Wah Yan Ryan Wan sc16wyrw

COMP1921: SDL Game

i) Structure of the code

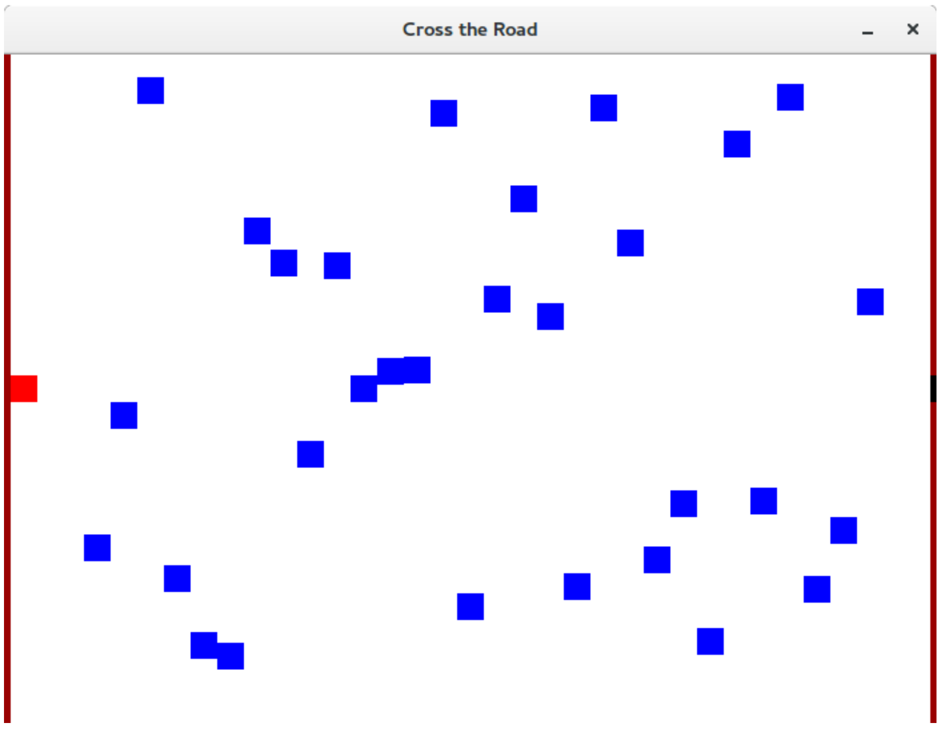
Modules:
main.c – This contains the input for controlling the red square, actions from collisions and also deals with completing and exiting the game.
game.c – Mostly deals with rendering items and destroying them.
obstacles.c – This dealtswith how the obstacle squares.

Functions:
init* SDL_Setup(char *name, int popupX, int popupY, int windowX, int windowY)
- Sets up a new window and checks for potential errors.
void renderColour(SDL_Renderer *ren, SDL_Rect *map, SDL_Rect *user, SDL_Rect *g, SDL_Rect *block, int blockNum)
- Renders objects in various colours.
SDL_Texture* renderText(SDL_Renderer *ren, str message, str f_type, int f_size, SDL_Color colour)
- Sets up True Font Type message and checks for potential errors.
void renderMess(SDL_Renderer *ren, SDL_Texture *tex, int x, int y, SDL_Rect *bit)
- Renders True Font Type message.
void cleanup(char *type, …)
- Destroys renderer and window.
bool has(SDL_Rect *subj, SDL_Rect *obj)
- Checks if there are two variables present.
bool box2box(SDL_Rect *box1, SDL_Rect *box2)
- Deals with interaction of two boxes (including border).
bool crash(SDL_Rect *player, SDL_Rect *block, int blockNum)
- Deals with player to obstacle collisions.
SDL_Rect* addBlocks(int blockNum)
- Add obstacle blocks into the game.
void roadCrossing(SDL_Rect *block, int blockNum, int speed)
- Deals with the vertical movement of the obstacle blocks.

Header Files:
obstacles.h – This contains functions for used in main.c and obstacles.c
game.h – This contains functions used in main.c and game.c

ii.) My Project

For my project, I chose to create a game using the SDL libraries. My game involves the player (controlling the red square) crossing from the left side of the window to the other, with the aim being to reach the black gate on the righthand side. There are obstacles in the way of the player, some which are stationary and some that move vertically. If the player hits any of these obstacles or the borders of the window, then they will be transported back to the starting position as shown below.



The goal was to make a game like Frogger, however I realised that this would be a complex task, so starting off simple and then adding more layers was the plan. I decided to work with as few functions as possible in order to create the fundamentals of the game before adding other extra features. The control of the user square, movement of the obstacles, collision with the obstacles and reaching the end point were identified as the core concepts. Creating borders on the left and right, colliding with the borders, along with the victory message were concepts that were thought of after planning and during the implementation of the core concepts.

The first iteration of the game simply involved moving the player's square around an empty window. This was to make sure the movement with the arrow keys would work correctly. The next stage was to add in stationary obstacles before creating rules for collision between the two types of squares. There would simply be a print message in the terminal saying "Crash" in order prove the function was working properly. After this worked successfully I added the reset to starting position feature if a collision occurred. Making the obstacles move in vertical fashion was the next task. The final core concept was to provide an end point for the user, which would stop the game.

After all the core concepts had been successfully implemented I decided upon some extra features. The first was to provide a clear starting and ending border, next I decided that collision with the borders would also be an interesting feature to have. Generating a message using SDL_ttf upon the game's completion was the final addition to game.

I generated most of the objects in the game using SDL_Rect, since both the player and obstacle squares were the same size. The only difference was the colours, which I changed using a combination of SDL_RenderFillRect and SDL_SetRenderDrawColor. The left and right hand side borders and end gate was also created using these two functions, along with the end point.

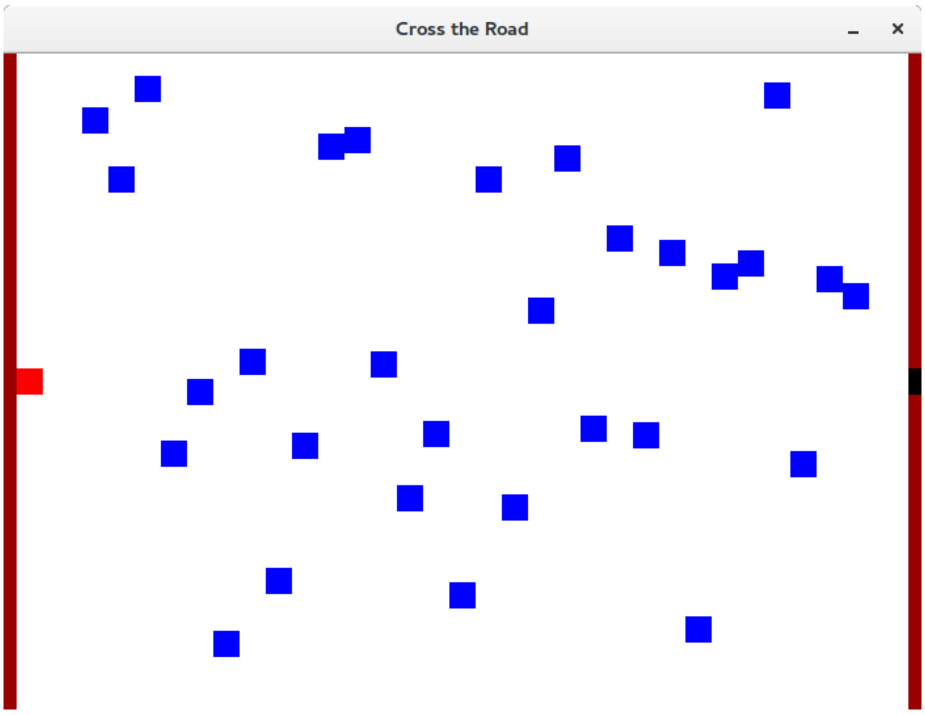
iii) Test cases and results

1) User collision with obstacles

The main difficulty of the game revolved around avoid the blue squared obstacles in order to reach the end point on the map. If the user collided into the obstacles then a notification in the terminal saying "Crashed into an obstacle" would be print along with resetting the player to the original starting position in the game.

```
else if (has(&map, &user) || crash(&user, traffic, blockNum))
{
  printf("Crashed into an obstacle\n");

  user.x = map.x;
  user.y = window_y / 2 - user.h / 2;
}
```

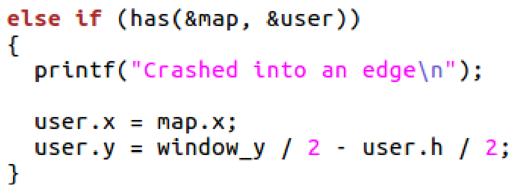


```
Number of cars created = 30
Crashed into an obstacle
Victory!
ESC: Game closed
```

Here there we can see that there was one collision. "Victory!" is displayed because the player completed the game, so the map will be cleared and the game stopped.

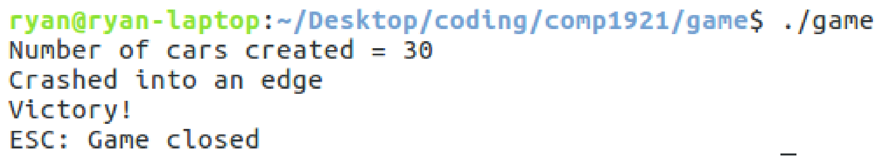
2) Setting the borders of the window

The game needs to be contained within the confines of the window that was created. If the player tries to move beyond the confines of the map, then "Crash into an edge" will be displayed in the terminal and the player will be reset to the starting position of the game. This applies to all four edges of the map, the left and right hand ones being limited by the red border strips.



```
else if (has(&map, &user))
{
  printf("Crashed into an edge\n");

  user.x = map.x;
  user.y = window_y / 2 - user.h / 2;
}
```

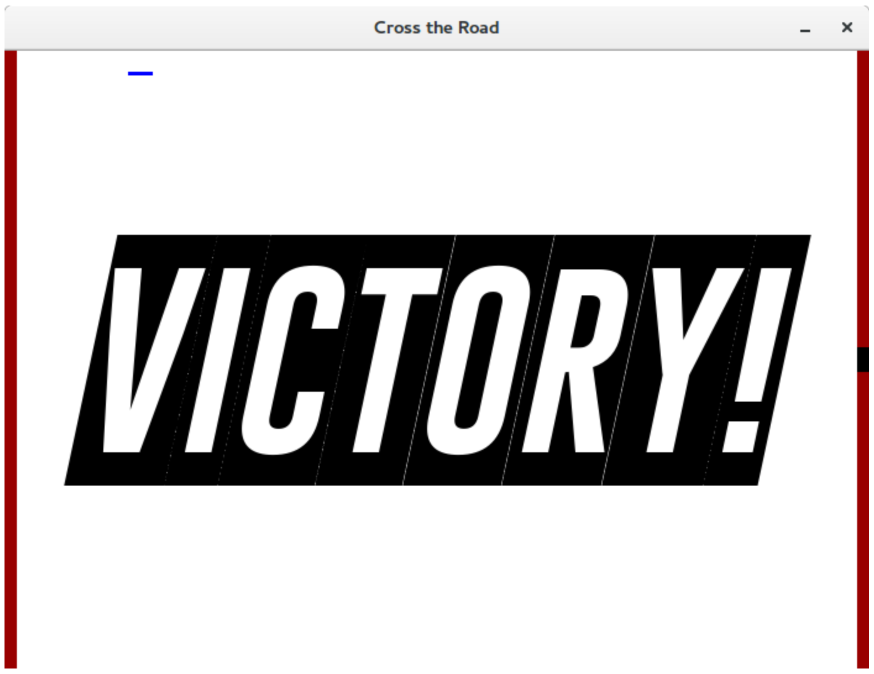


```
ryan@ryan-laptop:~/Desktop/coding/comp1921/game$ ./game
Number of cars created = 30
Crashed into an edge
Victory!
ESC: Game closed
```

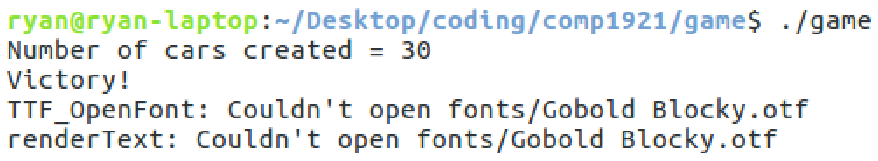
Here we can see that the the user did try to go beyond the confines of the map in the window.

3) Error checking for True Type Fonts

There should be a message after completing the game that says “Victory!” that uses the Gobold Blocky Italic.otf file in the “game/fonts” folder, as well a a “Victory!” message in the terminal.



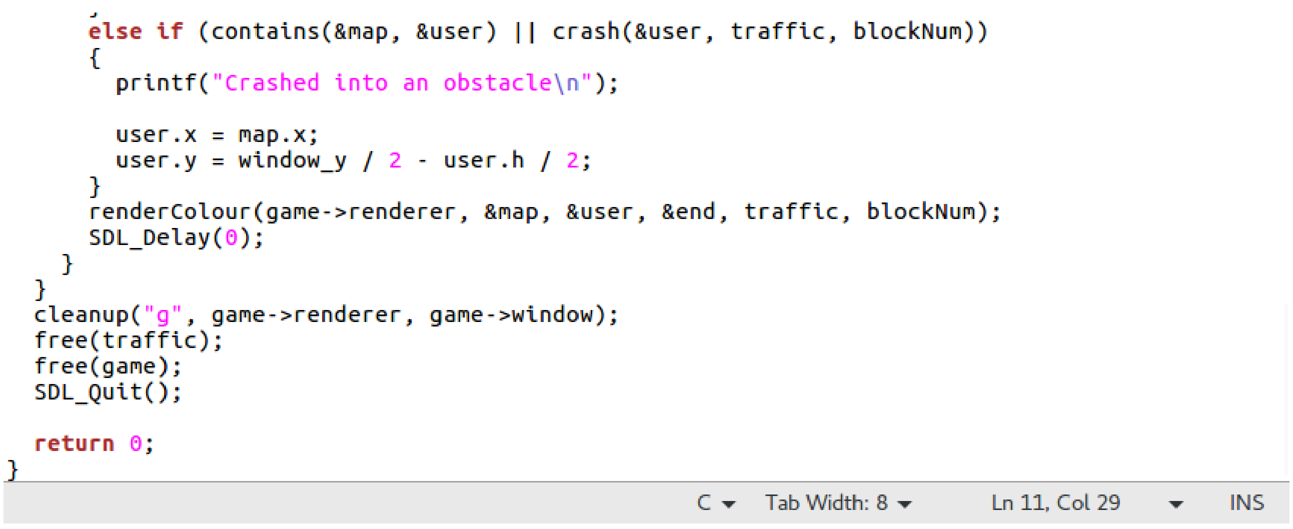
However, when there is not a valid font file in the “game/fonts” it will display an error message as shown below. The game should still run as normal, but not display the message above after the game is completed; instead it should produce an error message.



```
ryan@ryan-laptop:~/Desktop/coding/comp1921/game$ ./game
Number of cars created = 30
Victory!
TTF_OpenFont: Couldn't open fonts/Gobold Blocky.otf
renderText: Couldn't open fonts/Gobold Blocky.otf
```

4) Checking for memory leaks

There should not be any memory leaks in the game since the function “cleanup” destroys the window and renderer, alongside using the free() function.



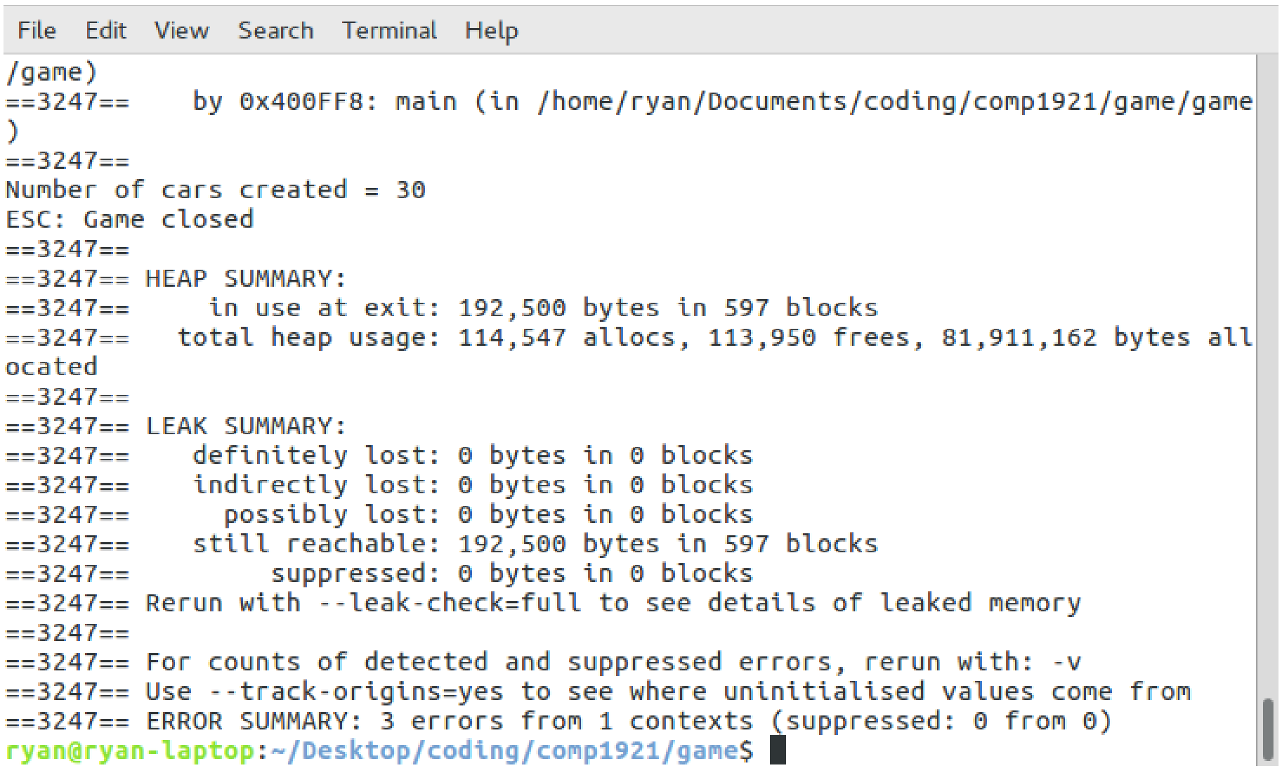
```
      else if (contains(&map, &user) || crash(&user, traffic, blockNum))
      {
        printf("Crashed into an obstacle\n");

        user.x = map.x;
        user.y = window_y / 2 - user.h / 2;
      }
      renderColour(game->renderer, &map, &user, &end, traffic, blockNum);
      SDL_Delay(0);
    }
  }
  cleanup("g", game->renderer, game->window);
  free(traffic);
  free(game);
  SDL_Quit();

  return 0;
}
```

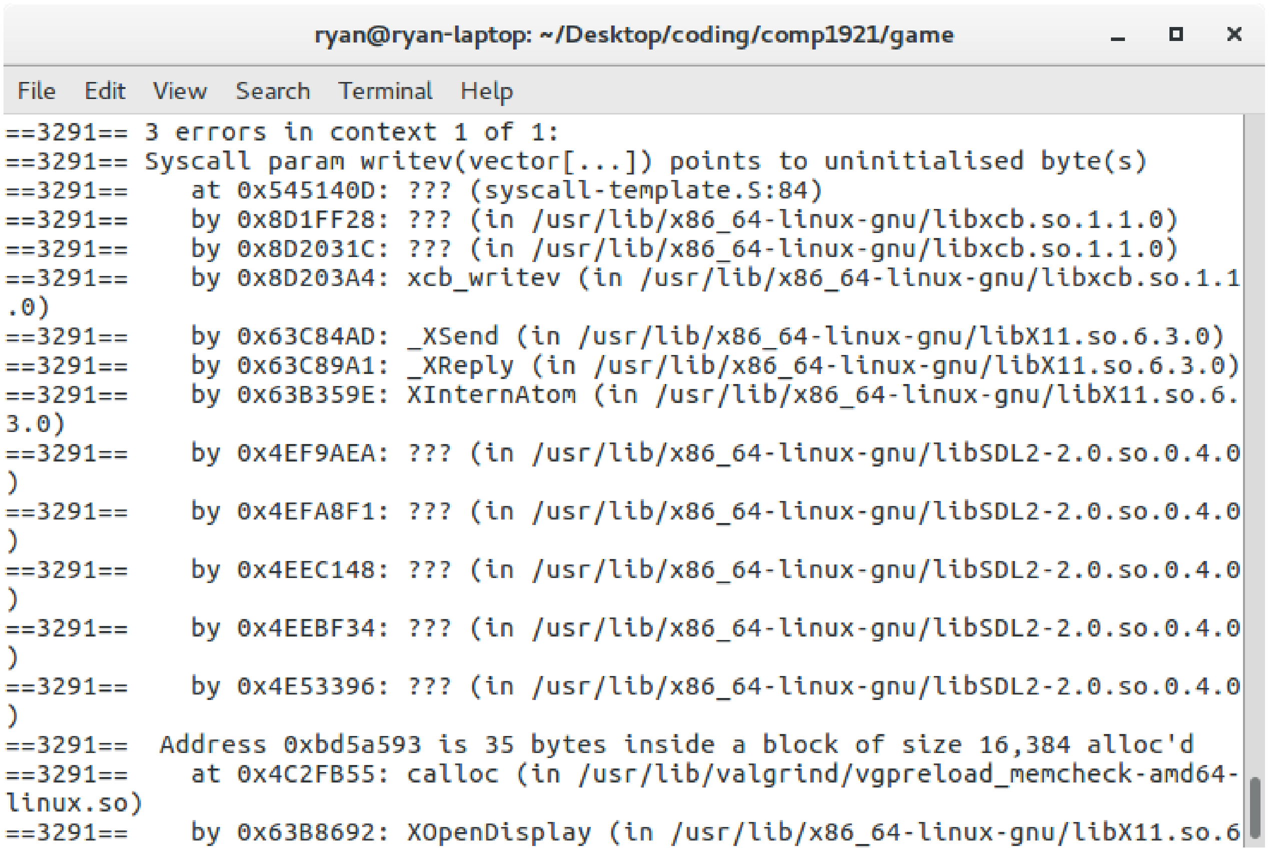
C ▾ Tab Width: 8 ▾ Ln 11, Col 29 ▾ INS

I used valgrind to check that there were no memory leaks.



```
/game)
==3247==    by 0x400FF8: main (in /home/ryan/Documents/coding/comp1921/game/game
)
==3247==
Number of cars created = 30
ESC: Game closed
==3247==
==3247== HEAP SUMMARY:
==3247==     in use at exit: 192,500 bytes in 597 blocks
==3247==   total heap usage: 114,547 allocs, 113,950 frees, 81,911,162 bytes all
ocated
==3247==
==3247== LEAK SUMMARY:
==3247==    definitely lost: 0 bytes in 0 blocks
==3247==    indirectly lost: 0 bytes in 0 blocks
==3247==      possibly lost: 0 bytes in 0 blocks
==3247==    still reachable: 192,500 bytes in 597 blocks
==3247==         suppressed: 0 bytes in 0 blocks
==3247== Rerun with --leak-check=full to see details of leaked memory
==3247==
==3247== For counts of detected and suppressed errors, rerun with: -v
==3247== Use --track-origins=yes to see where uninitialised values come from
==3247== ERROR SUMMARY: 3 errors from 1 contexts (suppressed: 0 from 0)
ryan@ryan-laptop:~/Desktop/coding/comp1921/game$
```

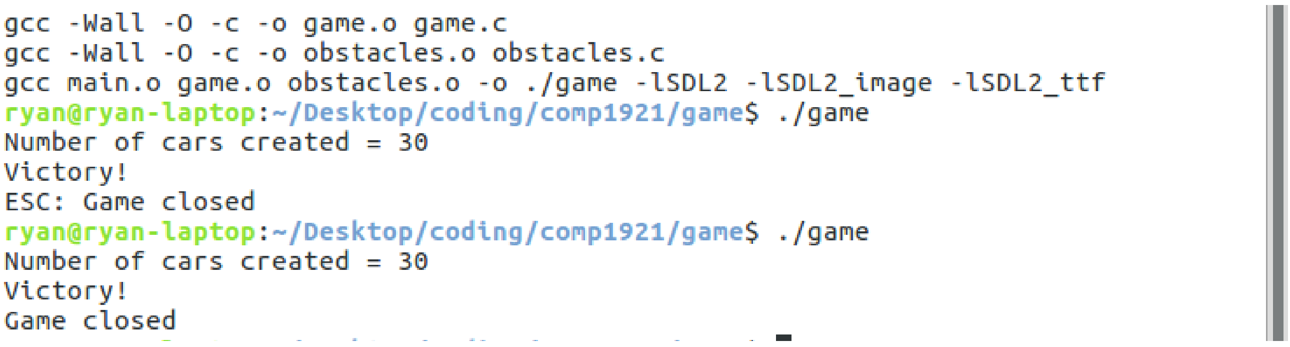
Although there were no obvous leaks, I noticed that there were still some “still reachable” bytes. Upon further investigation of these errors, it turned out that the unfreed allocated memory was coming from the libraries that I was using, as show below. These errors were not to do with the program, so I left them.


ryan@ryan-laptop: ~/Desktop/coding/comp1921/game
File Edit View Search Terminal Help

```
==3291== 3 errors in context 1 of 1:
==3291== Syscall param writev(vector[...]) points to uninitialised byte(s)
==3291==    at 0x545140D: ??? (syscall-template.S:84)
==3291==    by 0x8D1FF28: ??? (in /usr/lib/x86_64-linux-gnu/libxcb.so.1.1.0)
==3291==    by 0x8D2031C: ??? (in /usr/lib/x86_64-linux-gnu/libxcb.so.1.1.0)
==3291==    by 0x8D203A4: xcb_writev (in /usr/lib/x86_64-linux-gnu/libxcb.so.1.1
.0)
==3291==    by 0x63C84AD: _XSend (in /usr/lib/x86_64-linux-gnu/libX11.so.6.3.0)
==3291==    by 0x63C89A1: _XReply (in /usr/lib/x86_64-linux-gnu/libX11.so.6.3.0)
==3291==    by 0x63B359E: XInternAtom (in /usr/lib/x86_64-linux-gnu/libX11.so.6.
3.0)
==3291==    by 0x4EF9AEA: ??? (in /usr/lib/x86_64-linux-gnu/libSDL2-2.0.so.0.4.0
)
==3291==    by 0x4EFA8F1: ??? (in /usr/lib/x86_64-linux-gnu/libSDL2-2.0.so.0.4.0
)
==3291==    by 0x4EEC148: ??? (in /usr/lib/x86_64-linux-gnu/libSDL2-2.0.so.0.4.0
)
==3291==    by 0x4EEBF34: ??? (in /usr/lib/x86_64-linux-gnu/libSDL2-2.0.so.0.4.0
)
==3291==    by 0x4E53396: ??? (in /usr/lib/x86_64-linux-gnu/libSDL2-2.0.so.0.4.0
)
==3291==  Address 0xbd5a593 is 35 bytes inside a block of size 16,384 alloc'd
==3291==    at 0x4C2FB55: calloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-
linux.so)
==3291==    by 0x63B8692: XOpenDisplay (in /usr/lib/x86_64-linux-gnu/libX11.so.6
```

5) Exiting the game

The game does not close automatically because of the True Type Font message after completing the game, so the program must be closed manually. There are two ways of exiting the game, the first is to simply press the ESC key and the other is to use the mouse and click on the X button in the corner of the window. Since ESC is tied to the keyboard input, which is different to using the mouse, I created used two seperate print messages to differentiate them.



```
gcc -Wall -O -c -o game.o game.c
gcc -Wall -O -c -o obstacles.o obstacles.c
gcc main.o game.o obstacles.o -o ./game -lSDL2 -lSDL2_image -lSDL2_ttf
ryan@ryan-laptop:~/Desktop/coding/comp1921/game$ ./game
Number of cars created = 30
Victory!
ESC: Game closed
ryan@ryan-laptop:~/Desktop/coding/comp1921/game$ ./game
Number of cars created = 30
Victory!
Game closed
```

The differences are shown in the messages that are printed out in the terminal with the ESC key explicitly stating that it has been pressed and the X button will only give a normal “Game closed” message.

iv.) Personal reflection

The steps in creating the core functionality of the game was quite a linear process, largely being the same as had outlined in my plan. The renderering of both the user and obstacle squares was relatively simple with the use of SDL_Rect, especially once I had done the user square, setting the same size for both user and obstacle. The colour renderering of the objects required some online research for each colour code. Coding error messages was very useful when it came to testing because I would know, based on the error message that the terminal returned, if the program was catching the correct error like in the third test described earlier.

The most difficult part was having to learning the SDL libraries and what functions that could be used. Integrating and implementing SDL functions with the exisitng C knowledge that I had was challenging because sometimes SDL would have more convient ways of coding relating to graphics like SDL_DestroyWindow to get rid of the window that the game was in. I think that part of this is the lack of experience I had with SDL, but I think that I should have looked at some tutorials sequentially instead just looking for what I needed when coding a certain part or when I ran into a problem.

I have learnt that thinking about the modularisation of code can greatly help with the overall design of your code. This is because the modularisation of code helps to divide the overall project into smaller, more managable tasks that is able to be more easily tested throughout the project. This also helps when adding more features later on in design process since they can slot in quite easily. Writing down psudocode definitely helped here since this was a bigger project with less guidance than we have had before.