# Array Assessment

Arrays are a foundation concept to learn and understand, broadly speaking they are used to store a collection of values for a given data type. Defining an array is easy but using arrays in various contexts requires us to create algorithms and define logic to achieve a desired outcome.

For this assessment task, you are going to complete a collection of exercises that uses arrays in a variety of different ways.

Download the Visual Studio Solution: **IntroToCPP\_Arrays.**  
This solution contains the below required and optional exercises.

# Exercise Checklist

|  |  |
| --- | --- |
| Required Exercises | DONE |
| Exercise 1: Looping   * Task 1: Print values in order * Task 2: Print values in reverse order | [YES / NO]  [YES / NO] |
| Exercise 2: Passing Arrays to Functions | [YES / NO] |
| Exercise 3: Binary Search | [YES / NO] |
| Exercise 4: Sort Array | [YES / NO] |
| Exercise 5: 2D Array Indexing   * Task 1 * Task 2 * Task 3 | [YES / NO]  [YES / NO]  [YES / NO] |
| Exercise 6: Simple Tile Map   * Task 1: Initialise array * Task 2: Draw Tile Map * Task 3: Modify Tile Map on Mouse Click | [YES / NO]  [YES / NO]  [YES / NO] |

These Optional Exercises aim to provide you with some small and simple real world examples of using arrays. At our current level of experience, these exercises are quite advanced, but should still be achievable within a few hours for each task.

If you find your unable to get started or are completely lost with these exercises, chances are you require more time learning the basics of C++ syntax, breaking down problems and debugging.

|  |  |
| --- | --- |
| Optional Exercises | DONE |
| Conway’s Game of Life   * Task 1: Calculate Tile State | [YES / NO] |
| Othello   * Task 1a: Is Valid Move (console) * Task 1b: Is Valid Move (raylib) * Task 2: Make Move (raylib) | [YES / NO]  [YES / NO]  [YES / NO] |
| Scrabble   * Task 1: Can Place Word * Task 2: Calculate Word Score | [YES / NO]  [YES / NO] |

Contents

[Array Assessment 1](#_Toc87008535)

[Exercise Checklist 1](#_Toc87008536)

[Exercise 1: Looping 4](#_Toc87008537)

[Task 1: Print each value of the array in order. 4](#_Toc87008538)

[Task 2: Print each value of the array in reverse order. 4](#_Toc87008539)

[Exercise 1: Passing Arrays to Functions 5](#_Toc87008540)

[Exercise 3: Binary Search 6](#_Toc87008541)

[Exercise 4: Sort Array 7](#_Toc87008542)

[Exercise 5: 2D Array Indexing 8](#_Toc87008543)

[Task1: 8](#_Toc87008544)

[Task 2: 9](#_Toc87008545)

[Task 3: 9](#_Toc87008546)

[Exercise 6: Simple Tile map 10](#_Toc87008547)

[Task 1: 10](#_Toc87008548)

[Task 2: 11](#_Toc87008549)

[Task 3 12](#_Toc87008550)

[Optional Exercise: Conway’s Game of Life 13](#_Toc87008551)

[Task 1: Calculate Tile State 14](#_Toc87008552)

[Optional Exercise: Othello 15](#_Toc87008553)

[Task 1a: Is Valid Move 16](#_Toc87008554)

[Task 1b: Is Valid Move - Raylib 17](#_Toc87008555)

[Task 2: Make Move - Raylib 18](#_Toc87008556)

[Optional Exercise: Scrabble Scoring 19](#_Toc87008557)

[Task 1: Can Place Text 20](#_Toc87008558)

[Task 2: Calculate Score 21](#_Toc87008559)

# Exercise 1: Looping

Open the Project **AIE\_01** and **AIE\_01.cpp**You will see we have defined an array for you

|  |
| --- |
| const int NUM\_NUMBERS = 5;  int numbers[NUM\_NUMBERS] = { 10, 20, 30, 40, 50 }; |

Task 1: Print each value of the array in order.

**Expected console Output:**

|  |
| --- |
| 10, 20, 30, 40, 50, |

## Task 2: Print each value of the array in reverse order.

**Expected Output**

|  |
| --- |
| 50, 40, 30, 20, 10, |

# Exercise 2: Passing Arrays to Functions

Open the Project **AIE\_02** and **AIE\_02.cpp**You will see we have defined the following functions

|  |
| --- |
| int MinValue(const int\* arr, int count);  int MaxValue(const int\* arr, int count);  int CountOccurencesOfValue(const int\* arr, int count, int search); |

An array in C++ is simply a pointer to the first value within an array, when we pass an array into a function, we also need to pass the number of values the array contains. So, the above methods also include a count parameter.

These methods should not modify the passed in array, therefore we have defined the array as const so the compiler will prevent us from modifying the array we pass into the method.

We have defined an array in the main method.

|  |
| --- |
| const int NUM\_NUMBERS = 10;  int numbers[NUM\_NUMBERS] = { 10, 20, 30, 40, 50, 10, 20, 30, 40, 50 }; |

In **AIE\_02.cpp**, You will see that we have invoked the above methods and expect these methods to return an expected value.

Your task is to implement the methods, so that the expected values are correctly returned. When you have implemented these methods, you should see the following output:

|  |
| --- |
| Expected: 10 - Got: 10  Expected: 50 - Got: 50  Expected: 2 - Got: 2  Expected: 0 - Got: 0 |

# Exercise 3: Binary Search

Open the Project **AIE\_03** and **AIE\_03.cpp**You will see we have defined an array that is sorted in ascending order.

|  |
| --- |
| const int NUM\_ITEMS = 8;  int arr1[NUM\_ITEMS] = {1, 3, 5, 7, 9, 11, 13, 15}; |

We have also defined a “BinarySearch” function that you are required to implement.

|  |
| --- |
| const int\* BinarySearch(const int\* arr, int count, int searchVal); |

This method should:

* Return a pointer the found value if it exists, or
* Return nullptr if the value does not exist.

We have provided a few tests to verify if your implementation works as expected, below we are searching for the values 9, 1, 15, 0 and 16 for the array defined above.

|  |
| --- |
| TestResult( BinarySearch(arr1, NUM\_ITEMS, 9), &arr1[4] );  TestResult( BinarySearch(arr1, NUM\_ITEMS, 1), &arr1[0] );  TestResult( BinarySearch(arr1, NUM\_ITEMS, 15), &arr1[7] );  TestResult( BinarySearch(arr1, NUM\_ITEMS, 0), nullptr );  TestResult( BinarySearch(arr1, NUM\_ITEMS, 16), nullptr ); |

Expected Output:

|  |
| --- |
| Pass: result(9)  Pass: result(1)  Pass: result(15)  Pass: result(nullptr)  Pass: result(nullptr) |

# Exercise 4: Sort Array

Open the Project **AIE\_04** and **AIE\_04.cpp**You will see we have defined an array that is not sorted.

|  |
| --- |
| const int NUM\_ITEMS = 8;  int numbers[NUM\_ITEMS] = { 13, 1, 7, 5, 11, 9, 15, 3 }; |

You will see we have defined a SortArray function that you are to implement.

|  |
| --- |
| void SortArray(int\* arr, int count); |

This method should modify the passed in array, so that by the time the method returns, the array has been re-ordered so that it is sorted in ascending order.

# Exercise 5: 2D Array Indexing

Open the Project **AIE\_05** and **AIE\_05.cpp**You will see we have defined a 2D array of strings, here we have defined 3 rows and 5 columns.

|  |
| --- |
| const int ROWS = 3;  const int COLS = 5;  const char\* grid2D[ROWS][COLS] =  {  { "a0", "a1", "a2", "a3", "a4" },  { "b0", "b1", "b2", "b3", "b4" },  { "c0", "c1", "c2", "c3", "c4" },  }; |

We have also defined a second 1D array with the same values.

|  |
| --- |
| const char\* grid1D[ROWS \* COLS] =  {  "a0", "a1", "a2", "a3", "a4",  "b0", "b1", "b2", "b3", "b4",  "c0", "c1", "c2", "c3", "c4"  }; |

For this exercise, we have 3 tasks for you to complete. These will require you to loop through grid2D.

## Task1:

Print all values in grid2D. You will need 2 loops to iterate over the rows and columns.  
  
**Expected output:**

|  |
| --- |
| a0, a1, a2, a3, a4,  b0, b1, b2, b3, b4,  c0, c1, c2, c3, c4, |

## Task 2:

Unlike task1, we have defined a single loop. You are required to convert the loop index value into a row and column index value. To achieve this you must know how many columns are defined for the array. The Formula is:

col Index = index modulus num columns  
 row index = index divided by num columns

**Expected output:**

|  |
| --- |
| a0, a1, a2, a3, a4, b0, b1, b2, b3, b4, c0, c1, c2, c3, c4, |

## Task 3:

For Task 3, we have defined a single 1D array called grid1D. we have also provided a nested loop that is used to iterate over rows and columns. Our task is to convert the row and column index into a single index.

Formula:

Index = rowIndex \* NUM\_COLS + colIndex

**Expected output:**

|  |
| --- |
| a0, a1, a2, a3, a4,  b0, b1, b2, b3, b4,  c0, c1, c2, c3, c4, |

# Exercise 6: Simple Tile map

Open the Project **AIE\_06**, **Application.h** and **Application.cpp.**This project uses raylib to render a tile map to the screen, we have define a 1D array that will be used in a 2D context.

You’ll see we have defined the following member variables in Application.h

|  |
| --- |
| static const int ROWS = 40;  static const int COLS = 40;  int m\_tiles[ROWS \* COLS];  int m\_tileWidth = 20;  int m\_tileHeight = 20; |

## Task 1:

Initialise all values in the m\_tiles array to a random value between 0 and 5 exclusive – these values will represent a colour for the tile.

* 0 = WHITE
* 1 = RED
* 2 = GREEN
* 3 = BLUE
* 4 = YELLOW
* Any other value = BLACK

Open **Application.cpp** navigate to the **Load()** method.  
Iterate over all values in the array and assign them a random value between 0 and 5 exclusives.

|  |
| --- |
| void Application::Load()  {  // Task1:  // Initialise all values in m\_tiles array to a random  // value between 0 and 5 exclusive;  // -----------------------------------------------------    // write your code here  // -----------------------------------------------------  } |

## Task 2:

We are going to use raylib to draw a coloured rectangle to the screen, here you will need to use a nested loop to iterate over rows and columns.

Open **Application.cpp** navigate to the **Draw()**

* Use the row and column value to calculate the position.
  + xPos = colId \* tileWidth
  + yPos = rowId \* tileHeight
* Use the tile value to find the colour required to render the tile.  
  To get the tile value – you will need to convert the row and col to an array index using the following formula:
  + index = rowId \* COLS + colId

Once you finish task 2, you should be able to run the project. You should see a window with random colours rendered.

|  |
| --- |
| void Application::Draw()  {  BeginDrawing();  ClearBackground(RAYWHITE);  // Task2:  // use a nested loop to iterate over rows and columns  // Use raylib's DrawRect method to draw each tile in the array.  // use the row and col index multipled by m\_tileHeight/m\_tileWidth  // to calculate the x and y position for each rectangle.  //  // change the color of the rect drawn based on the value of the tile.  // We have created a helper function you can use "GetTileColor"  // --------------------------------------------------------------------  // write your code here  float xPos = 0;  float yPos = 0;  Color color = GetTileColor(1); // pass in the tilevalue  DrawRectangle(xPos, yPos, m\_tileWidth, m\_tileHeight, color);  // --------------------------------------------------------------------  EndDrawing();  } |

Qr code

Description automatically generated

## Task 3

Now that we have rendered our tiles to the screen, we will modify values within the array when we click with the mouse.

* We need to convert the mouse position to a row / column. We can do this by dividing the mouse coordinates by the tileWidth and tileHeight.
* Once we have the row and column, we can calculate the array index through the following formula: index = rowId \* COLS + colId

Open **Application.cpp** navigate to the **Update()** method.  
Change the code so that we calculate the rowIndex, colIndex and tileIndex values.

|  |
| --- |
| void Application::Update(float deltaTime)  {  if (IsMouseButtonPressed(MOUSE\_LEFT\_BUTTON))  {  Vector2 mousePos = GetMousePosition();  // Task 3:  // TODO: Calculate row and col index based on the mouse positon  int rowIndex = 0;  int colIndex = 0;  // TODO: calculate the index of the tile clicked on based on the row/col index  int tileIndex = 0;  m\_tiles[tileIndex] += 1;  if (m\_tiles[tileIndex] >= 5)  m\_tiles[tileIndex] = 0;  }  } |

When complete, you should be able click a tile in the window, and have it cycle through to the next colour.

# Optional Exercise: Conway’s Game of Life

This project is the Conway’s Game of Life simulation. We have defined a dynamic array of integers representing a 2d tile set.

Each tile can have a value of either 0 or 1.

* 0 = The tile is dead
* 1 = The tile is alive

Here are some links to learn more about Conway’s game of life.

* <https://en.wikipedia.org/wiki/Conway%27s_Game_of_Life>
* <https://www.youtube.com/watch?v=R9Plq-D1gEk>

Your task is to implement the rules for to calculate if a given tile should be “Alive” or “dead” for the next generation.

Open **Application.cpp** navigate to the **Load** method.  
Here we are allocating 2 arrays of the same size: m\_grid and m\_gridBuffer.

|  |
| --- |
| // allocate memory for the grid  m\_grid = new int[m\_rows \* m\_cols];  m\_gridBuffer = new int[m\_rows \* m\_cols];  // initialise cells to a random value (0 or 1)  for (int i = 0; i < m\_rows \* m\_cols; i++)  {  m\_grid[i] = rand() % 100 < 10;  m\_gridBuffer[i] = m\_grid[i];  } |

Open **Application.cpp** navigate to the **Update** method.  
Here we calculate the new state for each tile in the grid. Notice we call the CalculateTileState method for each tile.

|  |
| --- |
| // copy "grid" to "gridBuffer"  memcpy(m\_gridBuffer, m\_grid, sizeof(int) \* m\_rows \* m\_cols);  // update the grid state for this frame  for (int i = 0; i < m\_rows \* m\_cols; i++) {  m\_grid[i] = CalculateTileState(i, m\_gridBuffer);  } |

## Task 1: Calculate Tile State

Open **Application.cpp** navigate to the **CalculateTileState** method  
This method is called every frame, for each tile (see above)

Currently, this method returns the current state of the tile, your task is to implement the Conway’s game of life rules, and return if the tile should be “alive” or “dead”

|  |
| --- |
| int Application::CalculateTileState(int index, int\* grid)  {  // Calculate the row/col index  int col = index % m\_cols;  int row = index / m\_cols;  bool isAlive = grid[index] != 0;   // Task:  // implement the rules for Conway’s game of life, the method should  // update the isAlive value based on the rules defined here:  // <https://en.wikipedia.org/wiki/Conway%27s_Game_of_Life>  return isAlive;  **}** |

By the time you’ve finished, you should be able to run the project and observe the Game of Life!

Chart, scatter chart

Description automatically generated

# Optional Exercise: Othello

Othello is a classic board game that can be played on an 8x8 board.

We have provided 2 projects:

1. OthelloConsole  
   This project isolates the exercise for Task 1 and provides some simple unit tests for us to validate our solution.
2. OthelloRaylib  
   This project adds raylib rendering and input, so that we can plug in our exercise solution and have a fully playable game of Othello.

Othello Rules:

* <https://www.youtube.com/watch?v=jlll7wfEKaI>
* <https://www.youtube.com/watch?v=lO2pEK33SSw&t=33s>

For this exercise, we will implement the logic for the following methods:

* Task 1: IsValidMove  
  This method returns true/false if the ‘player’ can place a token on the specified row/col on the grid.
* Task 2: MakeMove  
  This method returns a new OthelloBoardState where the ‘player’ token has been placed on the board and the appropriate tiles have been flipped.

|  |
| --- |
| bool IsValidMove(char player, int col, int row, const OthelloBoardState& grid);  OthelloBoardState MakeMove(char player, int col, int row, OthelloBoardState grid); |

## Task 1a: Is Valid Move

Open the Project **OthelloConsole** and **OthelloConsole.cpp**You will see we have defined an initial OthelloBoardState as follows

|  |
| --- |
| OthelloBoardState state1 =  {// 0 1 2 3 4 5 6 7  {' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '}, // 0  {' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '}, // 1  {' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '}, // 2  {' ', ' ', ' ', 'W', 'B', ' ', ' ', ' '}, // 3  {' ', ' ', ' ', 'B', 'W', ' ', ' ', ' '}, // 4  {' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '}, // 5  {' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '}, // 6  {' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '} // 7  }; |

We have also defined some basic unit tests for the Is Valid Move method.  
Below is an example calling the IsValidMove method it should return true/false if the ‘B’ token can be placed on the 5th column and 4th row. In this case, the method should return true as it is a valid position to place the token.

|  |
| --- |
| bool canPlaceToken = IsValidMove('B', 5, 4); |

We have written a few unit tests to verify the correct behaviour for this method.

|  |
| --- |
| ShouldBe(IsValidMove('B', 5, 4, state1), true); // true  ShouldBe(IsValidMove('W', 5, 5, state1), false); // false |

Your task is complete the implementation for this method.

|  |
| --- |
| bool IsValidMove(char player, int col, int row, const OthelloBoardState& grid)  {  // Task 1:  // this method should return true if the 'player'  // can be placed on the grid at the specified row/col location.  return false;  } |

When you run the program, these unit tests will run. By successfully implementing the IsValidMove method, all these unit tests should display as [PASS]

**Expected Output**

Shape, rectangle

Description automatically generated

## Task 1b: Is Valid Move - Raylib

Required:  
Complete Task 1A before progressing with Task 1B

Open the Project **OthelloRaylib** and **Application.cpp** then navigate to the **IsValidMove** method

Copy your solution from the previous exercise.  
When complete, you should be able to run the project.

The IsValidMove method is called for each tile on the board, and a marker is displayed on each valid tile where the current player can place their token.

Expected Output:

Chart

Description automatically generated

## Task 2: Make Move - Raylib

Complete Task 1A and Task 1B before progressing with this Task.

Open the Project **OthelloRaylib** and **Application.cpp** then navigate to the **MakeMove** method.

You are required to implement the logic to place the ‘player’ token and flip the tiles on the board.  
The method returns a new OthelloBoardState with the token placed, and appropriate tiles flipped.

The MakeMove method is called when the user clickes on a valid tile.  
The below code will place the token, but the flipping logic has not been implemented.

|  |
| --- |
| OthelloBoardState Application::MakeMove(char player, int col, int row, OthelloBoardState grid)  {  // place the player token  grid[row][col] = player;  // Task 2:  // Implement the make move method  // The method should return an altered OthelloBoardState  // modify the grid so the approprate tiles are flipped.  return grid;  } |

**Expected Output:**After this method has been implemented, you should have a fully playable “Othello” game.

# Optional Exercise: Scrabble Scoring

Scrabble is a word game in which two to four players score points by placing tiles, each bearing a single letter, onto a game board divided into a 15×15 grid of squares. The tiles must form words that, in crossword fashion, read left to right in rows or downward in columns, and be included in a standard dictionary or lexicon.

This project contains 2 tasks for us to implement.

* Task 1: CanPlaceText  
  This method return true/false if a given word can be placed on the board.
* Task 2: CalculateScore  
  This method returns a number representing the score a player would achieve when placing the word on a the scrabble board. Each letter should follow the point allocation of a traditional scrabble board, and apply score modifiers (Double Letter, Triple Letter, Double Word and Triple Word) based on where the word is placed in the grid.

Open the Project **ScrabbleScoring**, **ScrabbleBoard.h** and **ScrabbleBoard.cpp**You will notice we have defined some helper methods and some Enums, when implementing your solutions for this exercise, you should use these helper methods. In particular, the following methods will be helpful for you with implementing your solutions:

* GetBoardCharacter(xIndex, yIndex)  
  This method returns a ‘char’ for the given xIndex / yIndex index on the scrabble board.
* GetBoardModifier(xIndex, yIndex)  
  This method returns the EScoreMod enum for the given xIndex / yindex

## Task 1: Can Place Text

Open the Project **ScrabbleScoring** and **ScrabbleBoard.**cpp then navigate to the **CanPlaceText** method

|  |
| --- |
| bool ScrabbleBoard::CanPlaceText(int xIndex, int yIndex, const char\* text, EDirection dir)  {  // ========================================================================  // Task 1:  // xIndex and yIndex represent the location of the first character for the "text"  // being placed on the board.  //  // This method should return true if the supplied "text" can be placed  // on the board.  // - return false if the text runs out of bounds  // - return false if the text overlaps other characters (unless they are same)  // - return true if the text can be successfully placed on the board.  //  // use the method "GetBoardCharacter" to get the character on the board.  //  // ========================================================================  return true;  } |

**Expected Output:**When you run the project, the placed text turns red when it cannot be placed on the board.  
The below screenshots is one example where the text cannot be placed because all of the characters ‘hello’ cannot fit on the board.

Chart, treemap chart

Description automatically generated

## Task 2: Calculate Score

Open the Project **ScrabbleScoring** and **ScrabbleBoard.cpp** then navigate to the **CalculateScore** method

|  |
| --- |
| unsigned int ScrabbleBoard::CalculateScore(int xIndex, int yIndex, const char\* text, EDirection dir)  {  // ========================================================================  // Task 2  /\*  The method GetBoardModifier will tell you if there are any score  modifier values on the board at a given location.  Modifiers are:  NO: no score modification  DL: Double Letter Score  TL: Tripple Letter Score  DW: Double Word Score  TW: Tribble word score.  When calculating the score, letter modifiers apply first to calculate  the word score. Word modifiers are calculated after the total word score.  Letter scores:  (1 point) -A, E, I, O, U, L, N, S, T, R  (2 points)-D, G  (3 points)-B, C, M, P  (4 points)-F, H, V, W, Y  (5 points)-K  (8 points)-J, X  (10 points)-Q, Z  \*/  auto characterToPlace = text[0];  auto characterOnBoard = GetBoardCharacter(xIndex, yIndex);  auto modifier = GetBoardModifier(xIndex, yIndex);  int score = 1;  return score;  } |

**Expected Output:** In the below screenshot, a score of 20 has been returned for the word “Hello”.  
This word has been placed over 2 score modifiers.

* DW (DoubleWord): on the ‘h’ character
* TL (Triple Letter) on the ‘o’ character.

**The final score is calculated as follows:**

* H - 4 points
* E - 1 point
* L - 1 point
* L - 1 point
* O - 1 point \* 3 (triple letter)
* Word score: 10 points
* Total Score: 10 points \* 2 (double word score) = 20

A picture containing graphical user interface

Description automatically generated