Hi Alice,
We've heard so much about your work, it's great to hear you're taking over the project. The current code is hopefully self-explanatory, but we'll run through it for you here, with a few suggestions for directions you might go in.

- The (old!) EtC team

Installing EtC

Clone/download the repository from https://github.com/Kaahu/Escape-the-Clocktower. Open the project file **Escape-the-Clocktower/Application/EtC_Beta/EtC_1.xcodeproj/project.pbxproj** in XCode. Press XCode's build-and-run button.

Overview

Escape the Clocktower is a first-person dungeon crawler/escape the room game, with the theme of escaping the Otago University Clocktower.
Our coding standards are Allman layout, self-explanatory variable and method names, sensible structuring.

Data structures

The two main data structures we've created are the *map* and the *player*.

Map

The *map* is a two dimensional arrays of *MapTile* structs. There is exactly one; different 'levels' are simply separated by impassable areas, with teleportation from the exit of one to the entrance of the next. It is currently nine by eight tiles.
A MapTile struct contains pointers **N, *E, *S, *W* to the images that are displayed when the player is facing north, east, south and west, in the format of an *SDL_Surface*. The same images can be used by multiple tiles, and can be *NULL*.
A MapTile struct contains four boolean values *passable_from_N S, E, W* to indicate whether the player can enter that tile from that direction or not.
There are five variables associated with interacting with a tile:

- a pointer to the SDL_Surface *interact_image* which is displayed after the interaction has occurred
- *map_item* and
- *speaker*, pointers to char arrays for those tiles that contain items or speakers.
- the boolean *is_interactive*, that indicates whether pressing the interact button has an effect in the tile;
- and the character *dir_need_to_face* that specifies the direction the player must be facing to interact with the tile.

Player

The *player* structure contains the position of the player in the world: two integers for the x and y coordinates on the map, and a char that should contain 'N', 'E', 'S' or 'W' to indicate the direction the player is facing.
There are three integer variables that represent whether the player is carrying the *uniform, pineapple* and *apple*; since there are only three such objects, we decided this representation of an inventory was clear and efficient enough for the purpose. If you extend this aspect of the game, considering rewriting the inventory to a bitfield or similar.
The *endgame* variable is used to indicate that the player has reached the end of their play-through.
p is the pointer to the player struct. It is initialised by the method **init_player**.

Keys enum

MYKEYS enumerates the codes that are later used to distinguish pressing the up, down, left and right arrow keys or the spacebar.

IntroImageArray

An array to hold the introduction images with more safety structure than C arrays do; the instance is *image_array*.

Other global variables

done tracks whether the game is finished. It is a boolean value, initialised to false when the game begins, that is set to true when the user presses escape or closes the game window.
redraw tracks whether the graphics need to be updated. It is a boolean value, initialised to true when the **gameloop** begins. It is set to false if nothing happens that requires a graphics update; is confirmed as true if anything does.
key tracks what key has been pressed. It is a pointer to an array of boolean values, initialised to false when the array is declared. The entries in the array are to be accessed using the *MYKEYS* enum for indices.

width and *height* specify that the overall view, including all sections, is a square 480 pixels across.

columns and *rows* are integers used in the **load_map_file** method.

window, renderer, texture, border/text/inv_surface/texture and *free_image* are pointers to SDL data types that are used to display the images. If the SDL library becomes unusable for any reason, these will need to be replaced.
view_rect, text_rect and *inv_rect* define the size and position of the main view, text display, and inventory box.

num_intro_screens is the integer number of introduction screens, counted in **load_starting_images_from_line**.

Methods

abort_game ends the game in case of error. It takes an error message as a parameter, prints that message, and exits the program.

Main

The main method has the standard C parameters. It calls three methods, none of which take parameters or return values, then ends successfully.

init sets up the game. It calls a lot of methods:

- **load_starting_game_vars** opens the initialisation file, which is currently named init.txt in the same directory as the main.c file. It checks the file exists, then
  image_array_init creates the array for the intro images, in an empty state.
  It checks the file is the right length using
  check_min_num_lines_file; currently the right length is exactly 5 lines of up to 256 characters, so double-check that if your requirements change.

The five lines are currently the file name for the map data, a list of all the intro screens, then the starting x and y coordinate and direction.

load_starting_images_from_line splits up the file name of the introduction images on the line, counts them into *num_intro_screens,* and loads them into *IntroImageArray*.

check_char_is_num_in_range checks that the starting x and y coordinate are valid places to start.

Then it closes the init file.

- **init_map** creates the map. It sets the number of columns and rows then calls

  load_map_file takes a char array as input: the name of the file with the map data in it. It checks the file exists, then reads the lines to the map cells left to right and top to bottom.

  read_line_to_map takes three parameters: a pointer to the line of characters being read, and the integer x and y coordinates of the current map cell. It splits up the line into tokens and goes through each one. The first four tokens should be either NULL or the location of a BMP image corresponding to the view in facing North, East, South or West from the current cell. These tokens are passed into:

  check_image_null_or_load uses SDL to load a BMP file if it's given one, or sets that map cell view to the null pointer if the map file says 'NULL' there. Then it checks if it really did load the image using:

  check_image_loaded aborts the game if the SDL_Surface passed into it doesn't exist.

  The next five tokens should be 'true' or false; the first four referring to whether the tile can be entered from the four adjacent tiles, and fifth being whether there is an interaction within the tile.

  check_true_or_false converts the string to a boolean value.

  This is followed by the image displayed after interacting, and the direction the player needs to face to interact, using '0' if there is no interaction. The last two tokens are for those tiles that contain one of the items or speakers of the plot.

- **init_player** initialises the pointer p. It calls player_constructor to create the player struct, and sets the global variable p to point to that struct. The initial location on the map is set by passing the coordinates and direction into this method, then on to the constructor.

init then sets up SDL for the session. It initialises the assets from SDL: the library itself, window, renderer. It loads images the border, text and inventory sections, title image and display.

set_text takes the location of an image file and returns nothing. It creates the textures and surfaces for the text section, checking that this succeeds.

set_inv takes the location of an image file and returns nothing. It creates the textures and surfaces for the inventory section.
These are used repeatedly throughout the program to update these sections. At initialisation they are set to blank template images.

init checks that these have succeeded, and calls abort_game if not.

game_loop runs the game. While the game is not finished [*done*] it uses the SDL WaitEvent method to watch for input. Escape or closing the game window end the game. Otherwise, it calls get_user_input to handle user input. Then it calls update_graphics to redraw the game display if necessary.

shutdown closes the game. It calls functions to destroy the SDL texture, surface, renderer and window, the map data, the name of the map file, intro image array, and the player struct.
free_map_speaker_and_item and **free_map_array** free all the images, speakers, and items in the map that exist.

Player methods

player_get_position_x,
player_get_position_y and
player_get_direction_facing take the player struct as a parameter, and return the x coordinate, y coordinate and current direction.

player_set_position takes the player struct and two integer coordinates as parameters. Sets the player's new coordinates to the input coordinates.
player_set_direction takes the player struct and a single character direction as parameters. Sets the player's new direction to the input direction.

player_constructor makes the player at the starting point of the map. It allocates memory for the player struct, sets the position of the player, notes that the player is carrying nothing, and returns a pointer to this struct.
player_destruct takes player struct as a parameter, frees it. If you extend the struct so it needs other allocations, the corresponding frees will go here.

Gameplay methods

get_user_input takes the keypress event as a parameter and checks what key was pressed. It has no return value. 'Up', 'left' and 'right' cause the relevant value in the keys[] array to be set to true.

If up arrow key: update the player location by 1 in the direction that the player is facing, after checking is that tile to be entered is passable in that direction. Moving East or West is currently disabled. If movement is successful, it requires graphics update.
If left or right arrow key: update the direction that the player is facing, counterclockwise or clockwise respectively. Requires graphics update.
If 'space' is pressed, the player may be teleported to the next level if in the exit of the first level, or else will check for interactions. Requires graphics update.
Pressing 'escape' will exit the game, which is the only way to actually escape the clocktower, because we find that funny.
No other keypresses require the graphics to update.

graphics_show_direction_facing takes and returns nothing. It checks the direction that the player is facing, purges the current view, and sets *texture* to the corresponding image.
graphics_render_multiple_texture takes and returns nothing. It uses SDL to clear the window then show the four components of the display at once: main view, text, inventory, and the borders around them.
update_graphics changes the graphics to reflect the player's actions. It is called by the gameplay loop after get_user_input, and depends on that method, but through the global variables *num_intro_screens* and *keys[]*: it does not take parameters or return values. It does reset the key[] array to be ready for the next keypress.
If 'space' was pressed and there are intro screens to display, the game steps through to the next intro screen. After the last intro screen, the multi-part normal view is set up.
If 'space' is pressed later, the method checks if the player's current tile and direction faced can be interacted with, fetches the new image, sets the tile to be uninteractable, and sets the relevant direction image for the current tile to the new image. (If it isn't interactable, the current image is shown.) Then it calls run_conversation.
If 'up', 'left' or 'right' was pressed, the method simply resets the keys and shows the direction now being faced.
run_conversation takes the player struct as input, and basically runs the entire plot. The first door guard won't let you out unless you're wearing clothes; the second, unless you're carrying a pineapple. The third one wants an apple, and the apple dispenser wants the mechanic's attention, and the mechanic wants that pineapple. graphics_update resets interacted-with tiles to uninteractable; if the interaction must continue, this method sets it back to interactable and changes the interact image to the next in the sequence.
get_item takes the player struct and the name of an item as input. It updates the flags when an item is picked up, and also changes the relevant map tiles' passability and images.

restart_game takes no input and returns nothing. It resets the game to its initial state—deleting the present map data and initialising it afresh, refreshing the display and returning the player to the starting position—with one exception: the player is first told to press space to *WAKE UP*. But the nightmare cannot be escaped that way.

Map Methods

check_true_or_false is used to check passability.
load_map_file loads the external map file. It takes the file name as input, gets each line in turn, reads it, and closes the file when done.
read_line_to_map reads the map data. It reads in 10 items per line: the four images for the directions the player is facing, the four passable conditions for those directions, the image or lack thereof to be displayed if the square is interacted with, and the direction that the player must be facing to interact successfully.
init_map initialises the map.

Intro Image Methods

We implement some very standard management methods for a variable-size array.
image_array_init allocates space for the array, and starts it off empty.
image_array_push adds an image to the end of the array, doubling the array size if necessary.
image_array_length returns the number of images in the array.
image_array_insert inserts an image at a valid index in the array, replacing what is there.
image_array_free frees all the data in the array, then the array itself.
image_array_get returns the image stored at a valid index in the array.

Other Management methods

check_texture_and_destroy takes a pointer to an SDL texture, checks if there is a texture there, and destroys it if there is. Used to ensure that the display doesn't show old information when it should update.
check_texture_loaded takes a pointer to an SDL texture, checks if there is a texture there, and aborts the game if there isn't. Used to avoid empty sections of the display, ever.
clear_inv and
clear_text destroy the current textures in their areas, and reset the section to a default template image.

Going Forward

There have been plenty of bugs around memory allocation, particularly around changing the images displayed while moving and with the way the game loops until escaped. We think we've got them all, but of course it's always safe to assume there may be more.
The current build relies on XCode, which Apple is unlikely to suddenly cancel, but you might need to rewrite the software to avoid that dependency one day. Likewise with SDL, as previously mentioned.