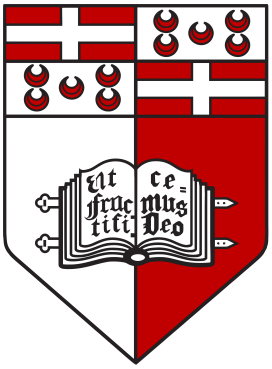

CPS2003 - Systems Programming
-
Assignment 2016/2017

Full name: Miguel Dingli
I.D: 49997M
Course: B.Sc. (Hons) in Computing Science

Table of Contents

1 – Introduction

As an introduction, note that the code in the project was subdivided into three main sections; **client**, **server**, and **general**. The *client* and *server* sections both have their own Main file with a main method which, when executed, runs the client or server, respectively. The *general* section includes structures, enumerations and functions that are common to the client and server and is meant to enforce standardization and reduce code duplication.

2 – Representations

2.1 – Representation of a Map

2.1.1 – Server Side

In the server side, a map is simply represented as a two-dimensional array of characters, where a character can be a blank space, a snake character ('o'), a fruit character ('@'), or a wall character ('#'), indicating the bounds of the map. The map size is dynamic in that it is set by a command-line argument. Any updates that are applied to this map are also sent to each client so that their respective maps are kept up to date.

2.1.2 – Client Side

In the client side, a map is also represented as a two-dimensional array of characters. A map update that is received from the server is represented by a **MapChange** struct (Figure 1), consisting of the coordinates of the location on the map which will change, and the new character that will occupy the location.

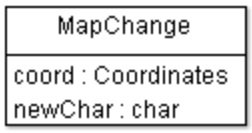

Figure 1 – MapChange Struct

2.2 – Representation of a Snake

2.2.1 – Server Side

In the server side, a snake is represented by a **Snake** struct (Figure 2). This struct conveniently uses the client's socket descriptor as a unique identifier. It also includes the snake's length, current direction, a pointer to the head and tail, and a boolean indicating whether the snake is playing or is a spectator. Note how the head and tail are pointers to a **SnakePart** struct (Figure 3). A snake's parts are essentially a linked list where each node stores a coordinate that the snake occupies and a pointer to the next snake part.

Along with these two structs, functions to create a new snake, to move the snake in its current direction, to lengthen the snake, and to free the snake (i.e. free(...) applied to its parts) are defined in Snake.h/.c. For a minimum length of 3, it is assumed that function *lengthen* will be called twice for each new snake.

Note that the server maintains a list of snakes which is updated whenever a client joins or exits the game.

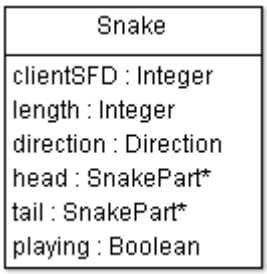

Figure 2 – Snake Struct

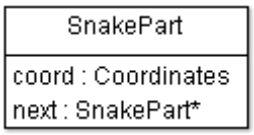

Figure 3 – SnakePart Struct

2.2.2 – Client Side

In the client side, a snake is essentially only represented by a single pair of x-y coordinates indicating the location of the snake's head. This coordinate is used to center the client's screen on the snake that the client is controlling. The remainder of the snake, and other clients' snake are represented through the characters in the map rather than through a list of snakes.

2.3 – Representation of a Fruit

2.3.1 – Server Side

In the server side, the coordinates of the fruit that is currently in the map are stored. Whenever the fruit is eaten, a new fruit is created and the map is updated accordingly. There is always exactly one fruit in the map at any point in time. Any logic having to do with snakes interacting with fruit is performed in the server side.

2.3.2 – Client Side

In the client side, a fruit is represented through a character of a fruit on the map. No coordinates of the fruit are stored on the client side. The fruit character only changes place when a map update is received.

3 – Communication

3.1 – Packets

Communication between the client and server happens through the sending of packets. A packet is represented by a **Packet** struct (Figure 4), which includes a header and an array of bytes (using *unsigned char*). The packet's header is represented by a **PacketHeader** struct (Figure 5), which stores the packet type and size. The array of bytes is essentially the data being sent between the server and client.

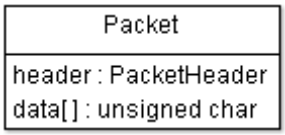

Figure 4 – Packet Struct

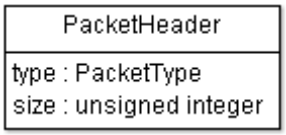

Figure 5 – PacketHeader Struct

3.1.1 – Types of Packets

Throughout the game, only four types of packets can be sent between the server and client. These are:

- **Game details**, where the packet data corresponds to a **GameDetails** struct (Listing 6). This packet is sent from the server to the client once, when the client joins the game. It consists of the snake head's start coordinates, the map's x and y coordinate limits, and an array of map changes that will bring the client up to date with the server's map, and includes all snake and fruit characters.
- **Game update**, where the packet data corresponds to a **GameUpdate** struct (Listing 7). A packet of this type is sent from the server to each client at every time period. For each snake, it consists of the new coordinates of the snake's head (unique for each client) after it has moved, and an array of map changes consisting of the latest changes that were applied to the server map. For every snake that moved, there will be a snake character added at the new snake's head, and a blank character added at the old location of the snake's tail.
- **Direction change**, where the packet data corresponds to a value from a **Direction** enumeration. This packet is sent form a client to the server whenever the client decides to change the direction of the snake. This is done using the keys 'w' (up), 'a' (left), 's' (down), and 'd' (right).
- **Game message**, where the packet data corresponds to a value from a **GameMessage** enumerat-ion. This packet is sent from the server to clients whenever an event occurs in the game. The enum value is used as an index to an array of strings containing the messages. The resultant message is displayed on the screen of each client that was sent the message.

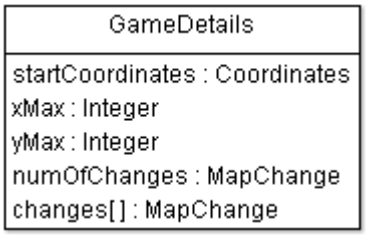

Figure 6 – GameDetails Struct

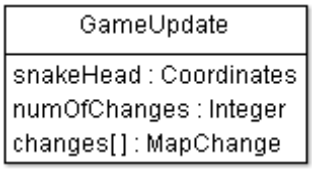

Figure 7 – GameUpdate Struct

3.1.2 – Functions for Sending and Receiving Packets

For sending and receiving packets, a function is defined for each purpose in the general section of the project (specifically, in General.c/.h).

The $recv\_packet(\dots)$ function takes a socket descriptor through which a packet is expected to arrive and creates a new packet populated with the received packet type and data when a packet arrives. On the other hand, the $send\_packet(\dots)$ function takes the socket descriptor of the socket to which the packet will be send, the packet type, packet size, and packet data, and sends this information to the receiving end based on the socket descriptor.

3.2 – Serialization and Deserialization Functions

An array of bytes in a packet has no form on its own. The data in a packet is given form by taking note of the packet type and converting the packet data to its expected form. Similarly, a form of conversion is needed to convert a structure of data into an array of bytes. These conversions are achieved through the use of serialization and deserialization functions.

3.2.1 – Serialization and Deserialization of Packet Data

When a packet is to be sent, a serialization function (or multiple thereof) is used to convert the data to be sent from a variable or struct into a sequence of bytes. Oppositely, when a packet is to be received, a deserialziation function (or multiple thereof) converts the sequence of bytes in a packet back to the original variable or struct representation. The serialization and deserialization functions cater for all possible types of packets:

- **Game details**: uses $serialize\_gameDetails(\dots)$ and multiple deserialization functions (consisting of deserialization functions for integers, coordinates, and an array of map changes).
- **Game update**: uses $serialize\_gameUpdate(\dots)$ and multiple deserialization functions (consisting of deserialization functions for integers, coordinates, and an array of map changes).
- **Direction change**: uses $serialize\_int(\dots)$ and $deserialize\_int(\dots)$
- **Game message**: uses $serialize\_int(\dots)$ and $deserialize\_int(\dots)$

Note how the direction change and game message do not have dedicated serialization functions. This is because these just consist of enumeration values that are essentially integers. Note also that since the structs involved in game details and game update consist of various types, the serialization and deserialization functions for these make use of a mix of more specific serialization and deserialization functions:

- $serialize\_int(\dots)$ and $deserialize\_int(\dots)$
- $serialize\_coordinates(\dots)$ and $deserialize\_coordinates(\dots)$
- $serialize\_mapChange(\dots)$ and $deserialize\_mapChange(\dots)$
- $deserialize\_mapChangeArray(\dots)$

Consider the serialization functions for game details and game update. Since the data held by the server is not stored in GameDetails and GameUpdate structs, the serialization functions for these do not take these structs as arguments per se, but instead provide all the values that make up the struct which can then be serialized in a way that the client can convert the data into the appropriate struct.

3.2.2 – Serialization and Deserialization of Packets

Packets themselves, or more specifically, a packet's header also makes use of serialization and deserialization functions since it will also be transmitted as a sequence of bytes. For the packet type and packet size, the integer serialization/deserialization functions are used. The packet data is simply sent as is, since it is already in the form of a sequence of bytes. The use of these functions for the packet header can be observed in the functions discussed in section 3.1.2, i.e. $recv\_packet(\dots)$ and $send\_packet(\dots)$.

3.2.3 – Byte Ordering

The serialization and deserialization functions for an integer make use of the functions $\boldsymbol{htonl}(\dots)$ and $\boldsymbol{ntohl}(\dots)$, respectively, to convert between network byte order and host byte order. This is done to ensure that no issues arise due to the endianness of the system running the game server or a client.

4 – Running the Game

4.1 – Start-Up

In this section, the start-up routines for both the server and client will be presented.

4.1.1 – Server Start-Up

The start-up routine for the server is presented in Figure 8.

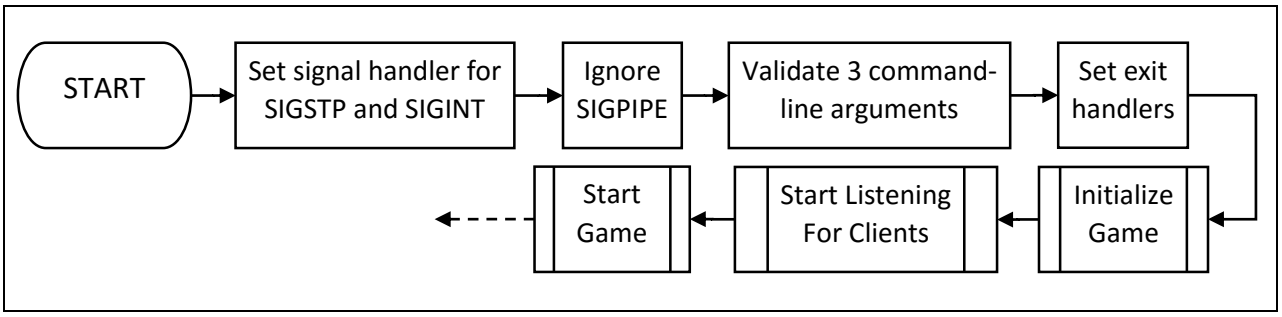

Figure 8 – Server Start-Up

The following are the main points regarding the above routine:

- Since the server can be terminated by CTRL+C or using the kill command, a signal handler was set to handle the two signals SIGINT and SIGSTP so that when any of the above two actions happen, the signal handler runs. The signal handler simply calls $exit(EXIT\_SUCCESS)$ so that the exit handlers are executed.
- The signal SIGPIPE which can occur when a client loses connection can cause a program to stop. For this reason, this signal is ignored and is manually handled whenever a send or read is performed.
- The three command-line arguments expected for the server are the **server port**, **map width**, and **map height**. A simply checks verifies that these are integers.
- In the server, three exit handlers are registered in the following order; one to close the server socket, one to close client sockets, and one to close down the game. Since exit handlers are executed in reverse, the game is first closed down, which includes deallocation of memory and sending a final message to clients.
- The **Initialize Game** block includes initializing the map and its bounds (based on the arguments to the program), adding a fruit, initializing a snakes list, and initializing a mutex lock for the snakes list and fruit.
- The **Start Listening For Clients** block includes creating the server's TCP/IP socket using *socket(...)*, binding the host address using *bind(...)*, and starting to listen for clients, using the *listen(...)* function.
- The **Start Game** block includes the creation of two *pthreads* that will be discussed in section 4.2.1.

4.1.2 – Client Start-Up

The start-up routine for the client is presented in Figure 9.

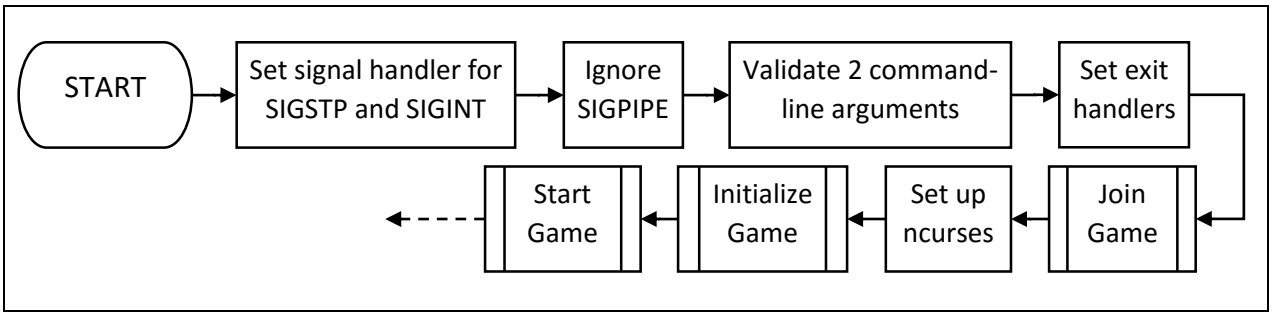

Figure 9 – Client Start-Up

The following are the main points regarding the above routine:

- Similar to the server, a signal handler is set for SIGSTP and SIGINT, and signal SIGPIPE is ignored.
- In the case of the client, two command-line arguments are expected; the server's **hostname** and the server's **port**. A simple check verifies that the port is an integer.
- The exit handlers in the client side are registered in the following order: one to close the socket used to communicate with the server, and one to close down the game, which is mainly deallocation of memory.
- The **Join Game** block includes finding the host using *gethostbyname(...)*, creating a TCP/IP socket using *socket(...)*, and connecting to the server using *connect(...)*.
- The **Initialize Game** block for the client includes obtaining the initial data (i.e. GameDetails) from the server, initializing the snake head, map bounds, and the map itself, applying the GameDetails changes to the map (including characters for the snakes and fruit), and initializing a mutex lock for the map.
- The **Start Game** block includes the creation of three *pthreads* that will be discussed in section 4.2.2.

4.2 – Threads

4.2.1 – Server Threads

The server program makes use of three types of threads, each using one from the below three functions. Each function consists of a loop with a condition that is satisfied as long as the game is running.

- $thread\_acceptClientsLoop(\dots)$ (refer to section 4.2.1.1)
- $thread\_mainLoop(\dots)$ (refer to section 4.2.1.2)
- $thread\_directionsLoop(\dots)$ (refer to section 4.2.1.3)

Whereas only one thread exists for each of the first two thread functions, a thread for the directions loop thread function exists for each client connected to the server. Additionally, the threads for the first two functions are created at the Start Game period (as seen in Figure 8) while the server is starting up, whereas threads using $thread\_directionsLoop(\dots)$ are created by the thread corresponding to $thread\_acceptClientsLoop(\dots)$.

4.2.1.1 – Accept Clients Loop

The following is a flow diagram indicating the responsibilities and behaviour of $thread\_acceptClientsLoop(\dots)$:

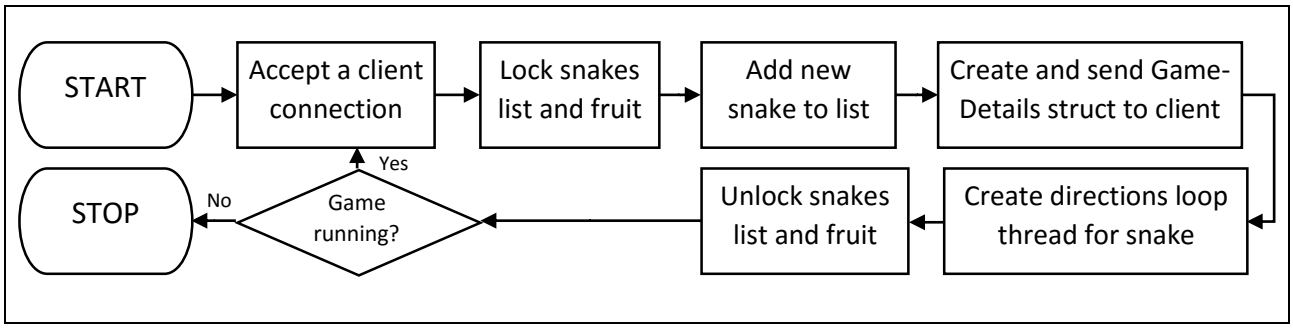

Figure 10 – Accept Clients Loop

This loop accepts one new client at a time, creates and adds a new snake to the snakes list for the client, creates and sends a GameDetails struct to the client, and starts a directions loop for the client.

4.2.1.2 – Main Loop

The following is a flow diagram indicating the responsibilities and behaviour of $thread\_mainLoop(\dots)$:

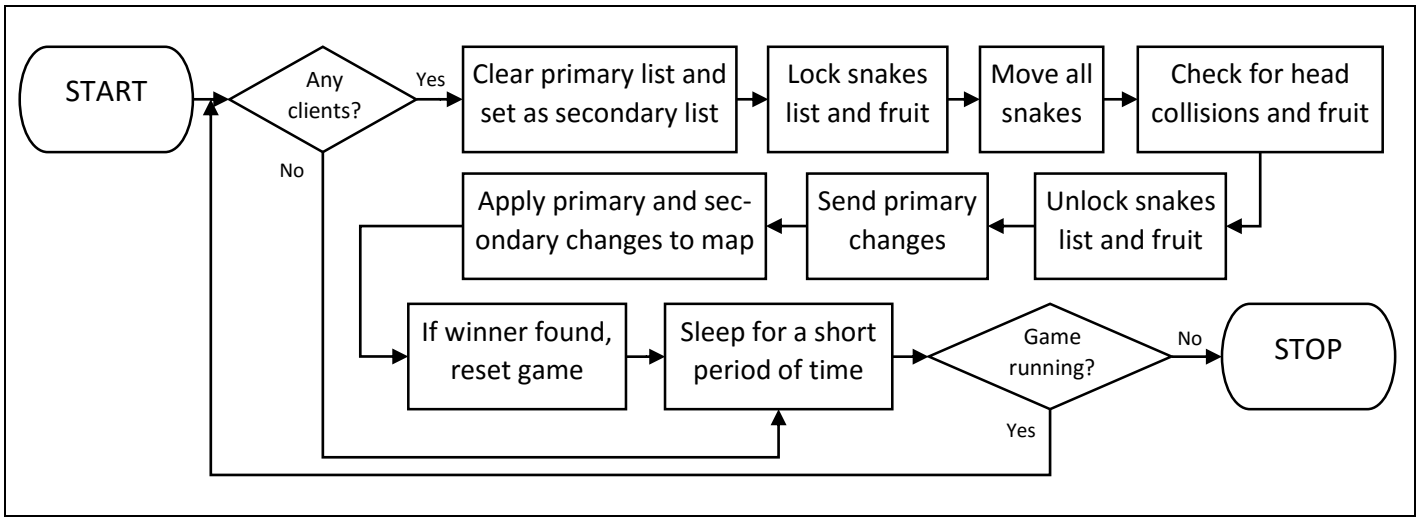

Figure 11 - Main Loop

This loop consists of most of the logic in the game. Notice the notions of a primary and secondary list. These are lists of map changes, where the primary list holds changes accumulated when the snakes are being moved, while the secondary list holds changes caused by snakes leaving the game, since these have to be cleared.

In the 'Move all snakes' period, for each snake, it is checked that the character at the new snake head's coordinates is either blank or a fruit and that the map limits were not exceeded. Otherwise, a collision has occurred. The primary changes are then sent to the clients in the form of a GameUpdate. Both the primary and secondary changes are now applied to the map, and if the winner was found, the game is reset. This involves clearing the map and providing a new snake for the players. Note that the secondary changes are sent to the players in the next loop by setting the primary list to the secondary list at the start of the main loop.

4.2.1.3 – Directions Loop

The following is a flow diagram indicating the responsibilities and behaviour of $thread\_directionsLoop(\dots)$:

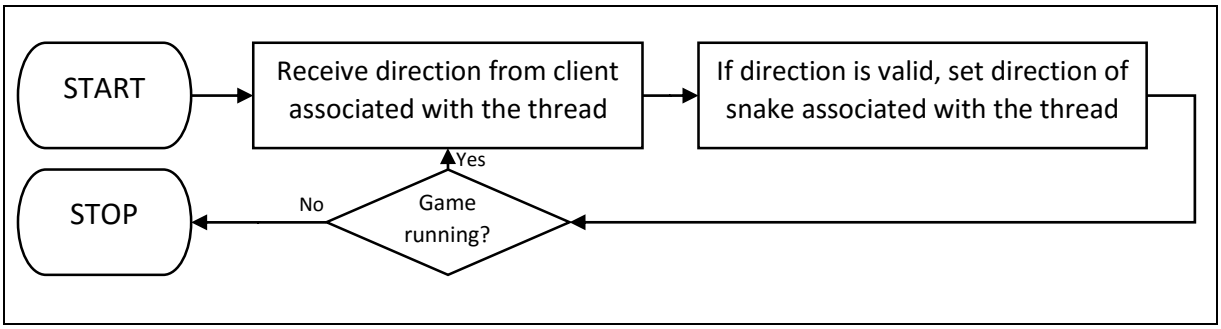

Figure 12 – Directions Loop

As stated previously, a thread for the directions loop exists for each client connected to the server. For each client, this thread waits for a direction from the client and applies this to the corresponding snake if the direction is valid. A valid direction is a direction that is not the opposite of the current snake's direction.

4.2.2 – Client Threads

The client program also makes use of three types of threads, each using one from the below three functions. Each function consists of a loop with a condition that is satisfied as long as the game is running.

- $thread\_mapPrintLoop(\dots)$ (refer to section 4.2.2.1)
- $thread\_updatesLoop(\dots)$ (refer to section 4.2.2.2)
- $thread\_directionsLoop(\dots)$ (refer to section 4.2.2.3)

For each of the above thread functions, only one thread exists throughout the client's lifetime. Additionally, all three of these threads are created at the Start Game period (as seen in Figure 9) while the client is starting up.

4.2.2.1 – Map Print Loop

The following is a flow diagram indicating the responsibilities and behaviour of $thread\_mapPrintLoop(\dots)$:

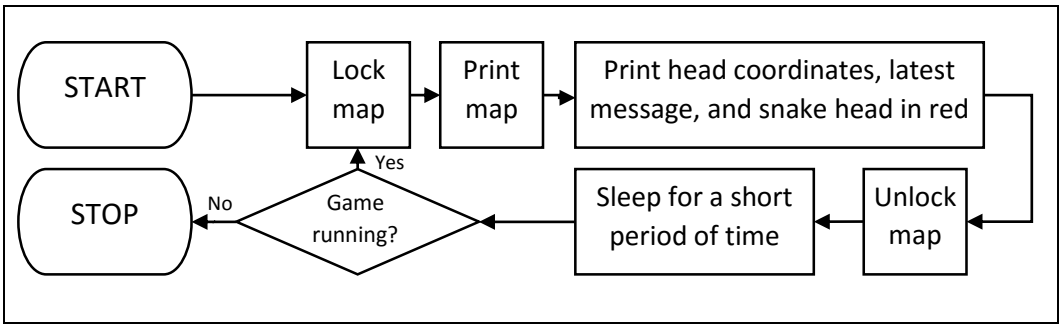

Figure 13 – Map Print Loop

The above loop is responsible for printing the map on the client's screen based on the stored array of characters (i.e. the map). Note that besides the map, the coordinates of the snake's head and the latest message from the server are displayed at the top left of the screen, and the snake's head is displayed as a red character, rather than black. The map is printed every time period, which must be smaller than that of the main loop in the server.

4.2.2.2 – Updates Loop

The following is a flow diagram indicating the responsibilities and behaviour of $thread\_updatesLoop(\dots)$:

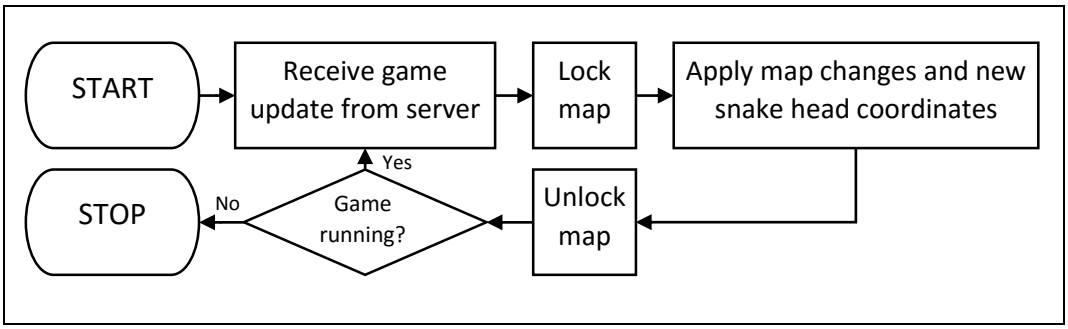

Figure 14 – Updates Loop

The above loop is responsible for receiving game updates from the server and applyin the changes received. This includes changes to the map and the latest coordinates of the snake's head.

4.2.2.3 – Directions Loop

The following is a flow diagram indicating the responsibilities and behaviour of $thread\_directionsLoop(\dots)$:

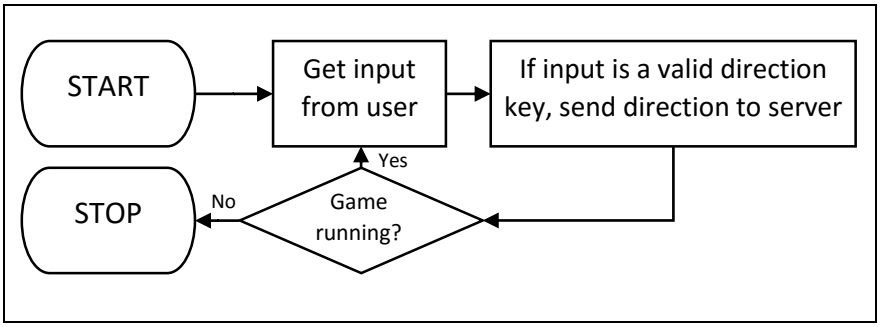

Figure 15 – Directions Loop

The above loop is responsible for reading the client's input to the terminal (through ncurses), and if this input is a valid direction key (i.e. 'w', 'a', 's', or 'd'), then the corresponding direction is sent to the server.

5 – Other Mechanisms and Features

5.1 – Introduction

This section will present implemented mechanisms that were either not discussed in the previous sections or were only brielf discussed.

5.2 – Linked List

In the general section of the project, two structs define a linked list data structure that is used in the server side for the snakes list and the lists of map changes (i.e. both the *primary* and *secondary* lists, from section 4.2.1.2). The structs used are **List** (Figure 16) and **Node** (Figure 17). Note that these are similar to the Snake annd SnakePart structs. The reason why these structs were not used to represent snakes is that a snake requires information such as the socket descriptor, and current direction, and not just the "node count", corresponding to the snake length. Additionally, no conversion of pointers from $void*$ to SnakePart* is required when dealing with Snake structs.

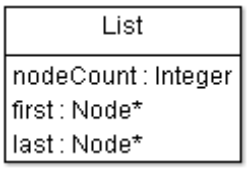

Figure 16 – List Struct

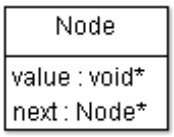

Figure 17 – Node Struct

A collection of functions to operate on this data structure were also provided. These include a function to create a list, to add a new value to the list, to clear the list, and a function to clear a specific node. Note that values in the linked list are stored as general pointers (void *) so as to support any type of value. The functions to clear the list and to clear a node both take a pointer to a function that is applied to the nodes to deallocate their memory.

5.3 – Spectator Mode

Once a snake collides with another snake (including itself) or hits a map bound, the client is not disconnected from the server, but is allowed to stay in the server and observe other snakes move, despite not being able to move. If another snake eventually wins, the game is reset and the spectators will be able to play.

Note that if all the clients in the game are spectators, the game does not reset on its own, meaning that spectators need to wait for a snake to win. However, clients can always disconnect and reconnect to be able to play again.

5.4 – Dynamic Map and the Client's Screen

As discussed in section 2.2.1, the map size is set by a command-line argument. This means that a map is very likely to be greater than the size of a client's terminal. For this reason, the client's screen was designed in a way that only a section of the map that fits in the terminal, centered around the snake's head, is shown. Figure 18 should be of help when going through the $thread\_mapPrintLoop(\dots)$ function in the client's Game.c file. It represents the case where the client's screen is within the map bounds and is smaller than the map.

Note that the fact that the map was made dynamic in size means that the game's requirement that there should be no limit on the number of clients becomes more realistic since the map's size can be specified according to the expected number of clients playing at the same time, for a better game experience.

Additionally, the size of a map has no effect on the size of the data transferred between the server and clients since the game was designed in a way that only map changes are required by the client, and not the whole map.

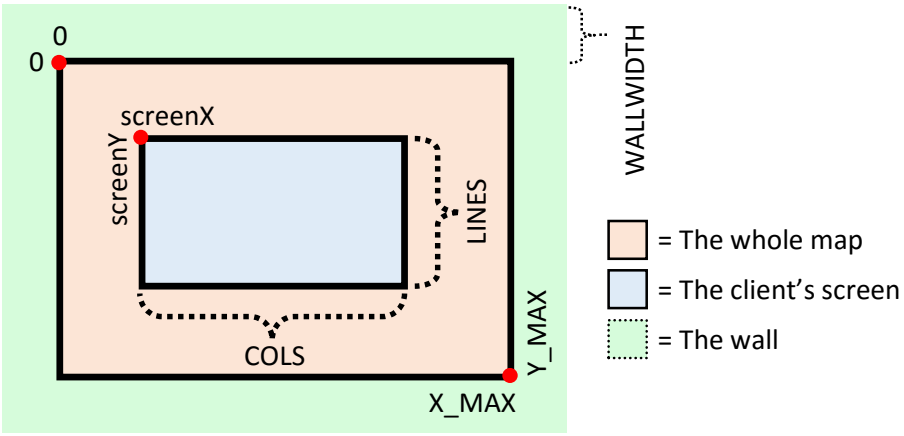

Figure 18 – Map and Client's Screen

5.5 – Error Handling

It is worth mentioning that an effort was made to handle many forms of errors that may occur during the game, especially in the server side when transmission of data to clients is involved, so that the server does not crash.

In the server's directions loop for a particular client, if the function for receiving a direction fails, the direction changes from the snake are disabled as it is assumed that the client disconnected. In this case, in the server's main loop, once the sending of a game update to the snake fails, the client's snake is erased from the game.

In the client side, when an error such as one arising from the sending or receiving of packets occurs, the next action is usually not to try to save the client, but to gracefully close down the client by use of the exit handlers.

6 – Bugs and Issues

The following is a list of known bugs and issues that are present in the game and which were not yet solved:

- If a snake changes direction twice before making a move, this can potentially result in the snake moving back through its own body. When moving in a particular direction, the snake is not allowed to change direction to the opposite direction. However, if within a single cycle the direction is changed in such a way that the "opposite direction" is no longer the opposite followed by the direction changing to this "opposite direction", the next move that the snake makes will effectively be the opposite of the previous moving direction, resulting in the snake trying to move through its body and hence colliding with itself. **Potential solution**: a solution to this would be to only allow the snake to change direction once in a single cycle.
- During the snake moving period, the characters on the map are checked to see if a snake will land on another snake. At this point, the map has not yet been updated with the latest moves, so it includes extra characters where the tail of each snake was located and excludes characters where the new heads of each snake will be located. Collisions between new head locations are handled after the moving period; however, if a snake lands on one of the extra tail characters, this is still counted as a collision. **Potential solution**: a solution to this would be to separately store the new coordinates of each snake's head and to then check if any of these coordinates match with the coordinates of any snake part, using the snakes list.
- Clients which are located on the same machine as the server do not make use of alternate IPC mechanism for performance and efficiency. However, due to the minimal amount of data that is transmitted between the server and clients, this was not assumed to be a big issue. If an alternate IPC mechanism was used, this would probably have been shared memory, where the map is shared to avoid having to send updates.
- A small issue that does not seem to cause long-term problems is the secondary list of map change. Since the changes in this list are applied to the map immediately rather than waiting for the next round, and since these changes are moved to the primary list which is then re-applied to the map, the changes in the secondary list are effectively applied twice to the map. However, this list only consists of characters erased due to disconnected snakes. On the same note, if a client joins mid-game, this client might be sent secondary changes that will try to erase a disconnected snake that was not even sent to the client. Again, since the list only consists of erased characters, this does not seem to cause any long-term problems.
- Finally, it seems as though there is some memory leakage. In the client side, some of this can be attributed to ncurses which leaks some memory. Fortunately, the memory leaks present in the server side do not seem to be proportional to the number of connected clients or the size of the map.

7 – Testing

7.1 – Snake Movement

Figure 19 shows normal snake movement in a stright line.

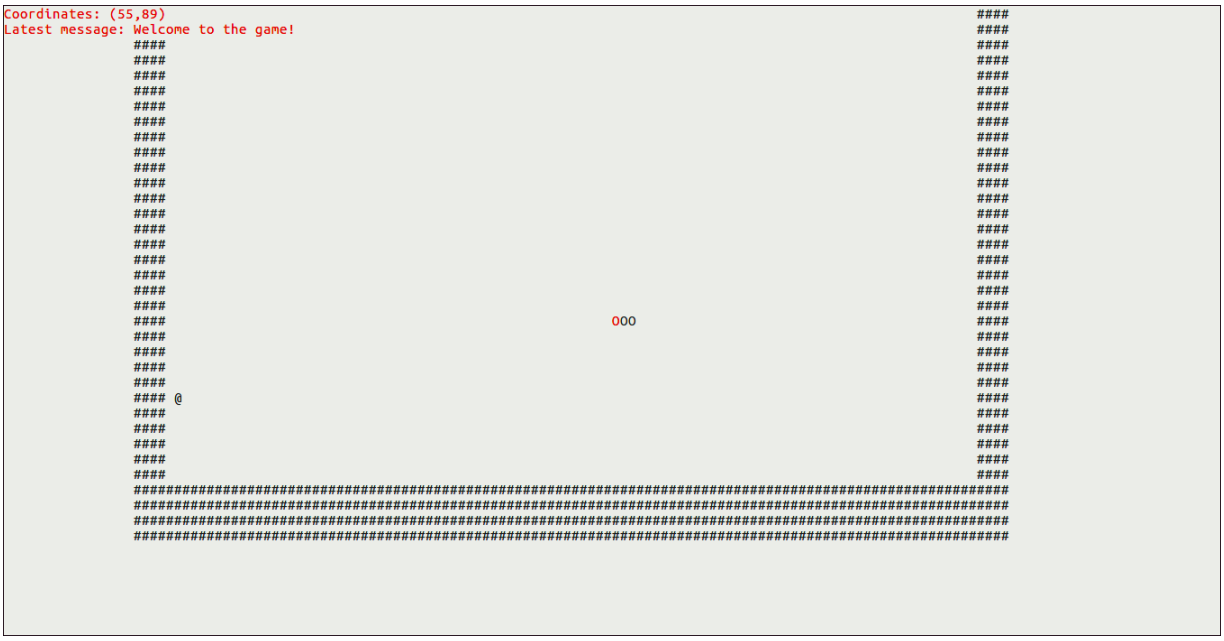

Figure 19 – Normal Snake Movement in a Straight Line

Figure 20 shows a change in direction.

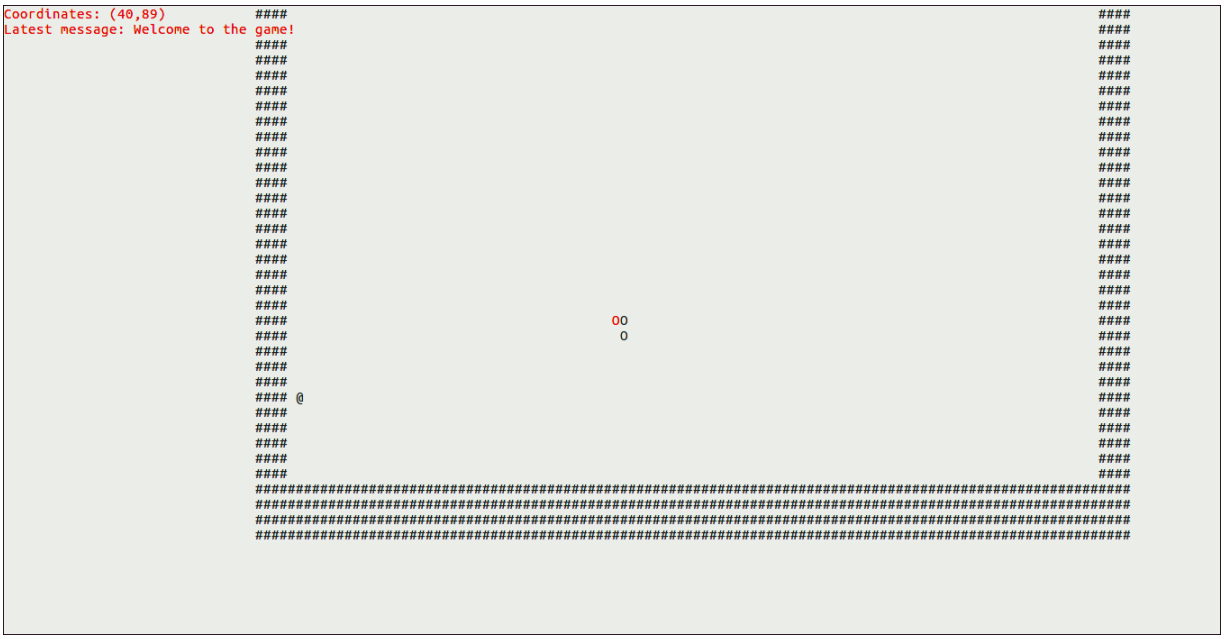

Figure 20 – Change in Direction

7.2 – Collisions

Figure 21 shows what happens when a snake collides with a wall. Note the snake's head inside the wall at the point of collision. Also note the message from the server at the top left corner.

Figure 21 – Collision with Wall

Figure 22 shows what happens when a snake collides with itself. Again, note that snake's head at the point of collision and the message from the server at the top left corner.

Figure 22 – Collision with Itself

Figure 23 shows two snakes about to collide with each other head to head.

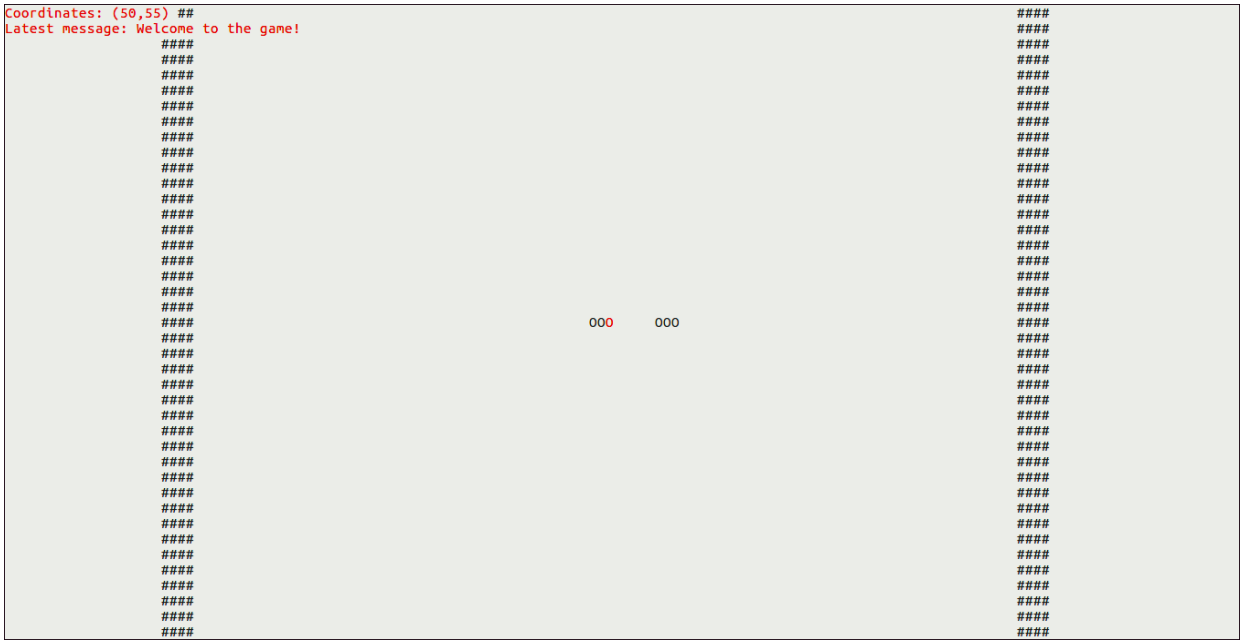

Figure 23 – Two Snakes about to Collide

Figure 24 shows what happens when two snakes collide with eachother head to head.

Figure 24 – Two Snakes Collide Head-to-Head

Figure 25 shows what happens when one snake collides with another snake's body. Note that the visible snake is not the snake that collided, but the snake with which the other snake collided.

Figure 25 – Snake Collides with Body of Other Snake

7.3 – Fruit-Eating, Winning, and Game Restart

Figure 26 shows a snake about to eat a fruit.

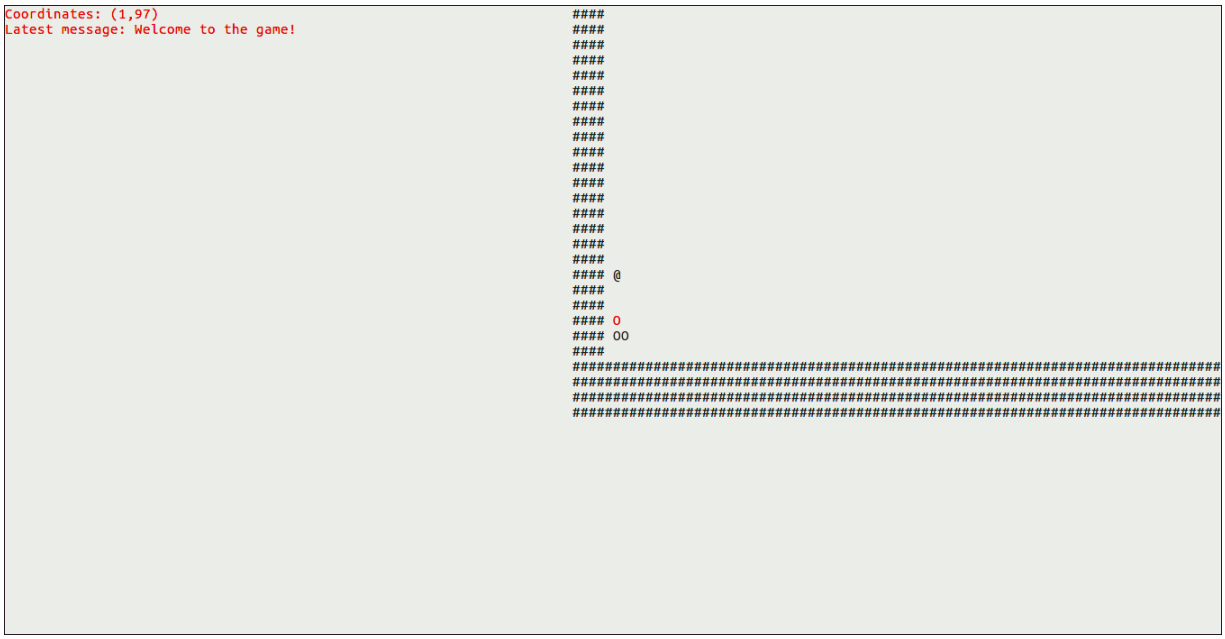

Figure 26 – About to Eat a Fruit

Figure 27 shows a snake that has just successfully eaten a fruit (note the message at the top left) and has grown by one snake part. Note also that another fruit has been created close to the snake.

Figure 27 – Fruit Eaten, Snake Grown, and New Fruit

Figure 28 shows a snake of length 14 about to eat a fruit and win.

Figure 28 – About to Win

Figure 29 shows a snake of length 15 that has just eaten the last fruit and won.

Figure 29 – Win

Figure 30 shows a game being restarted after a snake won. Note the counter at the top left which starts from 3 and gives the players a few seconds to prepare.

Figure 30 – Game Restart (countdown)

Figure 31 shows a game that has just started after a restart.

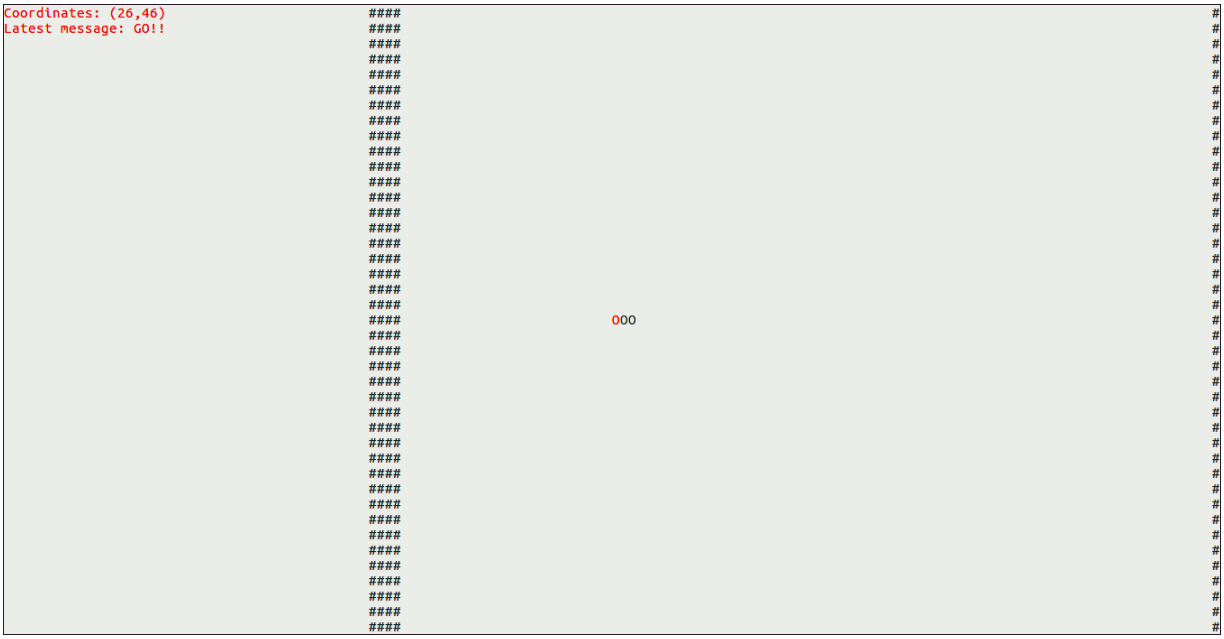

Figure 31 – Game Restart (started)

7.4 – CTRL+C and Kill Command

Listing 1 shows the text that appears in the terminal when the CTRL+C command or the Kill command are used on the server side.

```
Closing :: Set game to not running.
Closing :: Called free on snakes list.
Closing :: Called free on map.
Closing :: Destroyed snakes list lock.
Closing :: Closed client sockets.
Closing :: Called cose on server socket.
```

Listing 1 – CTRL+C or Kill Command on Server

Listing 2 shows the text that appears in the terminal when the CTRL+C command or the Kill command are used on the client side, or when the server closes down. The "Latest message received" depends on the latest message received from the server. Listing 3 shows what happens on the server side when a client disconnects. Note that the server keeps running after this text is shown. The snake number depends on the snake that disconnected.

```
Closing :: Stopped curses mode.
Closing :: Latest message received : Welcome to the game!
Closing :: Set game to not running.
Closing :: Called free on map.
Closing :: Destroyed map lock.
Closing :: Called close on socket.
```

Listing 2 – CTRL+C or Kill Command or Server Close on Client

```
Receiving of direction failed. Disabling direction changes for snake 4.
Client left the game (Reason: sending of update failed). New client count: 0.
```

Listing 3 – Client Loses Connection to Server

7.5 – Linked List Memory Leaks

A test file (LinkedListTest.c) was included in the *tests* folder to test the custom-made linked list structure and functions for memory leaks. Running *test1* from the compiled programs using *valgrind* produces the below results (Listing 4). Note how the *valgrind* results indicates that there were no memory leaks.

```
==19801== Memcheck, a memory error detector
==19801== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==19801== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==19801== Command: ./tests
==19801==
Snakes list size before remove: 4
Snakes list size after remove:  0
Snakes list size before remove: 4
Snakes list size after remove:  0
Snakes list size after last remove:  0
Changes list size before remove: 4
Changes list size after remove:  0
Changes list size before remove: 4
Changes list size after remove:  0
Changes list size after last remove:  0
==19801==
==19801== HEAP SUMMARY:
==19801==     in use at exit: 0 bytes in 0 blocks
==19801==   total heap usage: 103 allocs, 103 frees, 2,944 bytes allocated
==19801==
==19801== All heap blocks were freed -- no leaks are possible
==19801==
==19801== For counts of detected and suppressed errors, rerun with: -v
==19801== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Listing 4 – Linked List Memory Leaks

7.6 – Serialization and Deserialization Methods

A test file (SerAndDes.c) was included in the *tests* folder to test three serialization and three deserialization functions. Running *test2* from the compiled programs produces the below result (Listing 5).

```
Performing test 1...Success.
Performing test 2...Success.
Performing test 3...Success.
```

Listing 5 – Serialization and Deserialization Test Output