

Programming Project 01 - C programming

CS200 - Fall 2024-2025

Due Date: 11:55 pm - Sept 29, 2024

Lead TA(s): Namwar Ahmad Rauf (Part 01) & Eman Nabeel (Part 02)

Contents

1	Overview	2
1.1	Plagiarism policy	2
1.2	Submission instructions	2
1.3	Code Quality	2
1.4	Starter files	3
1.5	Grading distribution	4
1.6	Docker	4
1.7	Restrictions	4
2	Part 1: Abstracting contiguous memory	5
2.1	Stage 1	6
2.1.1	Part 1.1: Record keeping	7
2.1.2	Part 1.2: Contiguous memory abstraction	9
2.2	Stage 2	11
2.2.1	Part 1.3: Mapping for independent memory access	13
2.2.2	Part 1.4: Contiguous pointer	15
2.3	Testing	18
3	Part 2: CityBloxx	20
3.1	Overview	20
3.2	Introduction to CityBloxx	20
3.3	Skeleton Code	21
3.3.1	Struct Breakdown	22
3.3.2	Function Breakdown	22
3.4	Test Cases	27
3.5	Testing	28

1 Overview

1.1 Plagiarism policy

- Students must not share or show program code to other students.
- Copying code from the internet is strictly prohibited.
- Any external assistance, such as discussions, must be indicated at the time of submission.
- Course staff reserves the right to hold vivas at any point during or after the semester.
- All submissions will be subjected to plagiarism detection.
- Any act of plagiarism will be reported to the Disciplinary Committee.
- Some future assignments may allow using automated code generators such as ChatGPT. However, this programming assignment does NOT allow using such tools.

1.2 Submission instructions

- Zip the entire folder and use the following naming convention: `<rollnumber>_PA1.zip`
For example, if your roll number is 26100181, your zip file should be: `26100181_PA1.zip`
- All submissions must be uploaded on the respective LMS assignment tab before the deadline.
Any other forms of submission (email, dropbox, etc) will not be accepted.

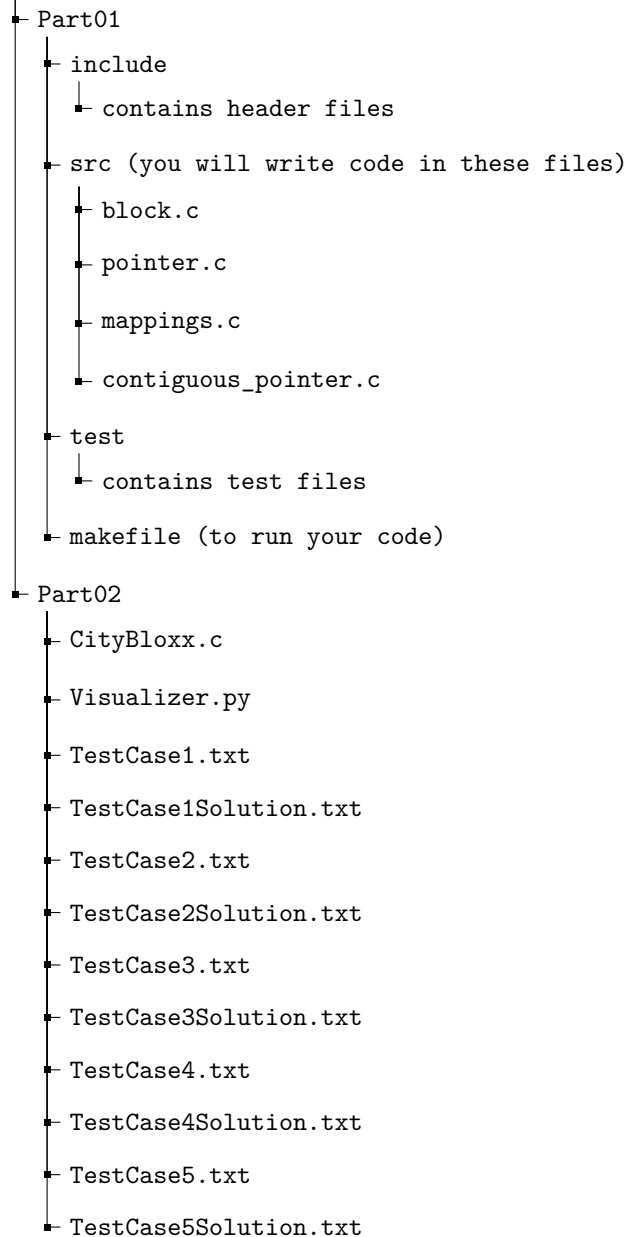
1.3 Code Quality

You are expected to follow good coding practices. We will manually look at your code and award points based on the following criteria:

- **Readability:** Use meaningful variable names as specified in the cpp coding standards and lectures. Assure consistent indentation. We will be looking at the following:
 - **Variables:** Use lowercase letters and separate words with under- scores. For example, `my_variable`
 - **Constants:** Use uppercase letters and separate words with underscores. For example, `MAX_SIZE`, `PI`
 - **Functions and Methods:** Use lower camelCase (capitalize the first letter of each word, except for the first word, without underscores). For example, `myFunction()`, `calculateSum()`
 - **Classes:** Use CamelCase (capitalize the first letter of each word without underscores). For example, `MyClass`, `CarModel`
- **Modularity and Reusability:** Break down your code into functions with single responsibilities.
- **Code Readability:** You should make sure that your code is readable. This means that your variables have meaningful names
- **Documentation** (Optional): Comment your code adequately to explain the logic and flow.
- **Error Handling:** Appropriately handle possible errors and edge cases.
- **Efficiency:** Write code that performs well and avoids unnecessary computations.

1.4 Starter files

PA1



1.5 Grading distribution

Part 1	
Part 1.1: Record Keeping	(11 points)
Part 1.2: Contiguous memory abstraction	(18 points)
Part 1.3: Mapping for independent access	(16 points)
Part 1.4: Contiguous pointer	(15 points)
Part 2	
Test Case 1	(3 points)
Test Case 2	(5 points)
Test Case 3	(7 points)
Test Case 4	(10 points)
Test Case 5	(10 points)
Hidden Test Case	(10 points)
Code Quality	(5 points)
Total	110 points

1.6 Docker

Please note that your assignments will be tested on Docker containers. As such, it is recommended that you run your code on it at least once before submitting it. Should any errors arise on our end due to incompatibility, you will be given the chance to contest your code.

1.7 Restrictions

Any violation of the below restrictions will result in a zero.

- You are not allowed to edit any structure provided.
- You are not allowed to declare your own structures.
- You are not allowed to edit function names, parameters, or return types.
- You are not allowed to declare helper functions unless specified.
- You are not allowed to declare any form of global or static variables unless specified.
- You are not allowed to edit the header files.
- You are not allowed to include any other libraries other than those already those provided in the starter code.
- You are not allowed to include any libraries that are not already provided.

2 Part 1: Abstracting contiguous memory

You have been hired as a memory management expert at Doofenshmirtz Programming Incorporated. Your boss, Dr. Doof, has tasked you with overseeing his latest invention, the Contiguous-inator, a powerful memory abstraction tool designed to access data in a flexible manner. He has (graciously) outlined a few core requirements for you:

1. It should be capable of storing various types of data.
2. The data must be capable of being accessed: i) Sequentially, where the n th data block must be accessed before the $n+1$ th block, or ii) Independently, where any data block can be accessed intentionally.

You must develop the Contiguous-inator before Dr. Doof's arch nemesis, pop star Katy Perry, hears news of this.

2.1 Stage 1

In the first stage of development, you aim to abstract contiguous memory across unique memory allocation calls. This means that, for a given program, it must appear that the n th allocation is adjacent to the $n-1$ th allocation. Typically, the memory allocator allocates requested memory wherever it finds appropriate space depending on memory availability and extent of fragmentation.

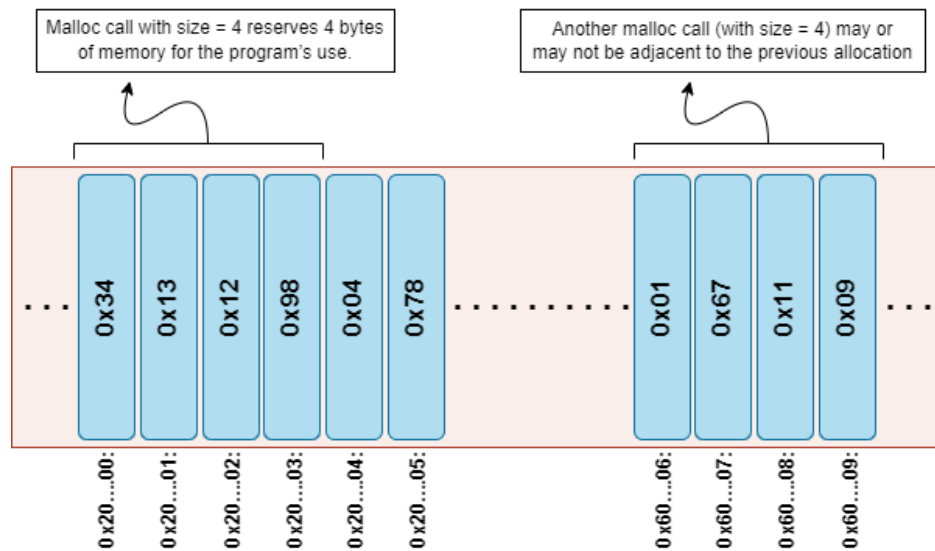


Figure 1: Independent malloc calls

Rather than accessing scattered memory, the intention is to make it appear as if multiple independent malloc calls are contiguously allocated, even if they are not actually contiguous inside the memory.

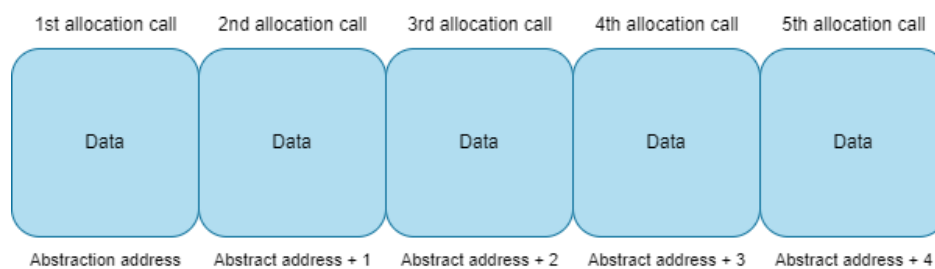


Figure 2: Abstracting independent allocations to appear contiguous

To do this, you will develop two libraries:

1. Layer 1: Record keeping – Part 1.1
2. Layer 2: Contiguous memory abstraction – Part 1.2

2.1.1 Part 1.1: Record keeping

This will be the first layer of your memory abstraction library. This layer has two main purposes:

1. Keep track of the location and size of allocated memory.
2. Store appropriately sized data inside the allocated memory.

The following structure is used to keep track of the required information regarding memory allocated at a given memory address.

```

1 typedef struct {
2     void* mem_space;
3     int bytes;
4 } Block;

```

A visual representation is shown, where a void pointer indicates the memory space where appropriate data is being stored.

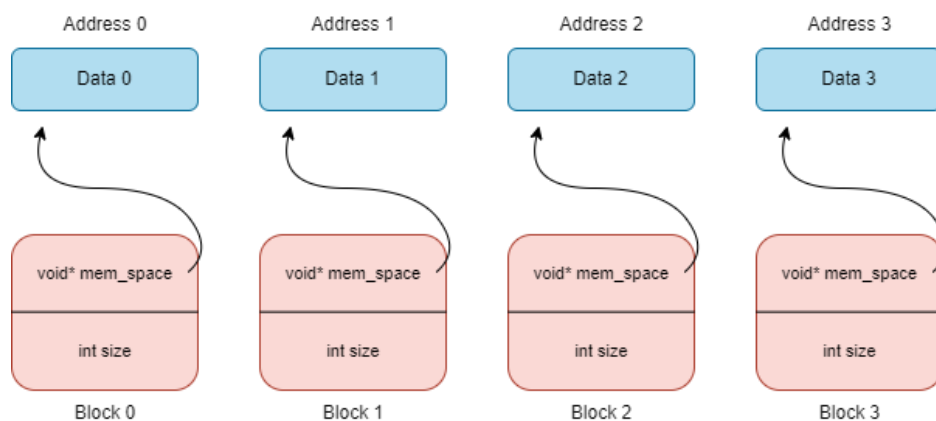


Figure 3: Storing metadata regarding the allocated memory space.

You have also been provided with an enumeration to allow type specification as input for memory allocation to this layer.

```

1 typedef enum {
2     INT_TYPE,
3     FLOAT_TYPE,
4     DOUBLE_TYPE,
5     CHAR_TYPE,
6 } DataType;

```

Library functions:

Implement the following functions, in 'block.c', as described.

For zero-sized allocations, return NULL.

- **Block* initializeBlock(DataType type, int units, void* data)**
The origin block must be created before any further memory allocation for data can be performed. Initialize Block 0, the origin block, as well as the memory space (size of the required memory space is indicated by the given type and number of units) for storing the specified data.
Note: you may assume that the size of the data to be stored will not be greater than the size (number of bytes) of the allocated memory.
- **Block* blockMalloc(DataType type, int units)**
Allocate a dynamic memory space of the specified size (bytes) and store its location in an instance of a Block structure. The allocated memory must be initialized to 0.
- **void blockFree(Block* block)**
Deallocate all the allocated memory related to the specified block.
- **void blockStoreData(Block* block, void* data)**
Store data in the specified block from the memory space indicated by the given pointer. You can assume that the pointer points to data of appropriate size with respect to the allocated memory space inside the block.
- **void blockAccessData(Block* block, void* dest, DataType type, int units)**
Access data from the specified block and store it in the memory space indicated by the given pointer. Note: The type and units represent the size of the destination memory space indicated by the pointer.
- **void blockRealloc(Block* block, DataType type, int new_units)**
Given a block that points to a specific memory space, you must reallocate the memory space to be of the new specified size while ensuring the data stored is not corrupted. You can assume that the DataType specified will be the same as before. However, number of units may differ.
For zero-sized reallocations, do nothing.
Hint: be wary of overwriting more memory than allocated, i.e. the new size may or may not be greater than the previous memory size.

2.1.2 Part 1.2: Contiguous memory abstraction

This is the second layer of the library that is built using layer one. This layer is meant to abstract memory to appear contiguous. It does so by keeping track of all the individual memory blocks created in layer 1 and connecting them together.

```

1 typedef struct BlockPointer{
2     Block* block;
3     DataType type;
4     int units;
5     int offset;
6     struct BlockPointer *next;
7 } BlockPointer;

```

This layer also keeps track of the type of data and the number of units of that datatype being allocated through a given block.

The offset is used to represent the block number relative to the origin block. (Note: This is a simplification; the true offset would be represented in bytes according to the space occupied by the blocks).

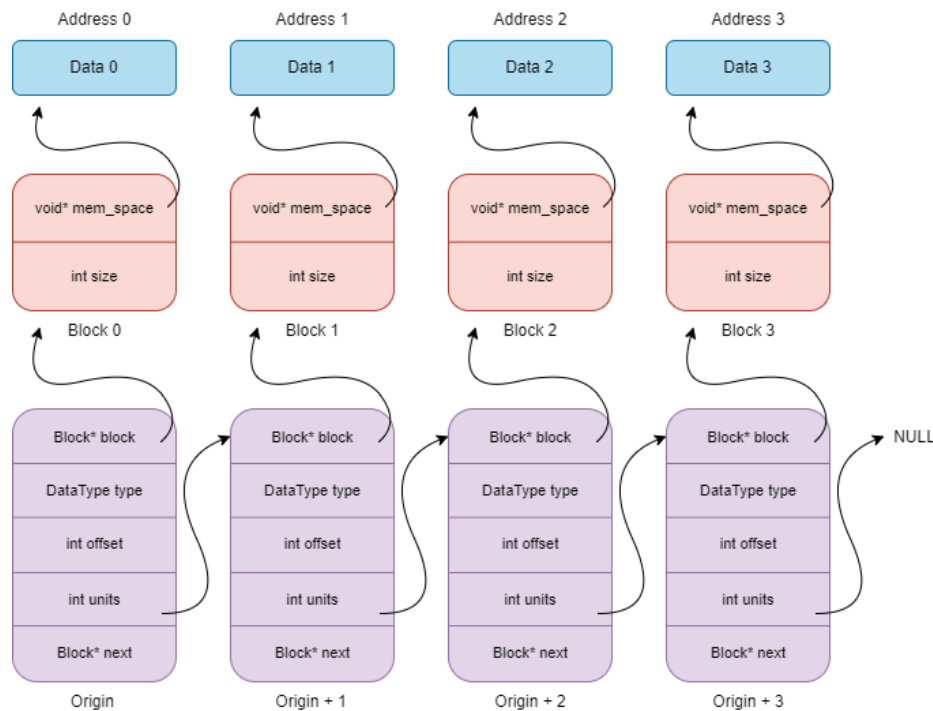


Figure 4: Linking the allocations together.

Library functions:

Implement the following functions, in 'pointer.c', as described.

For zero-sized allocations, return NULL.

- **BlockPointer* initializeBlockPointer(DataType type, int units, void* data)**
Initialize the block pointer that contains information regarding the origin block. You must allocate the origin block through this initialization and store its details accordingly.
Note: The offset of the origin block is 0.
- **BlockPointer* pointerMalloc(DataType type, int units, BlockPointer* origin)**
Create an instance of a block pointer that contains information regarding the block through which memory space of the specified size has been allocated.
Make sure to establish the linkage between this block pointer and the latest block pointer in the chain originating from the origin.
- **void pointerFree(BlockPointer* block_ptr, BlockPointer* origin)**
Deallocate all allocated memory related to the specified block pointer. Upon deallocation of the specified block pointer, connect its immediate predecessor and successor to each other.
- **void pointerStoreData(BlockPointer* block_ptr, void* data)**
Store the data in the memory location specified by the pointer to the allocated memory indicated by the block pointer.
- **void pointerAccessData(BlockPointer* block_ptr, void* dest)**
Access data through the specified block pointer and store it in the memory space indicated by the given pointer.
- **void pointerRealloc(BlockPointer* block_ptr, BlockPointer* origin, DataType type, int new_units)**
Reallocate the memory space holding the data indicated by the specified block pointer to the specified size. For zero-sized reallocations, do nothing.
Note: This does not ask you to reallocate the block pointer itself.
- **BlockPointer* getNext(BlockPointer* block_ptr)**
Given a block pointer, this function must return its immediate successor.
- **BlockPointer* getPrevious(BlockPointer* block_ptr, BlockPointer* origin)**
Given a block pointer, this function must return its immediate predecessor.
- **void pointerCompleteDeallocate(BlockPointer* origin)**
Given the origin block pointer, you must deallocate all allocated memory related to all block pointers allocated during a program's lifetime.

2.2 Stage 2

In this stage, you will develop the following two functionalities on top of the memory abstraction library.

- A mapping to allow direct access to independent memory blocks – Part 1.3
- A custom pointer type for sequential or independent access via querying the mappings library – Part 1.4

The custom pointer will allow traversing the entire allocated memory through “incrementing” and “decrementing” operations. This is equivalent to pointer arithmetic, except in this case the contiguous memory abstraction allows simply traversing the allocated memory, rather than performing actual arithmetic. (Note: actual arithmetic would be impossible in this abstraction as the intention is to hold any type of data in any allocated memory slot and the memory is not actually contiguous).

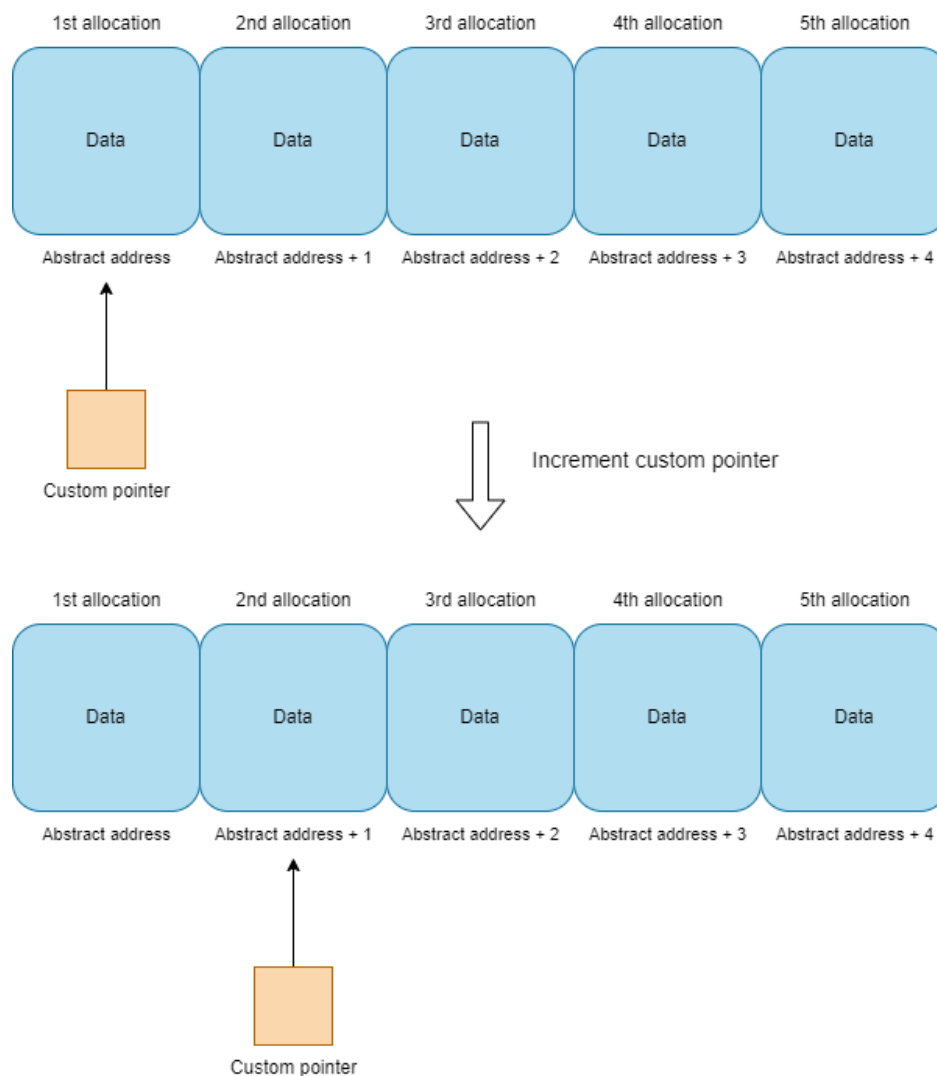


Figure 5: Sequentially accessing the data

However, due to the nature of abstraction, the custom pointer type created above cannot independently access specific known data, as the location of that data is unknown to the pointer. Hence, this requires creating a mapping that allows the storage addresses of known data to be looked up.

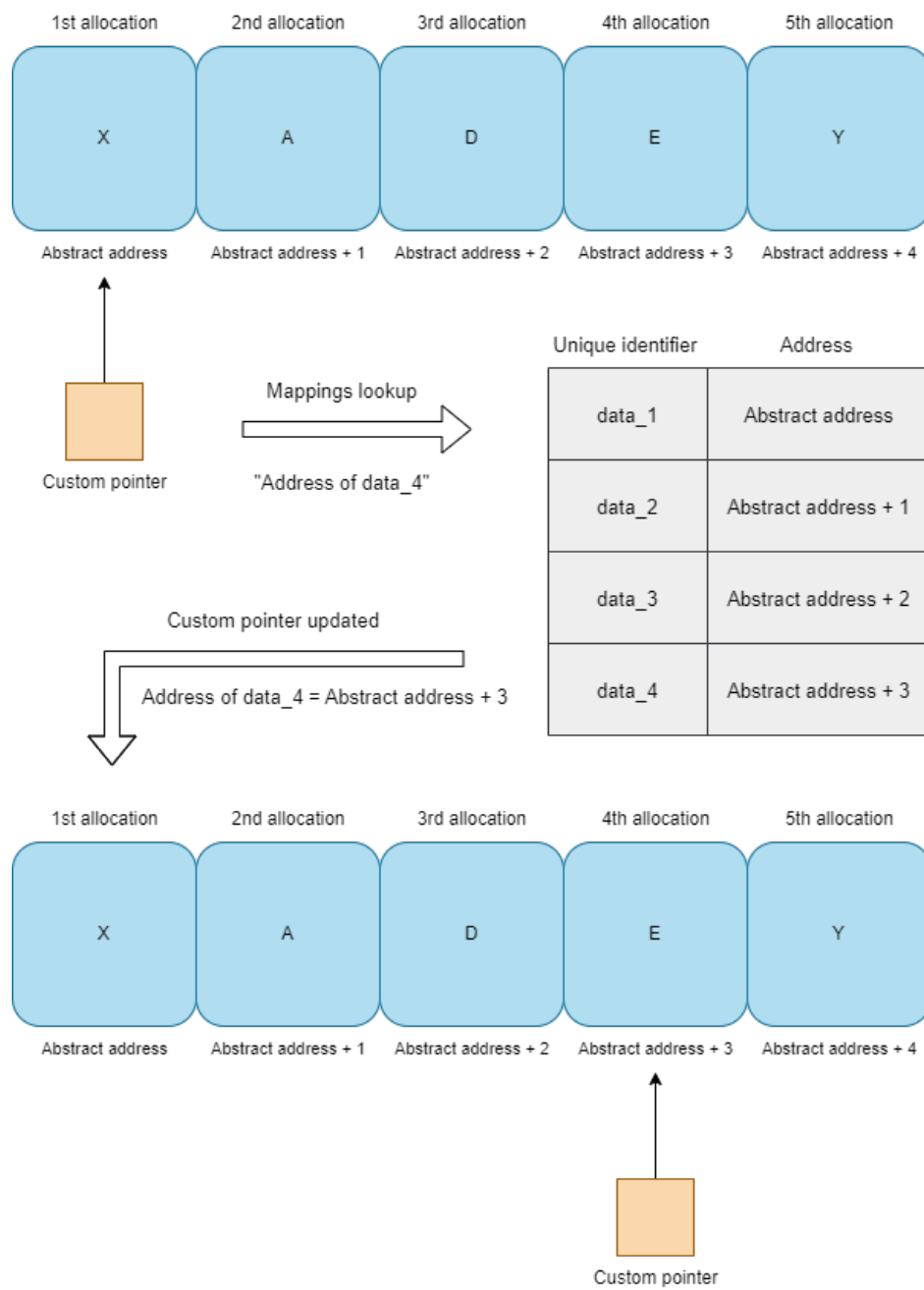


Figure 6: Independently accessing data through a mapping

2.2.1 Part 1.3: Mapping for independent memory access

The mappings program stores the memory addresses of all block pointers corresponding to the actual memory address that holds the data identified via a unique identifier.

```

1 typedef struct {
2     char* identifier;
3     uintptr_t address;
4 } Mappings;

```

The mappings library stores the address of the identifier's corresponding block pointer as an unsigned integer rather than a pointer.

Note: `uintptr_t` is a data type defined in the `stdint.h` library, capable of holding memory addresses in integer form. Moreover, the memory address can be explicitly type casted between a pointer type and an `uintptr_t` type (in either direction).

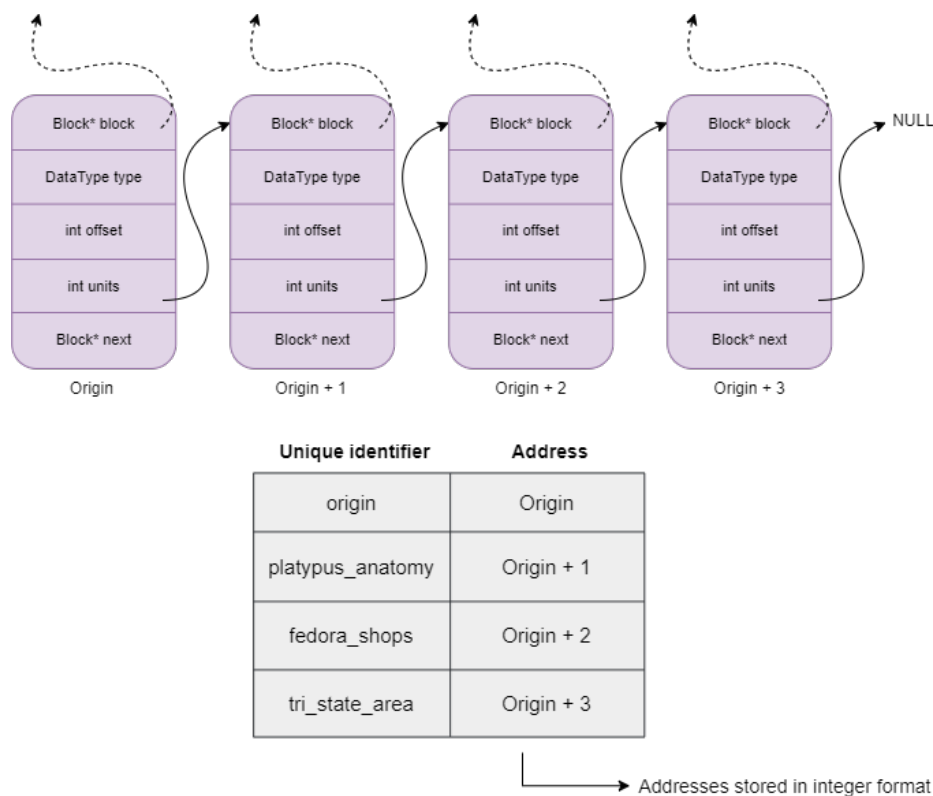


Figure 7: Mapping a unique identifier to a memory address

Refer to the example provided under part 1.4 for further clarity.

Library functions:

Implement the following functions, in 'mappings.c', as described.

Identifiers are passed as string literals (constants that exist throughout a program's lifetime). Hence, it is not compulsory to convert it to dynamic memory for this implementation.

- **Mappings* initializeMap(int num_allocations, void* addr, char* identifier)**
Initialize the map as an array capable of holding a num_allocations amount of mapping pairs excluding the origin, where num_allocations represent the maximum memory allocation calls we intend on making. Initialize the map to zero.
Note: The origin mapping pair is not included in the number of allocations being specified, as the origin is data kept internally to track the beginning of this contiguous memory abstraction. It is not "known" to any program that may be using these libraries.
- **void deallocateMap(Mappings* map)**
Deallocate all allocated memory space related to the map.
- **Mappings* resizeMap(Mappings* map, int new_num_allocs)**
To increase the number of allocations that can be made during a program's lifetime, resize a given map to a new size and copy all data present in the current map.
Free any allocated memory space that may no longer be in use.
Note: you may assume that the map will always be resized to a larger size.
- **void makeEntry(Mappings** map, void* addr, char* identifier)**
Make an entry in the given map of the given identifier-address pair at the next available position. If the map reaches its maximum capacity (number of allocations limit), resize it to double its current size.
You need not cater to cases of non-unique identifiers at this stage.
- **void removeEntry(Mappings* map, void* addr, char* identifier)**
Remove the identifier-address pair indicated by the specified identifier from the given map. The origin mapping pair can only be removed once there are no other entries remaining in the map.
- **void* getOrigin(Mappings* map)**
Return the address of the origin block pointer as a void pointer. If an origin has not been defined, return NULL.
- **void* getPointer(Mappings* map, char* identifier)**
Return the address corresponding to the specified identifier in a given map. If the identifier-address pair does not exist, return NULL.

2.2.2 Part 1.4: Contiguous pointer

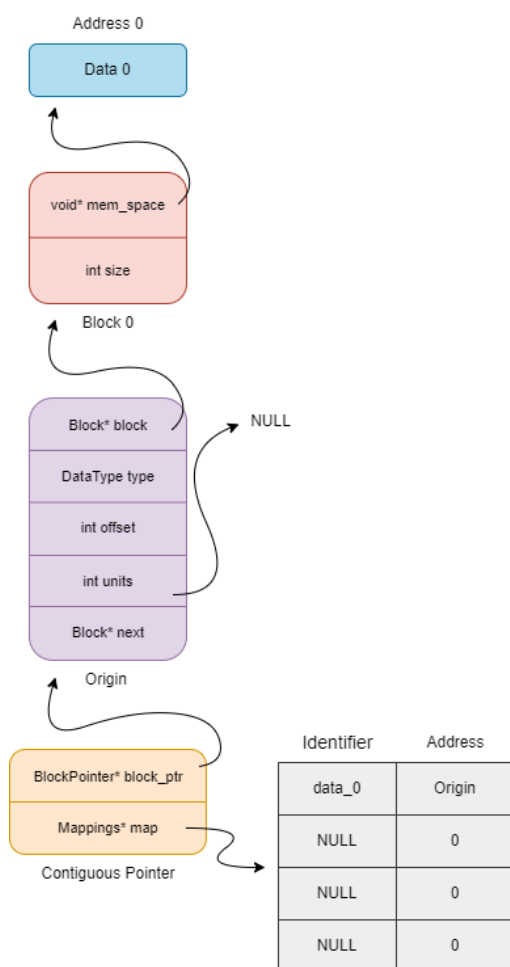
You have successfully created an abstraction for contiguous memory. Now, to allow for sequential memory access, a custom pointer type is needed that allows accessing the allocated memory directly or sequentially.

```

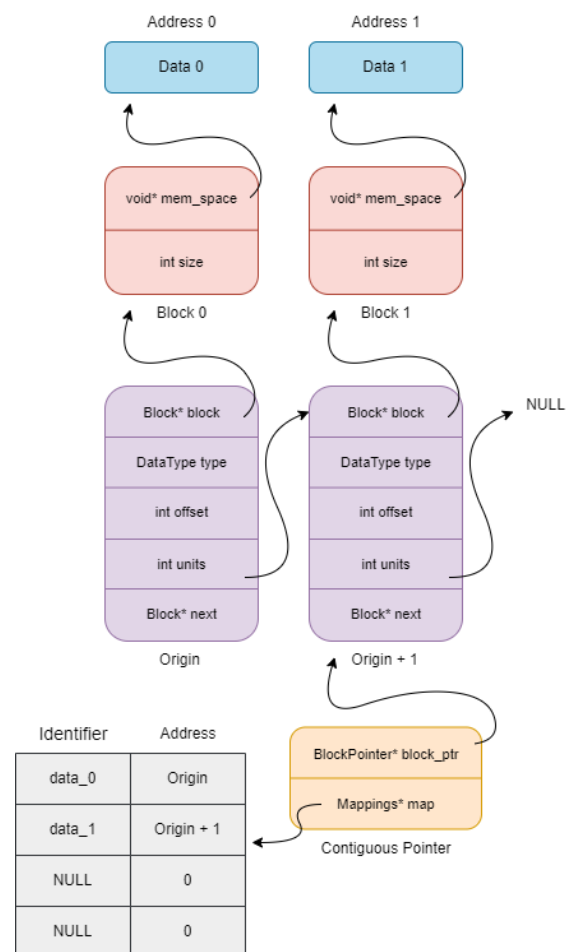
1 typedef struct {
2     BlockPointer* block_ptr;
3     Mappings* identifier_map;
4 } ContiguousPointer;

```

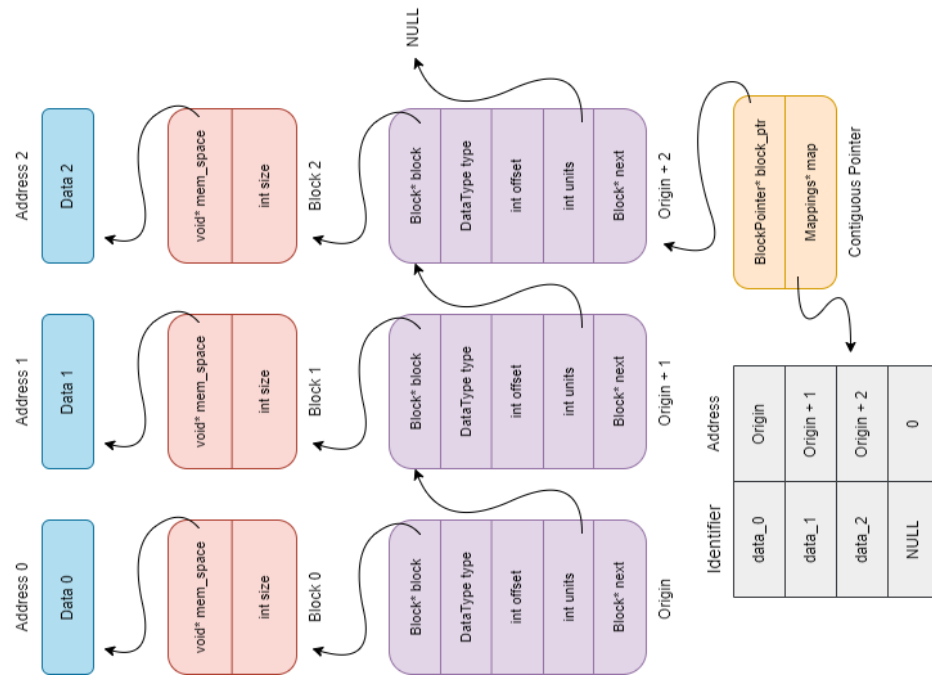
The example below assumes that we initially intend on performing a maximum amount of three memory allocations during a given program's lifetime



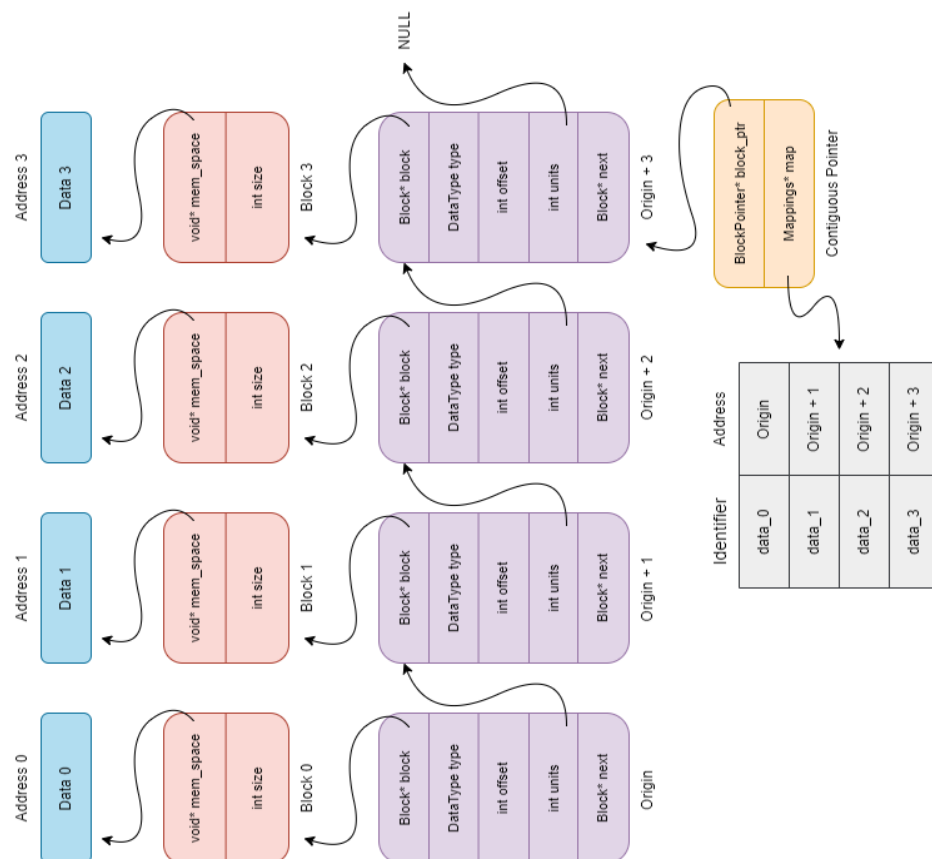
(a) The origin identifier-address pair is stored in the map on initialization.



(b) The map is updated upon another memory allocation.



(c) The contiguous pointer is updated as well upon a new entry.



(d) The third (and last) allocation is made and the map reaches its current capacity.

Library functions:

Implement the following functions, in 'contiguous_pointer.c', as described.

- **ContiguousPointer* initializeContiguous(int num_allocations)**
Initialize the contiguous pointer as well as the origin block. The identifier for the origin block will be "origin".
The contiguous pointer will not point to the origin after initialization. It should be NULL.
Hint: do not forget to account for the null terminator when storing a C-string in a memory space.
- **void contiguousMalloc(ContiguousPointer* c_ptr, DataType type, int units, char* identifier)**
Initialize a block pointer pointing to the specified size's memory space and store the corresponding mapping pair in the map.
The contiguous pointer must be updated accordingly to point to the block pointer corresponding to the latest allocation.
Make sure to check for non-unique identifiers.
- **void contiguousFree(ContiguousPointer* c_ptr, char* identifier)**
Free the memory space corresponding to the identifier and remove the entry from the map. Update the contiguous pointer to point to the first allocation.
Note: the first allocation does not refer to the origin. The origin should remain invisible to the external program using the contiguous pointer.
- **void storeData(ContiguousPointer* c_ptr, char* identifier, void* data)**
Store the data from the specified memory location to the allocated memory indicated by the identifier. Any attempts at overwriting the origin memory space should be prevented.
- **void accessData(ContiguousPointer* c_ptr, char* identifier, void* dest)**
Access data in the allocated memory corresponding to the identifier and store it in the destination memory space. Any attempts at accessing the origin memory space should be prevented.
- **void increaseAllocations(ContiguousPointer* c_ptr, int new_num_allocs)**
Increase the total number of possible allocation calls to the specified amount.
- **void incrementPointer(ContiguousPointer* c_ptr)**
The contiguous pointer should be "incremented" to now point to the immediate successor of the currently pointed-to memory space in the abstraction.
- **void decrementPointer(ContiguousPointer* c_ptr)**
The contiguous pointer should be "decremented" to now point to the immediate predecessor of the currently pointed-to memory space in the abstraction.
The origin must remain inaccessible.
- **void changePointer(ContiguousPointer* c_ptr, char* identifier)**
Update the contiguous pointer to now point to the memory space corresponding to the specified identifier.
The origin must remain inaccessible.
- **void completeDeallocation(Contiguous** c_ptr)**
Completely free all remaining allocated memory associated with the contiguous pointer and set it to NULL.

2.3 Testing

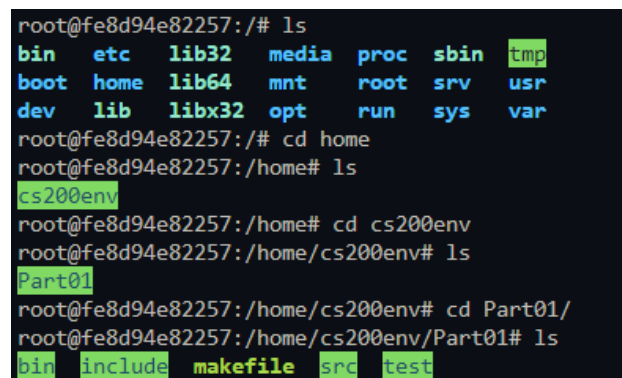
You have been provided with a Makefile for part 1, which handles the compilation targets of your program for you and allows selectively recompiling only those files where changes have been made.

The Make utility, which interprets the Makefile, is traditionally a Unix-based tool. Windows systems, by default, will not recognize the make command given below. It is highly recommended that you do not perform intermediate testing on Windows, even if you have installed the GNU Make tool. Instead, for Windows users, I would recommend installing Windows Subsystem for Linux (WSL) for intermediate convenient testing. However, make sure to test your program on docker atleast once before submitting as we will be exclusively testing on docker.

To test your program through the Makefile, first change your directory to the ../Part01 directory. To do this:

1. Use the ls command to view the files and subdirectories in your current directory.
2. Use the cd command followed by the directory name to change your current directory to the intended one.

Look at the figure below to get an idea on how to change your directory to the intended directory.



```
root@fe8d94e82257:/# ls
bin  etc  lib32  media  proc  sbin  tmp
boot home lib64  mnt    root  srv   usr
dev  lib  libx32 opt    run   sys   var
root@fe8d94e82257:/# cd home
root@fe8d94e82257:/home# ls
cs200env
root@fe8d94e82257:/home# cd cs200env
root@fe8d94e82257:/home/cs200env# ls
Part01
root@fe8d94e82257:/home/cs200env# cd Part01/
root@fe8d94e82257:/home/cs200env/Part01# ls
bin  include  makefile  src  test
```

Figure 9: Navigating directories to enter the intended one.

Once you have entered the ../Part01 directory, run the following commands.

To compile your program files:

```
make
```

Whenever you edit your program files, you need to recompile the files using the above make command. The Makefile will selectively recompile only those files that you edited.

To run the test cases for Part 1.1:

```
make run_block
```

To run the test cases for Part 1.2:

```
make run_pointer
```

To run the test cases for Part 1.3:

```
make run_mappings
```

To run the test cases for Part 1.4:

```
make run_contiguous
```

All object files (compiled program files), as well as Valgrind's text file outputs, will be created in the ../Part01/bin directory. To clear these files, simply run:

```
make clean
```

3 Part 2: CityBloxx

3.1 Overview

In this assignment, you'll delve into the world of dynamic 2D arrays by creating an engaging game called CityBloxx. This project must be completed individually, and you are required to use C programming.

3.2 Introduction to CityBloxx

In a bustling city, where buildings touch the sky, there was a unique challenge called "CityBloxx." Architects from all around were invited to design the tallest, most stable, and aesthetically pleasing apartment building using various blocks.

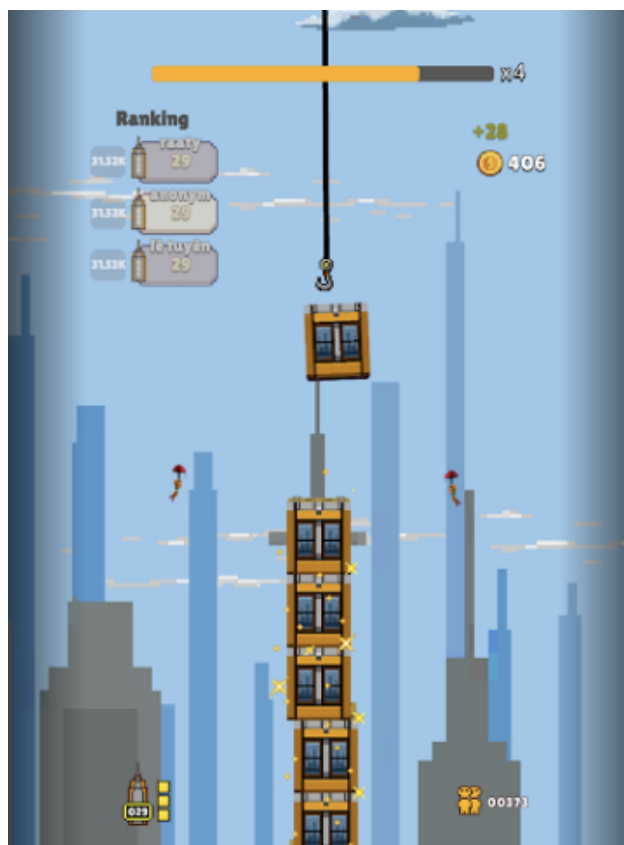


Figure 10: CityBloxx Game

Our version of CityBloxx is a unique adaptation of the classic game, designed to provide a more engaging and challenging experience. In this version, you'll take on the role of the programmer, building and managing the game's core mechanics. You'll be responsible for coding the game's logic, including handling user input, updating the game state, and displaying the results. This project will test your skills in C programming, as you'll need to implement a dynamic 2D array to represent the game grid, develop algorithms to manage block placement, and ensure the game's rules are correctly followed. By the end of this project, you'll have created a fully functional version of CityBloxx, showcasing your ability to apply programming concepts to create an interactive and enjoyable game.

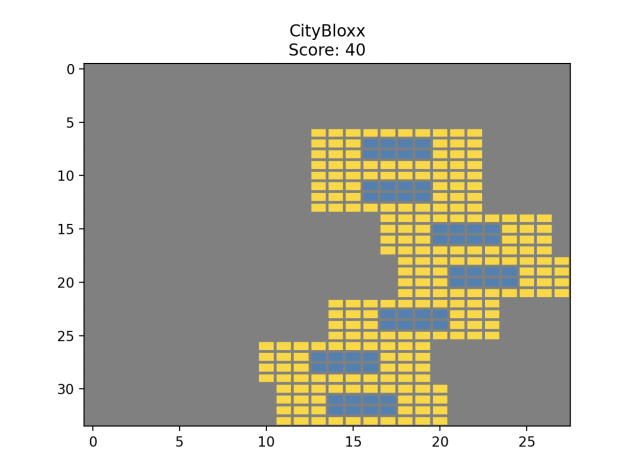


Figure 11: Our CityBloxx

3.3 Skeleton Code

You have been provided the following starter code. YOU ARE NOT TO EDIT THIS AT ALL.

```
typedef struct
{
    char **data; // Pointer to a pointer (2D array)
    int rows;    // Number of rows
    int cols;    // Number of columns
} Dynamic2DArray;

typedef struct
{
    int left_cord; // left coordinate of previous block
    int right_cord; // right coordinate of previous block
    int score;      // score of player
    int count; // variable to keep a check of perfect landings
    int height; // number of blocks in the tower
    int state; // variable to keep track of ending state of tower
    Dynamic2DArray *city; // pointer to 2D array to store the grid
} CityBloxx;
```

Figure 12: Skeleton Code

3.3.1 Struct Breakdown

1. **struct Dynamic2DArray**

As explained in figure 11, this basically denotes a 2D array simulation. This will act like your grid or city.

2. **struct CityBloxx**

As explained in figure 11, this will be the definition of your game attributes. Here is a breakdown of provided attributes:

- *int left_cord*: this will be used to store the left coordinate of your previous block or latest landed block.
- *int right_cord*: this will be used to store the right coordinate of your previous block or latest landed block.
- *int score*: this variable will be used to keep track of your score during the game.
- *int count*: this variable will be used to keep track of your perfect landings, i.e, when both coordinates of your new block are equal to coordinates of your previous block meaning that the 2 blocks are perfectly stacked onto each other. Please note that once a block is placed in a non perfect manner, count must reset to its initial state and must only keep incrementing in the case of continuous perfect landings.
- *int height*: this variable will be used to keep track of the height of your tower.
- *int state*: this variable will be used to keep track of the state of your tower. You may set this to any arbitrary number to denote the 2 possible states (either fallen or complete).

3.3.2 Function Breakdown

With each block placed, the building grew taller and more impressive. The architects cheered as they managed to balance creativity with precision, ensuring their apartment building stood tall and proud amidst the city's skyline.

In the end, the CityBloxx competition wasn't just about winning; it was about pushing the boundaries of what was possible and creating something beautiful that would stand the test of time.

And so, in this grand tale of architecture and engineering, every block told a story, and every story contributed to the magnificent saga of the CityBloxx apartments. Thus, we are to test your building.

1. **initialize_grid()**

```
void initialize_grid(Dynamic2DArray *city)
{
    int rows = 7;
    int cols = 70;
    // your implementation here
}
```

Figure 13: initialize_grid() function

Your grid initially needs to have 7 rows and 70 columns as shown in figure 13 meaning that you will declare a 7 by 70 grid. The function is pretty simple. In this function, you will be passed the city grid and you will allocate memory to it as mentioned above. You need to dynamically allocate memory. In addition to this, you must initialise the whole array with spaces since the testing will be done by comparing text files. Don't forget to set the rows and cols attribute of the city pointer itself.

2. simulate()

```
void simulate(const char *moves, CityBloxx *game)
{
    initialize_grid(game->city);
    // initialise all variables of game struct according to your implementation
    // open the file and start processing the moves iteratively. Each line has one move on the file.
    // check whether the move is valid or not. if yes, process the move.
    // else ignore the move and do relevant negative marking.
    // calculate the coordinates of incoming block according to your devised formula
    // check whether it is a valid block or not. if yes, land it and update variables accordingly.
    // else terminate the game
    // once block has landed, update the grid by simply extending the grid by rows upwards only
    // keep repeating this till the file ends or game terminates
}
```

Figure 14: simulate() function

The simulate function in your code is designed to process moves from a file and simulate the CityBloxx game accordingly. This function reads moves from a file, validates each move, updates the game grid accordingly, and manages game state and scoring. It's crucial to ensure the coordinates are correctly deduced based on the input format defined by the game rules and examples provided.

The presence of a landed block is denoted by the following characters filling the array accordingly. Each block takes up 4 rows and 10 columns.

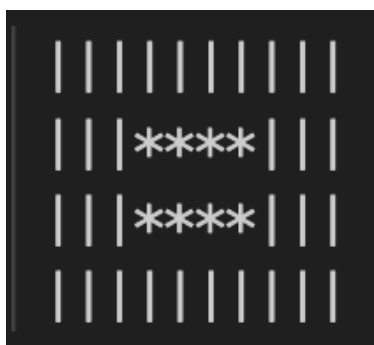


Figure 15: Block

Moreover, the following table consists of a few potential moves (moves are characters ranging from a-z and A-Z, one on each line of a text file) and their generated coordinates. You are to deduce a formula for generating these moves yourself. Be careful about edge cases and look at ascii values :)

Note: x_1 denotes the left coordinate of the incoming block and can be easily deduced by looking at the examples. x_2 , however, is simply $x_1 + 9$ since each block will take up space from column x_1 to x_2 with x_1 and x_2 included.

Move	Generated Coordinates
a	$x_1 = 48, x_2 = 57$
b	$x_1 = 49, x_2 = 58$
X	$x_1 = 39, x_2 = 48$
f	$x_1 = 21, x_2 = 30$
r	$x_1 = 16, x_2 = 25$
A	$x_1 = 16, x_2 = 25$

Table 1: Moves and Coordinates

Moreover, it is highly important to note that a policy passed by LDA does not allow you to build a tower less than 20 units from the road. Thus, your habitable area is only 50 UNITS.

Invalid Block: Your first block must always land as long as its within the range of the grid. For subsequent blocks, you must check whether stacking it would be a perfect stack or not. If yes, you must increment count and update the score by multiplying count with 10 and adding it to the current score. If not, you must set count = 0 and should update the score with a +5 award. However, if an incoming block is invalid, your game must terminate with landing that block anywhere. The formula to determine whether a block is valid or not is:

$$F = \frac{|count - height \times |diff||}{height}$$

diff is the difference between the left coordinate of previous block and next block or the difference between the right coordinate of previous block and next block. You must check both cases.

F must be **less than 7** for the block to land.

Processing Moves: The moves are essentially characters, one on each line of the text file being passed in the function "moves". You must read the file iteratively and simulate the game. Here is an example of what the input file looks like.

```

1  m
2  l
3  p
4  t
5  s
6  o
7  o|

```

Figure 16: Input File

Invalid Move: You must check whether your move is within the range of the defined set of moves. If not, you must ignore that move and move forward. You must also apply a -5 penalty on the player's current score.

Perfect Stackings: Once you calculate that the incoming block will be landing perfectly over the previous block, you must increment count and update score by multiplying count to 10 and adding to old score.

IMPORTANT:

In case, your game goes on perfectly, your state must be set to complete. If not, your state must be set to fallen. You may use any arbitrary integers to denote these states according to your implementation. After processing each move, you must extend your array upwards by 4 rows and fill empty spots by spaces. You must ensure efficient dynamic allocation of new array and deletion of the old one. It is recommended that you make a helper function for this purpose.

3. display()

This function is pretty simple and involves printing the contents of your grid onto the text file. The file is named *solution.txt* and is already included in the Assignment folder. Moreover, the part to be careful about is printing the state and score correctly. Here is an example of what your output must look like.

```

1  What a tower!   Score: 105
2
3
4
5
6
7
8  |             |
9  |             |
10 |             |
11 |             |
12 |             |
13 |             |
14 |             |
15 |             |
16 |             |
17 |             |
18 |             |
19 |             |
20 |             |
21 |             |
22 |             |
23 |             |
24 |             |
25 |             |
26 |             |
27 |             |

```

Figure 17: Output File

You must have noticed the tower state at the top. Hence, you must print **"What a tower!"** if your player successfully stacks all blocks and game ends successfully. However, if the game terminates due to invalid blocks, you must print **"Tower fallen!"**. Then, in the same line, exactly **5 spaces** away, you must print the score in the exact format displayed in figure 17.

```

1 Tower fallen! Score: 55
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23

```

Figure 18: Output File

Note: You must ensure that everything is printed in the correct format, including all empty spots denoted by spaces and all occupied spots correctly displayed by the relevant format. Otherwise your test cases won't pass, given that testing is being done by comparing text files.

If in any case, you're stuck, you can open the solution files (5 files in total named as TestCasexSolution where x is the relevant test case number) and compare your result manually.

4. Visualizer

In case, you're struggling in interpreting any miscalculation in your coordinates, you can run the visualizer provided. You are to run the following command in terminal before running the visualizer.

```

pip install matplotlib
pip install numpy

```

Figure 19: Installing Libraries

```

emannabeel@192 Assignment_1 % python3 Visualizer.py

```

Figure 20: Running the visualizer

Upon running the visualizer, you will see your tower like this.

The x-axis denotes the coordinates (column number of start and end) of your blocks and the y-axis denotes the $\text{height} * 4 + 7$.

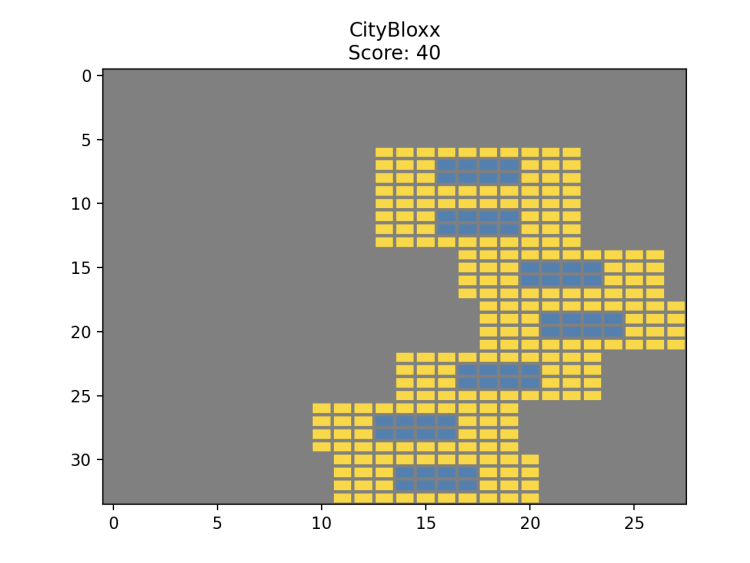


Figure 21: Visualizer

Moreover, in order to change the text file to visualize, you need to replace the following file name with your desired filename. The python file is also included in your assignment folder.

```

52
53 if __name__ == "__main__":
54     filename = "TestCase2Solution.txt" # Change this to the output file you want to visualize
55     blocks, score = parse_file(filename)
56     visualize(blocks, score)

```

Figure 22: Visualizer Code

Your solution is printed in the *solution.txt* file and the test case solution is in the *TestCaseSolution.txt* files. You may comment out other test cases in the main function to check or dry run any specific test case so that your solution.txt file only displays the output for that specific test case. Testing, however, on our end will be carried out by the default main function.

IMPORTANT: You must note that testing will be done automatically, and no partial marks will be awarded. You will be given the marks the test case shows, and no exceptions will be made. You also must not alter any code except the 2 functions you have to implement.

3.4 Test Cases

1. Test Case 1: (3 Marks)

This is a basic test case only checking whether your code correctly processes perfectly stacked blocks and updates the score accordingly. The game terminates successfully.

2. Test Case 2: (5 Marks)

This is also a basic test case only checking whether your code correctly processes imperfectly stacked blocks and updates the score accordingly. The game terminates successfully.

3. Test Case 3: (7 Marks)

This test case is an extension of the previous 2 test cases such that it checks whether your code processes both types of moves correctly and updates the score accordingly. The test case also checks whether your code ignores invalid moves and carries out negative marking or not. The game terminates successfully.

4. Test Case 4: (10 Marks)

This test case is an extension of test case 4 such that this time the game does not terminate successfully.

5. Test Case 5: (10 Marks)

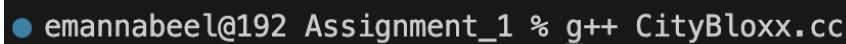
This test case is a collective case testing all possible moves a user may come up with.

6. Hidden Test Case: (10 Marks)

This test case is similar to all 5 of the test cases given to you ;)

3.5 Testing

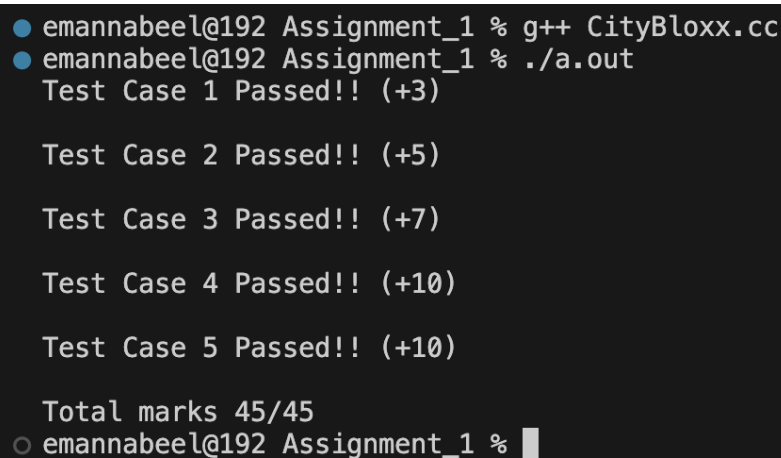
Testing is quite simple and you only need to run the following command after ensuring that you are in the correct directory.



```
● emannabeel@192 Assignment_1 % g++ CityBloxx.cc
```

Figure 23: Testing

Then, you need to run the following command.



```
● emannabeel@192 Assignment_1 % g++ CityBloxx.cc
● emannabeel@192 Assignment_1 % ./a.out
Test Case 1 Passed!! (+3)

Test Case 2 Passed!! (+5)

Test Case 3 Passed!! (+7)

Test Case 4 Passed!! (+10)

Test Case 5 Passed!! (+10)

Total marks 45/45
○ emannabeel@192 Assignment_1 %
```

Figure 24: Testing (My hidden test case passes)

The terminal will show detail of which test case isn't passing or if all are passing (if you're very smart ;)). In case, you don't understand why a particular test case isn't passing or want to manually inspect the solution.txt file for that test case, you may temporarily comment out all other test cases and observe only that particular case.

IMPORTANT: All required libraries and files have already been included for you. All of your must

be on CityBloxx.cc and any other code must not be altered. Marking will be strictly done on the basis of test cases and no partial marks for incomplete code will be given.

No need to worry!! You got this :)