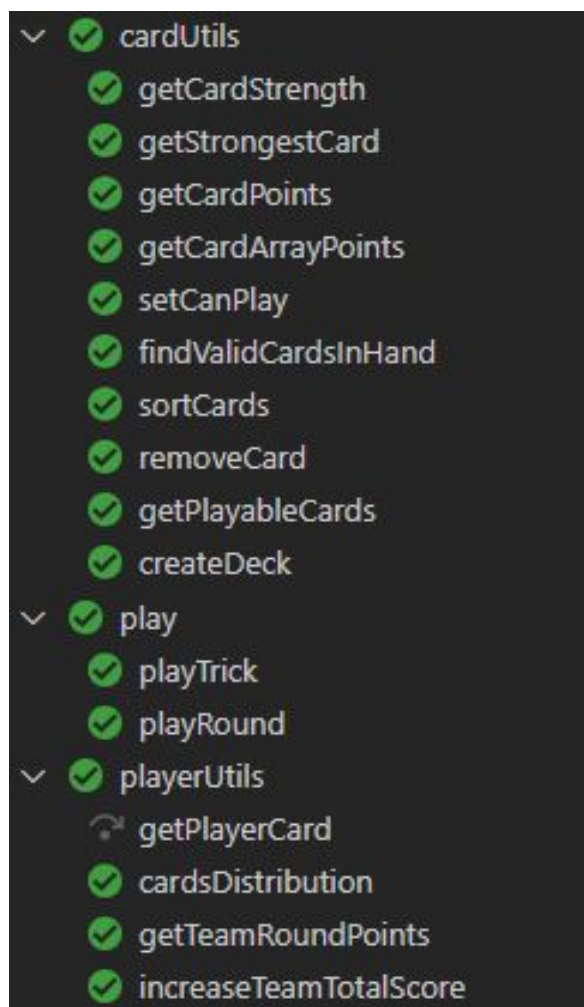
### Unit tests

As we built this project starting with the small core functions first (such as `getCardStrength`), the functions became increasingly complex with time. The coinche rules being a little overly complicated at times, we started having some trouble with the bigger functions: is this function fully respecting the coinche rules? How can we make sure it does?

The answer to these questions was found in unit tests. The idea is to have a piece of code dedicated to testing a single function, feeding it every possible input we can think of, and comparing the function's output to an expected result. If the expected result differs from the actual one, the test fails.

In the C language, setting up unit tests is far from easy. As it was pretty early on in the project, and we had plenty of time before the deadline, we decided to give it a try, mostly as a proof of concept. It was very interesting and we learned a lot along the way.

We are using [Ceedling](#), a Ruby build system for C projects. Its use in our case is to automate Unity, the unit test framework providing assertions (Is this int array equal to this expected array? Is this return value *true*?). We decided to use Ceedling because there is a Visual Studio Code extension providing a user interface for it. Here is the result of all this hard work:



These are the results of the unit tests we ran right before the 1.0 release. We did not write a lot of tests, as it was very time-consuming to write a single one while being thorough and thinking of every edge case. We also did not focus on writing nice commented code for these, but keep in mind this is just a proof of concept.



## Profiling data

When we finalized the function to play AI games, we started to wonder how many times the different functions were called, and how long was the execution of a single game. To find an answer, we started looking into profiling tools, and ended up using [gprof](#).

This is profiling data on the v1.0 of the program, simply running 10 000 AI games with the default AIs:

Flat profile:

Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
49.22	0.63	0.63				_mcount_private
14.06	0.81	0.18				__fentry__
6.25	0.89	0.08	3094080	0.00	0.00	removeCard
4.69	0.95	0.06	3373574	0.00	0.00	setCanPlay
4.69	1.01	0.06	49412	0.00	0.00	cardsDistribution
3.12	1.05	0.04	1512896	0.00	0.00	getAICardStandard
3.12	1.09	0.04	1512896	0.00	0.00	getPlayableCards
3.12	1.13	0.04	242253	0.00	0.00	getAIContractStandard
2.34	1.16	0.03	25652795	0.00	0.00	getCardStrength
1.56	1.18	0.02	3077067	0.00	0.00	getStrongestCard
1.56	1.20	0.02	1512896	0.00	0.00	findValidCardsInHand
1.56	1.22	0.02	1512896	0.00	0.00	sortCards
1.56	1.24	0.02	378224	0.00	0.00	getCardArrayPoints
1.56	1.26	0.02	378224	0.00	0.00	playTrick
1.56	1.28	0.02	10000	0.00	0.05	playGame
0.00	1.28	0.00	1512896	0.00	0.00	getCardPoints
0.00	1.28	0.00	1512896	0.00	0.00	getPlayerCard
0.00	1.28	0.00	242253	0.00	0.00	getPlayerContract
0.00	1.28	0.00	81521	0.00	0.00	getTeamRoundPoints
0.00	1.28	0.00	81521	0.00	0.00	increaseTeamTotalScore
0.00	1.28	0.00	49412	0.00	0.00	bidAttempt
0.00	1.28	0.00	49412	0.00	0.00	createDeck
0.00	1.28	0.00	47278	0.00	0.00	awardTeamPoints
0.00	1.28	0.00	47278	0.00	0.00	bidUntilContract
0.00	1.28	0.00	47278	0.00	0.01	playRound
0.00	1.28	0.00	14	0.00	0.00	clearInfoMsg
0.00	1.28	0.00	6	0.00	0.00	cropStr
0.00	1.28	0.00	6	0.00	0.00	formatStr
0.00	1.28	0.00	5	0.00	0.00	displayInfoMsg
0.00	1.28	0.00	5	0.00	0.00	inputUserStr
0.00	1.28	0.00	3	0.00	0.00	displayFrame
0.00	1.28	0.00	3	0.00	0.00	inputUserInt
0.00	1.28	0.00	2	0.00	0.00	displayMenu
0.00	1.28	0.00	1	0.00	0.00	inputUserAcknowledgement
0.00	1.28	0.00	1	0.00	470.00	mainMenu
0.00	1.28	0.00	1	0.00	470.00	playAIGames
0.00	1.28	0.00	1	0.00	0.00	resizeCmdWindow
0.00	1.28	0.00	1	0.00	0.00	setUp
0.00	1.28	0.00	1	0.00	0.00	tearDown



There's a lot of interesting data to discuss here. Firstly, functions are stored by execution time. The two heaviest functions, `_mcount_private` and `__fentry__` are actually the functions used to produce this data, so they aren't present in a non-debug compilation. Thus, the three functions taking the most time are `removeCard`, `setCanPlay` and `cardsDistribution`: this is surprising, because these are really small functions. All they have in common is that they use a loop. The most surprising is `cardsDistribution`, because of its relatively small number of calls; maybe the `rand()` function in it is taking a long time to execute.

We can also see that the most called function is `getCardStrength`, the very first function we wrote for this project, with an average of 2565 calls per game. Without the profiling functions, we can see that executing 10 000 AI games takes less than a second, which is a reasonable performance.

## Als

Each AI is composed of two smaller Als: a card AI, and a contract AI. The card one makes the AI decide which card to play during a trick, while the contract one decides which contract the AI should make during the bidding phase. From very early on, we knew that we wanted to do statistics on the Als to see how much stronger a better AI is in regards to the project topic ones. We currently have two card Als and two contract Als:

### Card Als:

- First available: this AI is very simple: it looks through its hand and plays the first available card. It's almost a random AI, and was used as a placeholder before coding the next AI.
- Standard: this is the AI presented in the project topic. If it can win, then it does so with the lowest possible card; otherwise, it plays its lowest card.

### Contract Als:

- Always eighty: this AI makes an 80 points contract in the color of its first hand card if it can, otherwise it passes. It was used as a placeholder before coding the next AI.
- Standard: this AI makes an 80 points contract if it has 3 strong cards in a given color, or a 120 points one if it has 4 or more strong cards. A strong card is defined as being a queen or better: on 10 000 games, this AI consistently wins around 1% less games (against a team of "Always eighty") than if the queen is not considered a strong card, but this is probably because taking less contracts means that the "Always eighty" team takes more unfounded contracts (thus failing more often). For this reason, we decided to stick with the queen being a strong card.  
On games with standard Als only, all 4 Als will pass on 4,5% of the rounds, on average (meaning that the cards must be dealt again).

The table below shows, for an average of 100 000 games, the win rate of the North-South team, depending on the selected AIs (both team members have the same AIs):

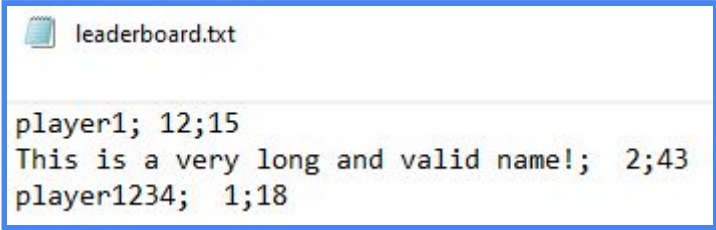
Average of 100 000 games Team N-S	Team E-W	Contract: Always eighty		Contract: Standard	
		Card: First available	Card: Standard	Card: First available	Card: Standard
Contract: Always eighty	Card: First available	53,11%	35,97%	38,67%	23,88%
	Card: Standard	70,98%	54,21%	57,87%	37,35%
Contract: Standard	Card: First available	64,53%	46,03%	52,06%	34,60%
	Card: Standard	79,03%	66,06%	69,33%	52,09%

We can see that having a good card AI seems to help more than having a good contract AI. Also, you might notice that the center win rates, when both teams have the same AIs, are not at 50%, but slightly above. We have no idea why that happens, maybe this reveals a bug in the code somewhere. The consequence of this is that the N-S team seems to have a consistent 2% win rate advantage.

Sadly, we would have liked to develop more AIs, but we didn't by lack of time. We could have made a much bigger table with more statistics; plus having a bigger challenge for the user.

## Leaderboard

The leaderboard file was one of the last things we added to the project. When deciding how store data in it, we had a couple decisions to make. Indeed, we wanted the user to be able to enter a relatively long name, eventually with spaces in it. We decided on a name with a maximum length of 50 characters, because that's the approximate width of our display frame. We decided that we didn't want fixed-length lines in the file, because most users wouldn't have a name 50 characters long. That's why we made variable-length lines in the file, to make the file a couple bytes lighter:



```
leaderboard.txt

player1; 12;15
This is a very long and valid name!; 2;43
player1234; 1;18
```

The picture above shows a leaderboard file generated by our code. Each line needs to contain two pieces of data: the player name, and its number of wins. You might notice that there are actually three pieces of data; more on that later. We also decided to have the file being always sorted, with the best player on top, to make displaying the leaderboard very efficient. Thus, we needed to sort the file whenever a player got an additional win.

When reading and writing to the file, moving the cursor forward is pretty easy, as the `fgets()` function always reads a full line. But we also needed to move the cursor backwards, and that's where we encountered a problem: how much characters should the cursor go back to move to the start of the line?

This would have been easily solved with fixed-length lines... The solution we decided to use is the following: store the line length at the end of the said line. This is the third piece of data you can see on the previous picture. The structure of a line is the following: <name>;<wins>;<line\_length>\n.

## The sorting problem

Eventually, we realized that we needed a card sorting function. Its main use would be for the standard AI function. We looked into the possible ways of sorting cards, and one solution did stick out: the `qsort()` function. This function is included in `stdlib.c`, and is an implementation of the [quicksort](#) algorithm, which is really efficient. This function takes as input an array of elements, the size of an individual element, and most importantly: a pointer to a comparison function.

We were really eager to try it out, since the `getCardStrength` function was exactly that: a comparison function. We just had to tweak it a little. That's when we realized the big issue: to compare two cards, the `getCardStrength` function needed at least one additional information, being the current trump. Thus, we were unable to use the `qsort` function...

In the end, we ended up making our own sorting function based on the [bubble sort](#) algorithm. Despite making a pretty optimized version of this algorithm, it is still one of the most performance-heavy part of the program. In a game with 4 standard AIs, out of the ~2500 calls to `getCardStrength`, around 1000 of them come from the `sortCards` function.

## About global variables

Having a structured program was really important to us. This means that every function gets only the information it needs, and that conveying information is strictly restricted to function calls. For this reason, we avoided global variables at all cost. However, we are using a few global constants. These are declared as global variables, except that they have the "const" keyword as well. Here is an example:

```
const int CARD_POINTS_TABLE[4][8];
```

Even though these are considered as global variable, they do not break the key principle stated above: conveying information is strictly restricted to function calls. This is because they are read-only. Wherever these global constants are used, there could often be a switch or an if/else-if chain, but using them is much more efficient and makes for a more concise code.

## Project documentation with Doxygen

While moving forward on the project, we were constantly adding some documentation on our functions. A few days before the deadline for submitting this report, we learned that a software called [Doxygen](#) can generate a full documentation using the comments in our code.

Doxygen must recognize specific tags in these comments in order to create the documentation. We firstly ran Doxygen without modifying the annotations in our code, just to see what the software can already recognize: the header files, the structures and the constants were detected.

We had to find out how the software could detect all the other parts of our code. First of all, the comments giving the informations to Doxygen had to be preceded by `/**` (or `/*!`) and ended by `*/`; which is pretty close to syntax of the C language comments so this modification didn't take too much time to be done. Then, inside those comments, the specific Doxygen tags must be used to announce what kind of code will follow. In our case, we used:

- `\file` (at the beginning of a C file);
- `\enum` (to announce an enumeration);
- `\def` (to announce a global constant);
- `\fn` (to announce a function);
- `@param` (to announce a parameter of the function);
- `@return` (to announce the return of a function).

We already knew and used the last two tags before discovering Doxygen thanks to our IT classes, so we used this syntax right from the start, every time we wrote function comments. Each tag must be followed by the declaration of the element in our code. A description of an element can be written after two line breaks (except for a parameter and the return of a function: the description can be added right after the tag and the declaration).

Here is an example of documentation with the `getStrongestCard` function (the code in Visual Studio Code on top, and the Doxygen-generated pdf below):

```
/**
 * \fn int getStrongestCard(Card cardArray[], int nbOfCards, Color trump, Color roundColor)
 * @param cardArray[]: array containing the cards to compare
 * @param nbOfCards: how many cards are being compared. Can be a single card
 * @param trump: the current trump
 * @param roundColor: the color of the first played card in the round
 * @return strongestCardPos -> the position of the strongest card in the set, where 0 is the first card of the Array
 *
 * Finds the strongest card in a set
 */
int getStrongestCard(Card cardArray[], int nbOfCards, Color trump, Color roundColor);
```

#### 4.3.1.7 getStrongestCard()

```
int getStrongestCard (
    Card cardArray[],
    int nbOfCards,
    Color trump,
    Color roundColor )
```

##### Parameters

<code>cardArray[]</code>	array containing the cards to compare
<code>nbOfCards</code>	how many cards are being compared. Can be a single card
<code>trump</code>	the current trump
<code>roundColor</code>	the color of the first played card in the round

##### Returns

`strongestCardPos` -> the position of the strongest card in the set, where 0 is the first card of the Array

Finds the strongest card in a set

We could have used other tags like `\author`; `\version` or `\date` but we considered that they weren't the most useful and that they'd have taken too much time to be implemented.

It took us a fair bit of time to review each function documentation and add documentation for the C files, the enumeration and the constant variables. We ran Doxygen after all these modifications and everything was recognized except the enumerations. Fortunately, some adjustments in the settings of the software made the enumerations recognizable.

Doxygen also offers a feature that “visualizes the relations between the various elements by means of include dependency graphs, inheritance diagrams, and collaboration diagrams, which are all generated automatically”, according to its website. We found this feature interesting and wanted to use it. Unfortunately, after having installed the [Graphviz](#) software (required to generate the graphs and diagrams) and configured Doxygen, we were unable to get the graphs working. The lack of time also conducted us to give up this feature of Doxygen.

Nevertheless we're already very happy with the pdf, generated by Doxygen, regrouping all of the elements of our project and their description in a very organized way.

### What's next?

Here is a list of things that, given enough time, we would have liked to add or improve. These are approximately sorted by interest and priority:

- Add more AIs
- Write more unit tests
- Do more (and better) error handling
- Make the game into a local multiplayer game
- Improve the display code by either switching to ncurses or doing a fully developed graphical interface

### Conclusion

This project was a very interesting experience, and we definitely fulfilled the goal of learning a lot of things along the way. Despite that, this is what the code feels like: a nice, solid code base for the game engine, plagued by dirty display code. This project would have been better either without user interface, or with a fully developed graphical one (which would, in retrospective, maybe not have been much more tedious than the current display functions). On a different note, maybe coding this game in an object-oriented programming language would have been better, so we are looking forward to our first OOP class.