ENGGEN 131 – Semester Two – 2019

C Programming Project

The Warehouse

Deadline: 11:59pm, Sunday 27th October
Worth: 12% of your final grade

A note before we begin…

Welcome to the C Programming project for ENGGEN131 2019!

This project is organized around a series of tasks. For each task there is a problem description, and you must implement *one function* to solve that problem. You may, of course, define *other functions* which these required functions call upon (i.e. these are called *helper* functions).

Do your very best, but don't worry if you cannot complete every task. You will get credit for each task that you do solve (and you may get partial credit for tasks solved partially).

IMPORTANT - Read carefully

This project is an assessment for the ENGGEN131 course. It is an **individual** project. You do not need to complete all of the tasks, but the tasks you do complete should be an accurate reflection of your capability. You may discuss ideas in general with other students, but writing code must be done by yourself. *No exceptions.* You must not give any other student a copy of your code in any form – and you must not receive code from any other student in any form. There are absolutely NO EXCEPTIONS to this rule.

Please follow this advice while working on the project – the penalties for plagiarism (which include your name being recorded on the misconduct register for the duration of your degree, and/or a period of suspension from Engineering) are simply not worth the risk.

Acceptable	***Unacceptable***
• Describing problems you are having to someone else, either in person or on Piazza, without revealing *any* code you have written • Asking for advice on how to solve a problem, where the advice received is general in nature and does not include *any* code • Discussing with a friend, away from a computer, ideas or general approaches for the algorithms that you plan to implement (but *not* working on the code together) • Drawing diagrams that are illustrative of the approach you are planning to take to solve a particular problem (but *not* writing source code with someone else)	• Working at a computer with another student • Writing code on paper or at a computer, and sharing that code in any way with anyone else • Giving or receiving any amount of code from anyone else in any form • Code sharing = NO

The rules are simple - write the code yourself!

OK, now, on with the project…

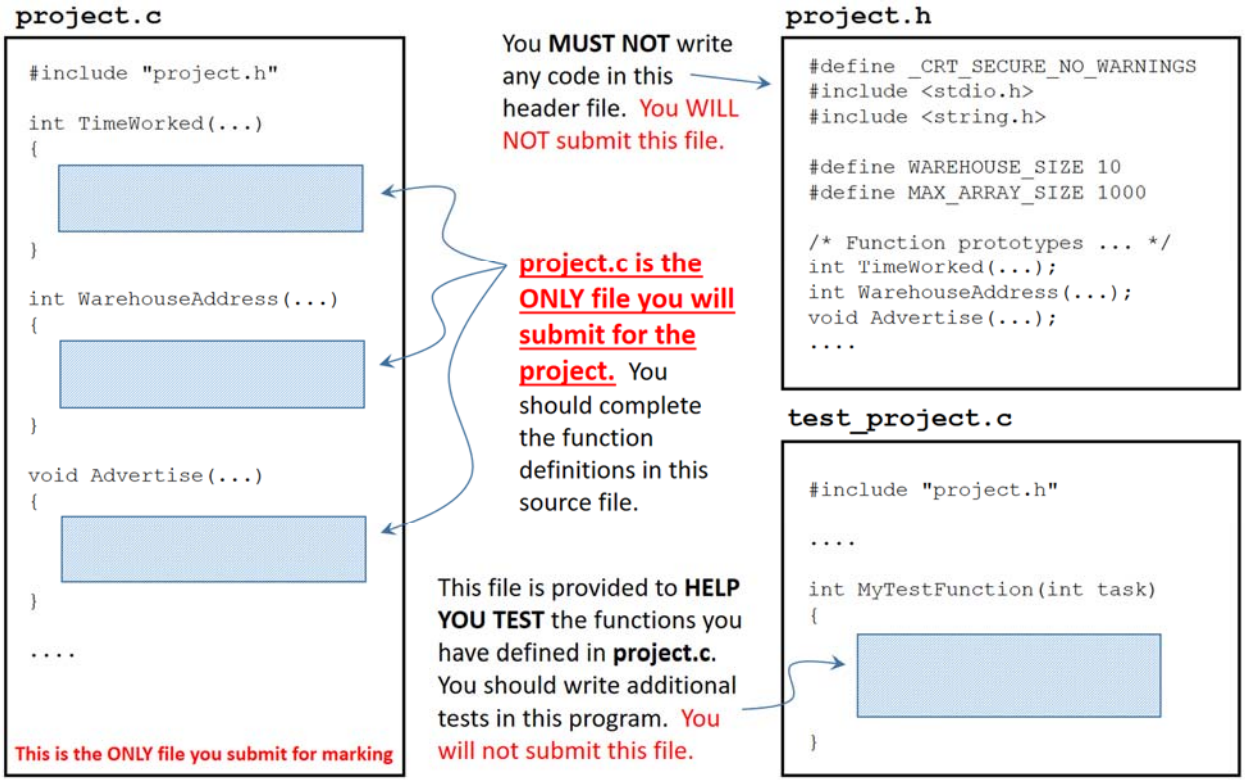
Understanding the project files

There are *three files* that you will be working with: **project.c**, **project.h** and **test_project.c**.

The most important of these three files is the source file **project.c**. This is the **ONLY** file that **you will submit for marking**. Please note the following:

- **project.c** is a source file that **ONLY CONTAINS FUNCTION DEFINITIONS**
- there is no **main()** function defined in **project.c** (and you **must not** add one)
- a separate program, **test_project.c**, contains a **main()** function and you can use this to help you test the function definitions that you have written in **project.c**

The diagram below illustrates the relationship between the three files.



The blue shaded regions in the above diagram indicate where you should write code when you are working on the project. There are three simple rules to keep in mind:

- You MUST NOT write any code in **project.h** (the header file)
- You MUST write implementations for the functions defined in **project.c (and this is the file you will submit for marking)**
- You SHOULD write additional test code in **test_project.c** to help you thoroughly test the code you write in **project.c**

Getting started

To begin, download the file called **CProjectResources.zip** from Canvas. There are three files in this archive:

project.c	This is the source file that you will ultimately submit for marking. In this source file you will find the functions that you should complete. Initially each function contains an *incorrect* implementation which you should correct. You may add other functions to this source file as you need. **You must not place a main() function in this source file**. This is the only file that you will submit for marking.
project.h	This is the header file that contains the prototype declarations for the functions you have to write. You must not edit this header file in any way. Both source files (**project.c** and **test_project.c**) include this header file, and the automated marking program will use the provided definitions in **project.h**. Modifying this header file in any way will cause an error.
test_project.c	This is the source file that contains the **main()** function. This file has been provided to you to help you test the functions that you write in **project.c**. In this file, you should create some example inputs and then call the functions that you have defined inside the **project.c** source file. Some simple examples have been included in this file to show you how this can be done.

You might like to start by looking at the **project.c** source file. In this source file you will find a series of function definitions, however initially they are all implemented *incorrectly*. The prototype declarations are as follows (and these declarations are defined in the **project.h** header file):

```
int TimeWorked(int minuteA, int secondA, int minuteB, int secondB);
int WarehouseAddress(int maximum);
void Advertise(char *words);
int WinningBid(int *values, int length);
void BoxDesign(char *design, int width, int height);
void WorkerRoute(int warehouse[10][10]);
int MakeMove(int warehouse[10][10], char move);
```

You need to modify and correct the definitions of these functions in **project.c**.

You will submit the file **project.c** for marking.

Run the test program
You should run the program provided to you in **test_project.c**. To do this, start by placing the three files (**project.c**, **project.h** and **test_project.c**) into an empty folder. You will need to compile both source files. For example, from the Visual Studio Developer Command Prompt, you could type:

```
cl /W4 project.c test_project.c
```

Or, simply:

```
cl /W4 *.c
```

You should see no warning messages generated when the code compiles.

If you run the program, you should see the following output:

```
ENGGEN131 2019 - C Project...
 _____ _             _    _                _
|_   _| |           | |  | |              | |
  | | | |__   ___   | |  | | __ _ _ __ ___| |__   ___  _   _ ___  ___
  | | | '_ \ / _ \ | |/\| |/ _` | '__/ _ \ '_ \ / _ \| | | / __|/ _ \
  | | | | | |  __/ \  /\  / (_| | | |  __/ | | | (_) | |_| \__ \  __/
  \_/ |_| |_|\___|  \/  \/ \__,_|_|  \___|_| |_|\___/ \__,_|___/\___|

Enter the number of the task that you would like to test:

  1 - 6  = this will execute the code in MyTestFunction()
  7 - 10 = this will play the Warehouse game
```

Now, simply enter the number of the task that you are working on that you would like to test. If you enter a number between 1 and 6, then the function **MyTestFunction()** which is at the top of the **test_project.c** source file will be called. As it is provided to you, this function includes some very simple tests for the first 6 tasks. These tests match the example code in this handout. You can compare the output generated by your functions with the expected output as listed in this handout. Please note, these are just a starting point for you - passing these provided tests does not mean that your solutions are entirely correct! You should design and add additional tests of your own.

If you enter a number between 7 and 10 then the Warehouse game will be launched. You can play this game to test the functionality of your MakeMove() function (which is the function that you will be working on for Tasks 7 – 10. Depending on which task you enter (between 7 and 10), the configuration of the Warehouse game will vary (i.e. the locations of the workers, boxes and targets). Feel free to edit these warehouse configurations yourself – you will find the 2D arrays near the bottom of the **test_project.c** source file (they are called *warehouse7*, *warehouse8*, *warehouse9* and *warehouse10*).

What to submit

You **must not** modify **project.h**, although you can modify **test_project.c**. You will not be submitting either of these files.

You must only **submit ONE source file** – **project.c** – for this project. This source file will be marked by a separate automated marking program which will call your functions with many different inputs and check that they produce the correct outputs.

Testing

Part of the challenge of this project is to **test your functions carefully** with a range of different inputs. It is very important that your functions will never cause the marking program to crash or freeze regardless of the input. If the marking program halts, you cannot earn any marks for the corresponding tasks. There are three common scenarios that will cause the program to crash and which you must avoid:

- Dividing by zero
- Accessing memory that you shouldn't (such as invalid array indices)
- Infinite loops

Using functions from the standard library

The **project.h** header file already includes <stdio.h> and <string.h>. You may use functions that are defined in these two libraries, but you may not use any other functions from the standard library. If you want some functionality, you must code it!

Marking

The correctness of your project will be marked by a program that calls the functions you have defined in **project.c** with many different input values. This program will check that your function definitions return the expected outputs for a large variety of possible inputs. Your mark for the project will essentially be the total number of these tests that are successful, across all of the tasks.

Some tasks are harder than others. If you are unable to complete a task, that is fine – just complete the tasks that you are able to. However, please do not delete any of the functions from the **project.c** source file. If you choose not to implement any given function, just leave the initial code in the function definition. All of the required functions must be present in the **project.c** file you submit for marking (so that the automated marking program is able to compile your code).

Never crash

One thing that you must pay important attention to is that your functions must never cause the marking program to crash. If they do, your will forfeit the marks for that task. This is your responsibility to check. There are three common situations that you must avoid:

- Never divide by zero
- Never access any memory location that you shouldn't (such as an invalid array access)
- Never have an infinite loop that causes the program to halt

You must guard against these **very carefully** – regardless of the input values that are passed to your functions. Think very carefully about every array access that you make. In particular, a common error is forgetting to initialise a variable (in which case it will store a "garbage" value), and then using that variable to access a particular index of an array. You cannot be sure what the "garbage" value will be, and it may cause the program to crash.

Array allocation

If you need to declare an array in any of your function definitions, and you are not sure what size to make it, you can use either of the following two constants from **project.h** (use the first one if you want to declare a 2D array for the Warehouse game, and use the second one if you want to declare a 1D array for any of the other functions in this project):

```
#define WAREHOUSE_SIZE 10
#define MAX_ARRAY_SIZE 1000
```

You must use a constant value when declaring an array (like the constants above). Do not use a *variable* to declare the size of an array - this will not compile on MSVC and you will get 0 for the project. This is your responsibility to check. **Please see page 38 of the coursebook**.

Comments

Every function you define in the **project.c** source file, which is the file you submit for marking, should begin with a short comment (two or three sentences) that describes the purpose of the function and the algorithm you have used to solve the task. This comment should be written in your own words. These functions should appear immediately above each function definition – for example:

```
/*
Your comment should go here – it should describe the purpose of the function
and a brief summary of the algorithm you have used to solve the task (this
comment must be written in your own words
*/
int TimeWorked(int minuteA, int secondA, int minuteB, int secondB)
{
      ....
```

Other than these opening comments, which must appear above the definition of each function in the **project.c** source file, you should comment the rest of the file sparingly. You may include comments within the body of your function definitions, where you feel it is appropriate. If there is a short block of code which is complex, or otherwise it is not immediately obvious what the code does, a short comment at the top of that block can be useful. It will help someone reading your code to understand what the purpose of the code is. However, you should not over-comment your code - that just makes it hard to read. You certainly do not want to have a comment for every line of source code!

If you have used good, meaningful variable names, then much of your code will not need commenting.

The rest of this document describes each of the tasks that you must solve.

Good luck!

Task One: "Clocking In, Clocking Out" (10 marks)

An enormous new warehouse is coming soon to your town! You have been hired to help develop various pieces of software to ensure that the warehouse opens on time and that activities run smoothly.

Recruitment has begun to hire staff for the warehouse. For your first task, you have been asked to help the payroll team. Each worker will clock in and clock out, and their pay will be calculated based on the *number of seconds* they have worked during their shift. When a worker clocks in, the current time (measured by the number of minutes and seconds that have elapsed since the warehouse opened) will be recorded. Likewise, when a worker clocks out, that time will also be recorded as the number of minutes and seconds that have elapsed since the warehouse opened.

To calculate the length of a worker's shift, you need to compute the number of seconds that have elapsed *between* them clocking in and clocking out. You need to define a function to calculate this elapsed time. The function will take four inputs (two pairs of minutes and seconds values).

Your function must compute and return how much time has elapsed, in seconds, between those two times. The clock in and clock out times may be provided as input to the function in either order – the elapsed time that you compute will always be a positive integer.

Function prototype declaration:

```
int TimeWorked(int minuteA, int secondA, int minuteB, int secondB)
```

Assumptions:

All inputs will be positive integers. Please note, the first pair of inputs (`minuteA` and `secondA`) may represent a time either before or after the second pair of inputs (`minuteB` and `secondB`).

Example:

```
printf("Time worked = %d \n", TimeWorked(1, 0, 2, 0));
printf("Time worked = %d \n", TimeWorked(55, 11, 42, 12));
printf("Time worked = %d \n", TimeWorked(33, 33, 33, 33));
```

Expected output:

```
Time worked = 60
Time worked = 779
Time worked = 0
```

Task Two: "Lucky Address" (10 marks)

The location of the warehouse is now being chosen. The boss of the warehouse is a very superstitious woman and she believes that prime numbers are extremely lucky. She therefore insists that the address of the warehouse is a prime number. However, she also prefers larger numbers to smaller numbers (thinking that larger numbers are a better omen for the company's profits).

As soon as the street is chosen, it will be necessary to locate the largest address on the street which is a prime number. To help with this, you should define a function called WarehouseAddress(). This function takes one input which is an *upper bound* on the address numbers for a given street. The function should return the *largest prime number* which is *less than* this upper bound.

Define this function to keep the boss happy!

Function prototype declaration:

```
int WarehouseAddress(int maximum)
```

Assumptions:

The input will be a positive integer greater than 2.

Example:

```
printf("Address = %d \n", WarehouseAddress(1000));
printf("Address = %d \n", WarehouseAddress(50));
printf("Address = %d \n", WarehouseAddress(104393));
```

Expected output:

```
Address = 997
Address = 47
Address = 104383
```

Task Three: "Scrolling Ads" (10 marks)

Outside the warehouse a series of large advertising displays and billboards will be installed. One type of display is a scrolling LED display where the letters rotate from right to left.

To control the characters on this display it will be useful to have a function which rotates the characters in an input string. For this task, you must define a function called Advertise() which takes a string as input. The input string represents the characters as they currently appear on the LED display.

The purpose of the function will be to rotate the characters – that is, every character must move one position to the left, and the leftmost character must "wrap-around" and join the right hand end of the string. Your function should not return any value – it should simply modify the characters in the input string.

Function prototype declaration:

```
void Advertise(char *words)
```

Assumptions:

You can assume that the input string contains at least one character (i.e. is length 1).

Example:

```
char message1[MAX_ARRAY_SIZE] = "Discount today only!";
char message2[MAX_ARRAY_SIZE] = "Hurry, hurry, hurry...";
char message3[MAX_ARRAY_SIZE] = "Good luck!";

Advertise(message1);
Advertise(message2);
Advertise(message3);

printf("Advertise = \"%s\"\n", message1);
printf("Advertise = \"%s\"\n", message2);
printf("Advertise = \"%s\"\n", message3);
```

Expected output:

```
Advertise = "iscount today only!D"
Advertise = "urry, hurry, hurry...H"
Advertise = "ood luck!G"
```

Task Four: "Unique Bid Auction" (10 marks)

Your boss wants to allow purchasers to buy goods from the warehouse using an interesting auction format. The format is something called a *unique bid auction*. A unique bid auction accepts an unlimited number of bids from many customers (each of whom pay a small amount to place each bid), and the winning bid is the lowest bid which is submitted by just one person (i.e. the lowest unique bid).

The legality of unique-bid auctions is unclear, as they border on lotteries and can therefore be classified as gambling, however your boss is a very forward thinking woman who likes to push boundaries. The following excerpt is taken from Wikipedia:

> *The legality of unique bid auctions depends on a combination of governing gambling laws and the design of the specific auction model. If an investigating authority were to determine that randomness or chance plays too large a role in the outcome, the auction may be considered a type of lottery. If, on the other hand, the investigating authority found strategy and skill played a sufficient enough role in the outcome, they may find the auction to be legal. Worldwide, there are no reported cases or statutes specifically outlawing the lowest-unique bid auction model.*
>
> **https://en.wikipedia.org/wiki/Unique_bid_auction**

Your job is therefore to define a function to calculate the winning bid in a unique bid auction. The input to the function will be the complete set of bids that have been made. The value returned by your function should be the lowest unique bid. If there is no lowest unique bid (in other words, if all bids appear more than once) then the function should return -1.

Function prototype declaration:

```
int WinningBid(int *values, int length)
```

Assumptions:

You can assume all values in the array are positive integers (greater than 0). The input array will contain at least one value (i.e. length will be at least 1).

Example:

```
int values1[MAX_ARRAY_SIZE] = {3, 7, 5, 6, 3, 4, 8, 4, 5, 8, 12, 11};
int values2[MAX_ARRAY_SIZE] = {6, 17, 6, 6, 6, 6, 12, 12, 17, 10000};
int values3[MAX_ARRAY_SIZE] = {5, 4, 3, 2, 1};

printf("Winning bid = %d \n", WinningBid(values1, 12));
printf("Winning bid = %d \n", WinningBid(values2, 10));
printf("Winning bid = %d \n", WinningBid(values3, 5));
```

Expected output:

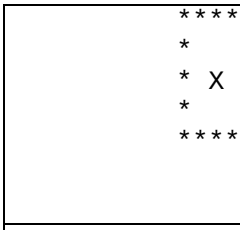
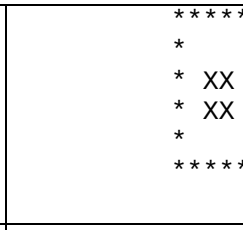
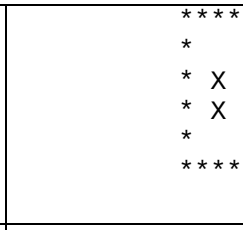
```
Winning bid = 6
Winning bid = 10000
Winning bid = 1
```

Task Five: "Box Designer" (10 marks)

The warehouse is going to contain boxes. Lots and lots of boxes. You have now been asked to help the design team produce an ASCII mock-up of what the boxes will look like. After much discussion, it has been decided that boxes will be rectangular. The warehouse workers are going to be pushing the boxes around the warehouse. The physics team have worked out that pushing a box at its exact center location is more efficient for moving the box. It will therefore be useful for the workers if the boxes have a marker printed on each side which indicates the center of that side of the box.

For this task, you must define a function called BoxDesign() which takes three inputs: a string into which the final design will be stored, the width of the box (an integer), and the height of the box (also an integer). The BoxDesign() function should generate a string representing the box (as viewed from the side). A box will simply be represented as a rectangle where the asterisk character ('*') is used to denote the edges. In addition, the *exact center* of the box must be denoted using the character 'X' (remember, the workers are going to be pushing the boxes and we don't want them to get too tired).

If the width and height of the box are both odd numbers, then there will be a single center position which can be denoted with a single 'X'. If the width and the height are both even numbers, then the center position will have to be denoted with a small square of four 'X' characters. If one of the width or height is an odd number, and the other is an even number, then two 'X' characters will be needed to denote the center position.

		
A 5x5 box (this will have a single center position)	A 6x6 box (this will have four center positions)	A 6 row, 5 column box (this will have two center positions)

Note: the length of the output string will be *exactly* **(width + 1) * height**. This is because there is a single new line character appearing at the end of each line (including the last line).

Function prototype declaration:

```
void BoxDesign(char *design, int width, int height)
```

Assumptions:

You can assume that both *width* and *height* will be at least 1. Note, the smallest box size for which you will need to include the center position is 3 (if the width or height is 2 or less, it is not possible to include the center position so there will be no 'X' characters in the string in that case).

Example:

```
char box1[MAX_ARRAY_SIZE] = {0};
char box2[MAX_ARRAY_SIZE] = {0};
char box3[MAX_ARRAY_SIZE] = {0};

BoxDesign(box1, 12, 5);
BoxDesign(box2, 15, 15);
BoxDesign(box3, 4, 4);

printf("Box 1 = \n%s\n", box1);
printf("Box 2 = \n%s\n", box2);
printf("Box 3 = \n%s\n", box3);

printf("Checking string lengths = %d %d %d\n",
                                strlen(box1), strlen(box2), strlen(box3));
```

Expected output:

```
Box 1 =
************
*          *
*    XX    *
*          *
************

Box 2 =
***************
*             *
*             *
*             *
*             *
*             *
*             *
*      X      *
*             *
*             *
*             *
*             *
*             *
*             *
***************

Box 3 =
****
*XX*
*XX*
****

Checking string lengths = 65 240 20
```

(Note: the string lengths are *exactly* **(width + 1) * height**)

Task Six: "Worker Routes" (10 marks)

The warehouse is now open for business!

The warehouse is at an enviable prime number address (thanks to your address calculator), workers are getting paid (thanks to your time calculation software), customers are seeing advertising and making bids in the unique bid auctions (thanks to your advertising and bid calculation software) and boxes of all shapes and sizes (thanks to your box design software) are being pushed around the warehouse floor by an army of workers.

However, the boss has noticed that some workers are not moving in a very efficient manner. This lack of efficiency is eating into the bottom line and the shareholders are demanding action. To improve efficiency in the warehouse, the boss wants software to calculate more efficient routes for the workers. Given the location of a worker, and the location of a box, the software should compute a direct route from the worker to the box. This computed route is then sent remotely to the worker on the warehouse floor. There is some skepticism amongst workers that the long term plan of this remote routing project is to replace them with robots, however your job is just to write the software.

For this task you must define a function called WorkerRoute(). This function takes just one input, a 2D array of integers (10 rows and 10 columns) which represents the layout of the warehouse floor. Every element in the 2D array will be equal to 0 (to represent empty space) with the exception of two values:

- the value **1** will appear exactly once and represents the location of the worker
- the value **2** will appear exactly once and represents the location of the box

The function must calculate a direct route from the worker to the box using this algorithm:

1) move the worker horizontally (left or right) if necessary, until they line up with the box
2) move the worker vertically (up or down) if necessary, until they reach the box

The route that you calculate should be indicated by setting all array elements on the route to the value **3**. Note, as implied by the algorithm above, the worker must move horizontally *first* before they move vertically. For example, the diagram on the left below shows the initial input to the function. The diagram on the right shows how the values in the array should be updated after the function has finished executing:

```
0000000000          0000000000
0000000000          0000000000
0000000000          0000000000
0000000000          0000000000
0010000000    ⇨     0013333000
0000000000          0000003000
0000002000          0000002000
0000000000          0000000000
0000000000          0000000000
0000000000          0000000000
```

Function prototype declaration:

```
void WorkerRoute(int warehouse[10][10])
```

Assumptions:

The input array is a 2D array of integers consisting of 10 rows and 10 columns. Every element in the array will be equal to zero except for two values: there will be a single "1" (representing the worker) and a single "2" (representing the box).

Example:

```
int i, j;
int warehouse1[WAREHOUSE_SIZE][WAREHOUSE_SIZE] = {
      {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
      {0, 0, 0, 0, 0, 0, 0, 0, 2, 0},
      {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
      {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
      {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
      {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
      {0, 0, 1, 0, 0, 0, 0, 0, 0, 0},
      {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
      {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
      {0, 0, 0, 0, 0, 0, 0, 0, 0, 0}
};
int warehouse2[WAREHOUSE_SIZE][WAREHOUSE_SIZE] = {
      {0, 0, 0, 0, 0, 0, 0, 0, 0, 1},
      {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
      {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
      {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
      {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
      {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
      {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
      {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
      {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
      {0, 0, 0, 0, 0, 0, 0, 0, 0, 2}
};

WorkerRoute(warehouse1);
WorkerRoute(warehouse2);

printf("Worker route 1: \n");
for (i = 0; i < WAREHOUSE_SIZE; i++) {
      for (j = 0; j < WAREHOUSE_SIZE; j++) {
            printf("%d", warehouse1[i][j]);
      }
      printf("\n");
}
printf("\nWorker route 2: \n");
for (i = 0; i < WAREHOUSE_SIZE; i++) {
      for (j = 0; j < WAREHOUSE_SIZE; j++) {
            printf("%d", warehouse2[i][j]);
      }
      printf("\n");
}
```

Expected output:

```
Worker route 1:
0000000000
0000000020
0000000030
0000000030
0000000030
0000000030
0013333330
0000000000
0000000000
0000000000

Worker route 2:
0000000001
0000000003
0000000003
0000000003
0000000003
0000000003
0000000003
0000000003
0000000003
0000000002
```

Task Seven: "One worker, no boxes" (10 marks)

Your boss has one final set of tasks for you to work on. Tasks 7 to 10 are all related, and build in functionality. For each of these final four tasks, you will be defining (and refining) a single function called MakeMove().

Your boss was very impressed by your route calculation software, however it only works in very simplistic situations where there is one box and no obstacles. The real warehouse is much more complex. Your boss now wants you to define a function called MakeMove() which allows workers on the warehouse floor to be controlled remotely.

There will be two inputs to the MakeMove() function:

1) a 2D array (10 rows and 10 columns) which represents the current state of the warehouse floor. For this task, Task 7, there are only three possible values stored in the elements of the array: **0** (displayed as ' ', representing an empty space), **1** (displayed as '#', representing a wall) and **5** (displayed as 'X', representing the worker)
2) a single character to represent the direction in which the worker should move by one position. This character will either be 'w' (up), 'a' (left), 's' (down) or 'd' (right).

Several constants have been defined in the **project.h** header file which you can make use of. The relevant constants for this task are:

```
#define SPACE 0
#define WALL 1
#define WORKER 5
```

There will be a complete border formed by a wall around the outside of the warehouse. In other words, the 2D array which represents the warehouse will have a '**1**' in rows 0 and 9 and in columns 0 and 9.

The rules for moving are as follows:

1) if there is a space in front of the worker in the direction of movement, they should move to that empty position
2) the worker cannot move if there is a wall in front of them (if an invalid move is provided, it should be ignored and the array will simply not be updated)

Return value

The MakeMove() function should return an integer. When you are implementing this task, Task 7, the function **should always return the value 0**. A value of 1 indicates that the worker has finished their job, but this will not be relevant until you reach Task 10 of this project.

Function prototype declaration:

```
int MakeMove(int warehouse[10][10], char move)
```

Assumptions:

> Every element in the outer rows and columns of the array is equal to 1 (representing the boundary wall). There may be many other values in the array equal to 1. There will be exactly one value in the array equal to 5 (representing the worker).

The MakeMove() function you define (for Tasks 7 – 10) should process **just a single move** – the move is specified by the second input to the function ("move") and the state of the warehouse before the move is given by the first input to the function ("warehouse"). The function should modify the contents of the "warehouse" array, to reflect the state of the warehouse after the single move has been made.

Example:

If the warehouse is as follows:

```
##########
#        #
#        #
#        #
#   X    #
#    #   #
#        #
#        #
#        #
##########
```

and the direction provided is 'd' (right), then the state of the warehouse should be updated as illustrated below:

```
##########
#        #
#        #
#        #
#    X   #
#    #   #
#        #
#        #
#        #
##########
```

In this state, a downward move ('s') would be illegal because there is a wall preventing the worker from moving. If an illegal move is provided as input to the function, it will be ignored and the state of the array should not change.

Task Eight: "One worker, one box" (10 marks)

For this task, you should extend your solution to Task 7. Now, the warehouse will also have a single box (displayed as 'O' and represented by the value 3) that the worker can push around. The diagram below illustrates one example of a state that the warehouse might be in:

```
##########
#     X  #
#        #
#     O  #
#        #
#        #
#        #
#        #
#        #
##########
```

The constants that have been defined for you in the **project.h** header file which are relevant for this task are:

```
#define SPACE 0
#define WALL 1
#define BOX 3
#define WORKER 5
```

The diagrams below illustrate a series of moves:

########## # X # # O # ########## (10 rows)	########## # X # # O # ########## (10 rows)	########## # X # # O # ########## (10 rows)	########## # X # # O # ########## (10 rows)
Starting state, the worker is at the top and positioned above a box	After implementing a single down move (the 's' key)	After implementing another down move. The box is pushed down by one position.	After one more down move

The rules for moving, which extend the rules from Task 7, are as follows:

1) if there is a space in front of the box that the worker is pushing, in the direction of movement, then the worker and the box can move one position in that direction
2) the box cannot be moved if there is a wall in front of it

For example, in the diagram on the right, it would **not** be allowed for the worker to move to the left (input 'a') because the box cannot be pushed onto the wall.

```
##########
#        #
#        #
#        #
#        #
#OX      #
#        #
#        #
#        #
##########
```

Function prototype declaration:

```
int MakeMove(int warehouse[10][10], char move)
```

Assumptions:

Every element in the outer rows and columns of the array is equal to 1 (representing the boundary wall). There may be many other values in the array equal to 1. There will be exactly one value in the array equal to 5 (representing the worker) and one value in the array equal to 3 (representing the box).

Return value

The MakeMove() function should return an integer. When you are implementing this task, Task 8, the function **should always return the value 0**. A value of 1 indicates that the worker has finished their job, but this will not be relevant until you reach Task 10 of this project.

Task Nine: "One worker, many boxes" (10 marks)

For this task, you should extend your solution to Task 8. Now, the warehouse can have multiple boxes (each displayed as 'O' and represented by the value 3) that the worker can push around. The diagram below illustrates one example of a state that the warehouse might be in:

```
##########
#        #
#        #
#        #
####X    #
#    OOOO#
#        #
#        #
#        #
##########
```

The rules for moving are the same as for Task 8. A box can only be pushed if there is an empty space in front of it in the direction in which it is being pushed. For example:

########## # # # X # # O # # # # OO # # # # # # # ##########	########## # # # # # X # # O # # OO # # # # # # # ##########	########## # # # # # # # XO # # OO # # # # # # # ##########	########## # # # # # # # O # # XO # # O # # # # # ##########
Starting state, there are now several boxes for the worker to push	After implementing a single down move (the 's' key). At this point, no more down moves are legal – a box cannot be pushed if there is another box in front of it.	After implementing two moves – one move to the left ('a') and another move down ('s').	After one more down move.

Function prototype declaration:

```
int MakeMove(int warehouse[10][10], char move)
```

Assumptions:

Every element in the outer rows and columns of the array is equal to 1 (representing the boundary wall). There may be many other values in the array equal to 1. There will be exactly one value in the array equal to 5 (representing the worker) and there will be at least one value in the array equal to 3 (representing a box).

Return value

The MakeMove() function should return an integer. When you are implementing this task, the function **should always return the value 0**. A value of 1 indicates that the worker has finished their job, but this will not be relevant until you reach Task 10 of this project.

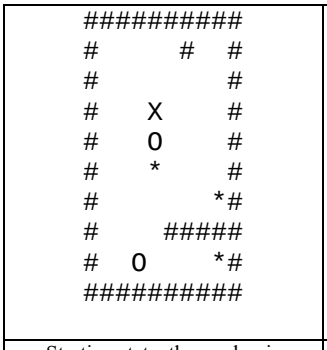
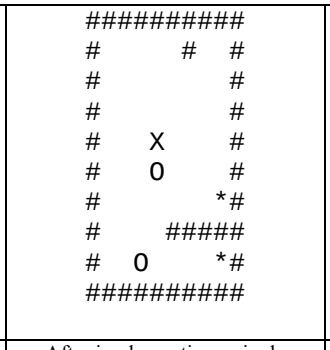
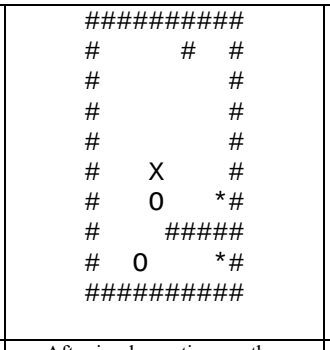
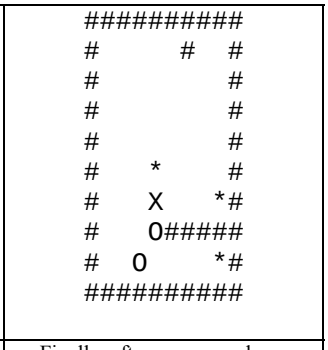
Task Ten: "One worker, many boxes and targets" (10 marks)

For this task, you should extend your solution to Task 9. Now, the warehouse floor has *targets* on it. A target is displayed as a '*' and is represented by the value 2. The diagram below illustrates one example of a state that the warehouse might be in:

```
##########
###   ####
#*XO  ####
### O*####
#*##O ####
# # * ####
#O OOO*###
#*  *  ###
##########
##########
```

The rules for moving are the same as for Task 9. However, now there is a *goal* for the worker. The goal is to push the boxes onto the targets. The number of targets is one more than the number of boxes – because there is also one target that the worker must stand on once they have reached the goal of pushing the boxes onto the targets. When all the boxes are positioned on top of targets, and when the worker is also standing on a target, then the goal has been reached!

Note: when a box is shifted onto a target, or when the worker is on top of a target, the target will not be visible. If the box or worker is moved off the target again, then the target will be visible again. Targets do not change position. The series of diagrams below illustrates this:

			
Starting state, the worker is aligned with a box which is above a target	After implementing a single down move (the 's' key). The box is on top of the target.	After implementing another down move – now the worker is on top of the target.	Finally, after one more down move, the target is visible once again.

The constants that have been defined for you in the **project.h** header file which are relevant for this task are (note, the values 4 and 6 are used to represent the position of a box or the worker when positioned on top of a target):

```
#define SPACE 0
#define WALL 1
#define TARGET 2
#define BOX 3
#define BOX_ON_TARGET 4
#define WORKER 5
#define WORKER_ON_TARGET 6
```

Function prototype declaration:

```
int MakeMove(int warehouse[10][10], char move)
```

Assumptions:

Every element in the outer rows and columns of the array is equal to 1 (representing the boundary wall). There may be many other values in the array equal to 1. Initially, there will be exactly one value in the array equal to either 5 (the worker) or 6 (the worker positioned on top of a target). There may be many values in the array equal to either 3 (a box) or 4 (a box positioned on top of a target). There may be many values in the array equal to 2 (representing targets which are not currently covered by a box or the worker).

Return value

The MakeMove() function should return an integer. After the move is complete, the function should return either 0 or 1. The function should return:

- **0** if the worker's task is not complete.
- **1** if the worker's task is complete

Task completion

The worker's task is complete when all of the boxes have been pushed onto targets and when the worker is also positioned on a target. When the worker is positioned on a target it will be represented by the value **6** (this is the special constant `WORKER_ON_TARGET` defined in **project.h**).

Once you have implemented the code for Task 10, make sure your code still satisfies the requirements for Tasks 7 – 9, especially in terms of the return value. For Tasks 7 – 9, the function should *always* return 0. For this reason, when you are checking for task completion in Task 10, make sure you include a test for the worker being on a target (i.e. having the value 6). This test will always fail for Tasks 7 – 9 (as there are no targets for those tasks), and so you will return the correct value in those scenarios.

Good luck!

BEFORE YOU SUBMIT YOUR PROJECT

Warning messages

Your code will be marked using the VS Developer Command Prompt environment. You should ensure that there are **no warning messages** produced by the compiler (using the `/W4` option from the VS Developer Command Prompt).

REQUIRED: Compile with Visual Studio before submission

The marking program uses the VS Developer Command Prompt environment.

Even if you haven't completed all of the tasks, your code **must** compile successfully. You will get some credit for a partially completed task if the expected output matches the output produced by your function. **If your code does not compile, your project mark will be 0.**

You may use any modern C environment to develop your solution, however *prior to submission* you must check that your code compiles and runs successfully using the Visual Studio Developer Command Prompt. This is not optional - it is a requirement for you to check this. During marking, if there is an error that is due to the environment you have used, and you have failed to check this using the Visual Studio Developer Command Prompt, you will receive 0 marks for the project. **If you feel this is unfair, and want to query this decision, you will be referred to Page 25 of the Project document**. Please adhere to this requirement.

In summary, before you submit your work for marking:

STEP 1:	Create an empty folder on disk
STEP 2:	Copy **just** the source files for this project (your completed **project.c** file and the unedited **test_project.c** and **project.h** files) into this empty folder
STEP 3:	Open a Visual Studio Developer Command Prompt window (as described in Lab 7) and change the current directory to the folder that contains these files
STEP 4:	Compile the program using the command line tool, with the warning level on 4: `cl /W4 *.c` If there are warnings for code you have written, **you should fix them**. You **should not submit** code that generates **any** warnings.

Do not submit code that does not compile!

And finally…

This project is an **assessed piece of coursework**, and it is essential that the work you submit reflects what you are capable of doing. You **must not copy any source code** for this project and submit it as your own work. You must also **not allow anyone to copy your work**. All submissions for this project will be checked, and any cases of copying/plagiarism will be dealt with severely. We really hope there are no issues this semester in ENGGEN131, as it is a painful process for everyone involved, so please be sensible!

Ask yourself:

have I written the source code for this project myself?

If the answer is "no", then **please talk to us before the projects are marked**.

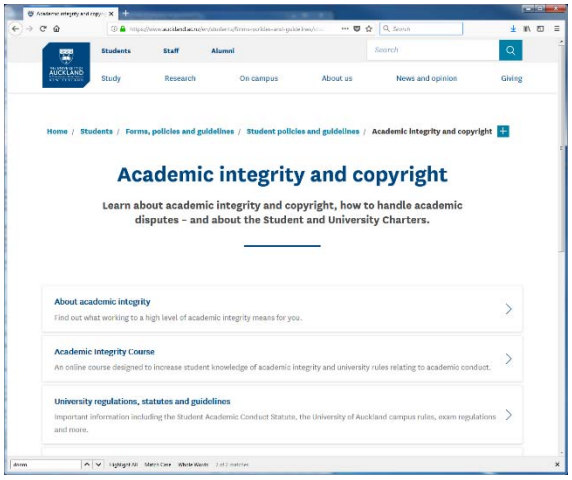
Ask yourself:

have I given anyone access to the source code
that I have written for this project?

If the answer is "yes", then **please talk to us before the projects are marked**.

Once the projects have been marked it is too late. There is more information regarding The University of Auckland's policies on academic honesty and plagiarism here:

`http://www.auckland.ac.nz/uoa/home/about/teaching-learning/honesty`



ENGGEN131 C Project 2019
Prepared by:

Dr Paul Denny
Programme Leader
School of Computer Science
The University of Auckland
New Zealand
Ph: +64 9 3737599 x 87087