ENGGEN 131 – Summer School – 2022

Project Description

Deadline: 11:59pm, Friday 11th February

Correctness: 80 marks
Code style: 10 marks

Worth: 12% of your final grade

No late submissions accepted

Introduction

Welcome to the final project for the ENGGEN131 course!

You have eight tasks to solve. For each task there is a problem description, and you must write *one function* to solve that problem. You may, of course, define *other functions* which these required functions call upon (such functions are often called "helper" functions).

Do your very best, but don't worry if you cannot complete every function. You will get credit for every task that you solve.

This must be addressed somewhere so we may as well get it out of the way – this 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**. Immediate course failure, as well as University disciplinary action, will result.

Please follow this advice while working on this 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, 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!

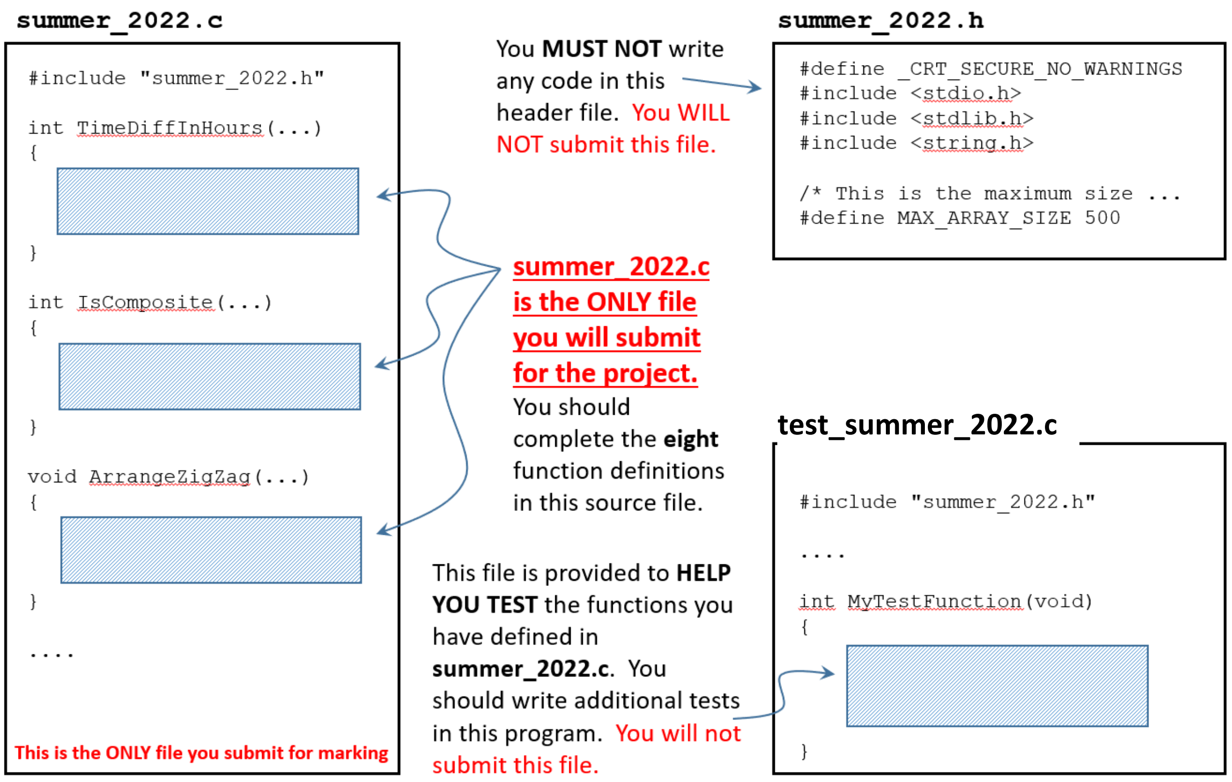
OK, now, on with the project…

Understanding the project files

There are *three files* that you will be working with when you are developing your solutions to the project tasks. The most important of these three files is **summer_2022.c**. This is the source file that you will submit for marking. Please note the following:

- **summer_2022.c** is a source file that **ONLY CONTAINS FUNCTION DEFINITIONS**
- there is no **main()** function defined in **summer_2022.c** (and you **must not** add one)
- a separate program, **test_summer_2022.c**, containing a **main()** function has been provided to you to help you test the function definitions you write in **summer_2022.c**

The diagram below illustrates the relationship between the three files.



summer_2022.c

```
#include "summer_2022.h"

int TimeDiffInHours(...)
{

}

int IsComposite(...)
{

}

void ArrangeZigZag(...)
{

}

....
```

This is the ONLY file you submit for marking

You **MUST NOT** write any code in this header file. You WILL NOT submit this file.

summer_2022.h

```
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/* This is the maximum size ...
#define MAX_ARRAY_SIZE 500
```

summer_2022.c is the ONLY file you will submit for the project. You should complete the **eight** function definitions in this source file.

This file is provided to **HELP YOU TEST** the functions you have defined in **summer_2022.c**. You should write additional tests in this program. You will not submit this file.

test_summer_2022.c

```
#include "summer_2022.h"

....

int MyTestFunction(void)
{

}
```

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 **summer_2022.h** (the header file)
- You MUST write implementations for the functions defined in **summer_2022.c**
- You SHOULD write additional test code in **test_summer_2022.c** to thoroughly test the code you write in **summer_2022.c**

Getting started

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

summer_2022.c	This is the source file that you will ultimately submit. In this source file you will find the eight functions that you should complete. Initially each function contains an *incorrect* implementation which you should *delete* and then 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.
summer_2022.h	This is the header file that contains the prototype declarations for the eight functions you have to write. You must not edit this header file in any way. Both source files (**summer_2022.c** and **test_summer_2022.c**) include this header file, and the automated marking program will use the provided definition of **summer_2022.h**. Modifying this header file in any way will be an error.
test_ summer_2022.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 this file, you should create some example inputs and then call the functions that you have defined inside the **summer_2022.c** source file. Some simple examples have been included in this file to show you how this can be done.

Place these three source files in an empty folder.

You might like to start by looking at the **summer_2022.c** source file. In this source file you will find eight function definitions, however they are all implemented *incorrectly*. The prototype declarations are as follows:

```
int TimeDiffInHours(int minA, int secA, int minB, int secB);
int IsComposite(int number);
int ArrangeZigZag(int *values, int size);
int NotebookRacksRequired(int delegates, int firstHour,
                     int unusedNotebooks, int notebooksPerRack);
void PositionOfMinimum(int *values, int rows, int cols,
                                         int *row, int *col);
void SwapAdjacent(char *words);
int CountNeuralValues(int *values, int numValues);
void DartBoard(char *board, int width, int height);
```

You need to modify and correct the definitions of these eight functions. You may add additional function definitions (i.e. “helper” functions) in this file.

Next, you should run the program in **test_summer_2022.c**. To do this, you will need to compile both source files. For example, from the Visual Studio Developer Command Prompt, you could type:

```
cl /W4 summer_2022.c test_summer_2022.c
```

Or, simply:

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

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

If you run the program, you will see that some test code has been provided for the first task of the project. You should add additional tests for this first task, and create your own tests for all of the other tasks in the project.

It is your responsibility to test the functions that you write carefully. Your functions should produce the expected output for any set of input values.

What and Where to submit

You **must not** modify **summer_2022.h**, although you can modify **test_summer_2022.c**. You will not be submitting either of these files.

You must only submit ONE source file – **summer_2022.c** – for this project to Assignment Dropbox (**https://adb.auckland.ac.nz/**). 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 function. 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 **summer_2022.h** header file already includes <stdio.h>, <stdlib.h> and <string.h>. You may not use any other functions from the standard library. If you want some functionality, you must code it!

Marking

Your submitted source file will be marked for style (use of commenting, consistent indentation, good use of additional "helper" functions rather than placing all of the logic in the required functions, etc.). Your code style will be assessed and marked by your lecturer.

For correctness, your submitted file will be marked by a program that calls your functions with lots of different input values. This program will check that your function definitions return the expected outputs for many possible inputs. Your mark will essentially be the total number of these tests that are successful, across all eight 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 eight functions from the **summer_2022.c** source file. You can simply leave the initial code in the function definition if you choose not to implement it. All eight required functions must be present in the **summer_2022.c** file you submit for marking.

Never crash

There is one thing that you must pay important attention to. Your functions must never cause the testing 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, you can make use of this constant from **summer_2022.h**:

```
#define MAX_ARRAY_SIZE 500
```

You can assume that you functions will not need to deal with arrays that are larger than this size.

Comments

You will see in the template **summer_2022.c** source file that on the line above each function definition there is a place-holder comment of the form:`/* Your comment goes here*/`

You must replace these place-holders with your own comments, written in your own words. For each function, you must briefly describe the problem that your function is trying to solve (in some sense, this will be a paraphrasing and summarising of the project task description). You must also briefly describe the algorithm that you used in your implementation for each task. You need to communicate your ideas clearly - this is a very important skill. Other than this, try to keep commenting *within* functions to a minimum.

Good luck!

Task One: "Time span in hours" (10 marks)

Write a function that is passed four integer inputs representing two different time periods (each expressed as a number of minutes and seconds). The function should calculate and return the difference between those two time periods expressed **in hours (*double* type)**.

Function prototype declaration:

```
double TimeDiffInHours(int minA, int secA,
                                        int minB, int secB);
```

Assumptions:

You can assume that all four input values will be greater than or equal to 0.

Example:

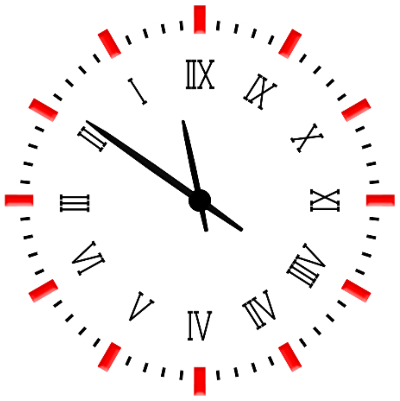
```
double diff1, diff2, diff3;

diff1 = TimeDiffinHours(500, 45, 46, 468);
diff2 = TimeDiffinHours(70, 30, 309, 45);
diff3 = TimeDiffinHours(20, 0, 0, 0);

printf("%f %f %f \n", diff1, diff2, diff3);
```

Expected output:

```
7.449167 3.987500 0.333333
```



Task Two: “Is Composite?” (10 marks)

A **composite number** is a positive integer that can be formed by multiplying two smaller positive integers. Equivalently, it is a positive integer that has at least one divisor other than 1 and itself. Every positive integer is composite, prime, or the unit 1, so the composite numbers are exactly the numbers that are not prime and not a unit.

For example, the integer 14 is a composite number because it is the product of the two smaller (than 14) integers 2 × 7. Likewise, the integers 2 and 3 are not composite numbers because each of them can only be divided by one and itself. The first few composite numbers are 4,6,8 and 9.

Write a function that can check if the number passed as parameter to the function is a composite number or not. The function must return 1 if the number is composite, otherwise it should return 0.

Function prototype declaration:

```
int IsComposite(int number);
```

Assumptions:

You may assume the input is positive and greater than 1.

Example:

```
printf("Composite check returned: %d\n", IsComposite(10));
printf("Composite check returned: %d\n", IsComposite(99));
printf("Composite check returned: %d\n", IsComposite(127));
```

Expected output:

```
Composite check returned: 1
Composite check returned: 1
Composite check returned: 0
```

Task Three: “Arrange them Zig Zag!” (10 marks)

Write a function that locates the middle element of an integer array, and then traverses and updates the elements of the array as described below. *The array must be updated to show the traversed order of the elements.*

The traversal starts from the middle element, then visits it’s immediate left element in the array, followed by the immediate right element of the middle element, and then goes back to the middle element. This process is repeated starting from the middle element, and in every iteration jumping to the next subsequent elements on each of the two sides of the middle element. The iteration ends at the last element of the array.

The function must traverse each array element exactly once. The function does not return anything. It just updates the order of the array elements. See the example below. The function takes two inputs: an integer array containing the elements and an integer representing the size of the array.

Function prototype declaration:

```
void ArrangeZigZag(int *values, int size);
```

Assumptions:

The number of elements in the array is always *odd*.

Example:

```
int numbers[7] = {-3, -2, -1, 0, 1, 2, 3};

ArrangeZigZag(numbers, 7);

for(int i = 0; i < 7; i++){
     printf(“%d ”, numbers[i]);
}
```

Expected output:

```
0 -1 1 -2 2 -3 3
```

Task Four: "Estimate the conference notebooks?" (10 marks)

You are organizing a one-day conference for which you need to order notebooks. You must have an estimate of the number of notebooks that will be required by the delegates. Each delegate must be given exactly one notebook. The notebooks are placed in notebook racks in a stationary room. You need to order notebooks in terms of the number of racks. Your order placed must be just enough for all the delegates, so that nobody misses out, but at the same time there is no wastage too.

The conference is scheduled for 6 one-hour sessions on the day. Each delegate will need exactly one notebook each session, except for the first session when they would need exactly two notebooks per delegate. The notebooks that remain unused may be distributed for the subsequent hour.

Write a function **NotebookRacksRequired()** that takes four integer inputs (assume all inputs to be non-negative): the number of delegates, an integer that determines if it is the first session or not (1 or 0), the number of unused notebooks from the previous session, and the number of notebooks on sale per notebook rack. The last (4th) input may be assumed to be greater than or equal to 1.

This function must return the minimum number of notebooks needed each hour, assuring no delegate gets zero notebooks.

Function prototype declaration:

```
    int NotebookRacksRequired(int delegates, int firstHour,
                     int unusedNotebooks, int notebooksPerRack);
```

Example:

```
printf("Racks Count = %d\n", NotebookRacksRequired(51, 1, 7, 9));
printf("Racks Count = %d\n", NotebookRacksRequired(24, 0, 4, 13));
printf("Racks Count = %d\n", NotebookRacksRequired(110, 1, 1, 20));
```

Expected output:

```
    Racks Count = 11
    Racks Count = 2
    Racks Count = 11
```

Task Five: “Locate the minimum!” (10 marks)

A matrix, or two-dimensional grid of numbers, can be represented using a one-dimensional array where all of the values in the matrix are listed one row after another.

For this task, you must write a function which takes as input a one-dimensional array (representing a matrix) along with the number of rows and columns in the matrix. You must calculate the row and column position of the smallest value in the matrix. The last two inputs to the function are pointers into which you should store this result.

<u>Function prototype declaration:</u>

```
void PositionOfMinimum(int *values, int rows, int cols,
                                             int *row, int *col);
```

<u>Assumptions:</u>

You can assume the number of elements provided in the first input match the given dimensions of the matrix, and that any input array will have at least one element and a unique minimum value (i.e. there won’t be two values which are both the smallest).

<u>Example:</u>

```
int minRow, minCol;
int values[15] =
                {4,2,7,70,20,90,732,612,108,44,65,52,0,1,9};

PositionOfMinimum(values, 5, 3, &minRow, &minCol);
printf("Minimum is at row: %d and col: %d", minRow,
                                                     minCol);
```

<u>Expected output:</u>

```
Minimum is at row: 4 and col: 0
```

Task Six: "Electronic Banner" (10 marks)

A shop needs to display its electronic banner using LEDs.

For a dynamic display, you have decided to momentarily randomize the characters on the electronic board. To implement this effect on this display, it will be useful to have a function which swaps the characters in an input string following a special rule (next paragraph). For this task, you must define a function called `SwapAdjacent()` which takes a string as input. The input string represents the characters as they currently appear on the electronic display.

The purpose of the function will be to swap the adjacent characters – that is, starting from the first letter, every pair of letters must be swapped by the function. For example, the first and the second letters must be swapped; similarly third and the fourth letters must be swapped and so on. In case the number of characters in the string is odd, then the last character must not be affected by the function.

Your function must not return any value – it should simply modify the characters in the input string.

Function prototype declaration:

```
void SwapAdjacent(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] = "buy 2 get 1 free!";
char message2[MAX_ARRAY_SIZE] = "special festival offers";
char message3[MAX_ARRAY_SIZE] = "we are open!";

SwapAdjacent(message1);
SwapAdjacent(message2);
SwapAdjacent(message3);

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

Expected output:

```
Display = "ub y 2eg t 1rfee!"
Display = "psceai leftsivlao ffres"
Display = "ewa ero ep!n"
```

Task Seven: “Who is neutral?” (10 marks)

Write a function that is passed two inputs: an array of integers, and the number of elements in the array. The array has both positive and negative values. There is no 0 value in the array. The function should return the number of distinct (i.e. different) values, for each of which the sum of their positive and negative instances comes out to be zero i.e. the overall effect of this/these value(s) is *neutral*.

For example, if the array is {11, -11, 12, -12, 3}, then the function must return 2, i.e. the number of unique values (11 and 12 in this case) for which the addition of their positive and negative instances comes out to be 0.

Function prototype declaration:

```
int CountNeuralValues(int *values, int numValues);
```

Assumptions:

You can assume that there will be at least one element in the input array.

Example:

```
int valuesA[7] = {-2, -4, 1, 4, -3, 2, -1};
int valuesB[5] = {-11, -11, 11, 23, -23};
int valuesC[7] = {50, 50, -50, 40, 60, -50, -40};

int result1, result2, result3;

result1 = CountNeuralValues(valuesA, 10);
result2 = CountNeuralValues(valuesB, 5);
result3 = CountNeuralValues(valuesC, 7);

printf("%d %d %d \n", result1, result2, result3);
```

Expected output:

```
3 1 2
```

Task Eight: "The Dart Board" (10 marks)

You have now been asked to design a rectangular dart board for a Darts PC game with a darting target at the center of the dart board.

Write a function called DartBoard() which takes three inputs: a string representing the board, the board's width (an integer), and the board's height (also an integer). This function should update the string representation of the board. The board should be drawn as a rectangle where the hash character ('#') is used to represent the four sides of the board. The center of the board must be represented using the O (capital 'o') character.

Depending upon the height and width of the board, the number of 'O' characters must be printed at the center: If the width and height are both odd numbers, then only one 'O' instance will be printed at the exact center. If the width and the height are both even numbers, then the center position will be represented with a small square of four 'O' characters. If one of the width or height is an odd number, and the other is an even number, then two 'O' characters will be needed to denote the center position.

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).

<u>Function prototype declaration:</u>

```
void DartBoard(char *board, int width, int height);
```

<u>Assumptions:</u>

You can assume that both *width* and *height* will be at least 1. Note, the smallest board 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 'O' characters in the string in that case).

```
#####          ######          #####
#   #          #    #          #   #
# O #          # OO #          # O #
#   #          # OO #          # O #
#####          #    #          #   #
               ######          #####
```

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

Example:

```
      char board1[MAX_ARRAY_SIZE] = {0};
      char board2[MAX_ARRAY_SIZE] = {0};
      char board3[MAX_ARRAY_SIZE] = {0};

      BoxDesign(board1, 12, 5);
      BoxDesign(board2, 15, 15);
      BoxDesign(board3, 4, 4);

      printf("Dart Board 1 = \n%s\n", board1);
      printf("Dart Board 2 = \n%s\n", board2);
      printf("Dart Board 3 = \n%s\n", board3);

      printf("Checking string lengths = %d %d %d\n",
                              strlen(board1), strlen(board2), strlen(board3));
```

Expected output:

```
Dart Board 1 =
############
#          #
#    OO    #
#          #
############

Dart Board 2 =
###############
#             #
#             #
#             #
#             #
#             #
#             #
#      O      #
#             #
#             #
#             #
#             #
#             #
#             #
###############

Dart Board 3 =
####
#OO#
#OO#
####

Checking string lengths = 65 240 20
```

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

BEFORE YOU SUBMIT YOUR PROJECT

Warning messages

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

Even if you haven't completed all of the tasks, your code **must** compile successfully. You will get some credit for partially completed tasks 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 failed to check this using the Visual Studio Developer Command Prompt, you will receive 0 marks for the project. 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 (the summer_2022.c and test_summer_2022.c source files and the unedited summer_2022.h header file) 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!

Style marking - components (10 marks)

To be eligible to earn full marks for style, you will need to have completed *at least half* of the eight required functions. Even if you have not completed all of the eight functions, you should still leave the templates of all eight functions in your submitted file (do not delete any of the eight functions from the template that was provided to you). The following provides a brief description of what style components will be assessed:

Comments (5 marks):
Read the description of what is required on Page 7 of the project document. You must write a comment for each of the required functions you have implemented at the very top of the function definition (by replacing the placeholder comment: "Your comment goes here" in the template file provided to you). Each function's comment should describe (in your own words) the problem that the function is solving, and (also in your own words) the approach that you took to solve the function. The expectation is that this comment will be a short paragraph, consisting of at least several sentences (written in your own words) that would serve as useful documentation for someone who wanted to understand what your code is all about. You are welcome to also include short comments within the function body, however you should avoid "over-commenting" the code body - this marks it hard to read.

Indentation (2 marks):
Your code should be indented consistently and laid out neatly. There are many examples in the coursebook (particularly at the end of each lecture) that you can refer to here, as well as examples on page 3. There is also a brief style guide on page 4. It is recommended that you follow these style guidelines, however if you strongly prefer a different style (such as placing the opening brace for an if statement on a new line, which differs from the advice on page 4 under the heading "Braces for other blocks of code") then that is fine - as long as you apply that style consistently throughout your source file. You should also lay out your code neatly - for example, do not place blank lines between every single line of code, but rather separate short "blocks" of code (lines that are related) with a single blank line.

Helper functions (3 marks):
You should define at least two "helper" functions, and call these functions from one or more of the required functions. All of the helper function definitions should appear at the *top of your source file* (where the comment "HELPER FUNCTIONS" appears in the template file provided to you) so that it is easy for your marker to locate them. You should apply the same style elements to these helper functions - that is, they must begin with a comment describing the purpose of the function - and you should also mention which of the required functions make use of each helper function. A good reason to define a helper function is to reduce the complexity of one of the required functions - particularly if the code would otherwise be particularly long. A good "rule of thumb" (derived from Google's style conventions for C-based code) is that if the length of a function exceeds about 40 lines of code then you should think carefully about whether a helper function could be used to reduce this length. Your marker will not be counting your lines of code exactly, so this 40-line rule is not a strict limit, but should serve as a useful guideline.

The final word

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!

<u>Ask yourself:</u>

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**.

<u>Ask yourself:</u>

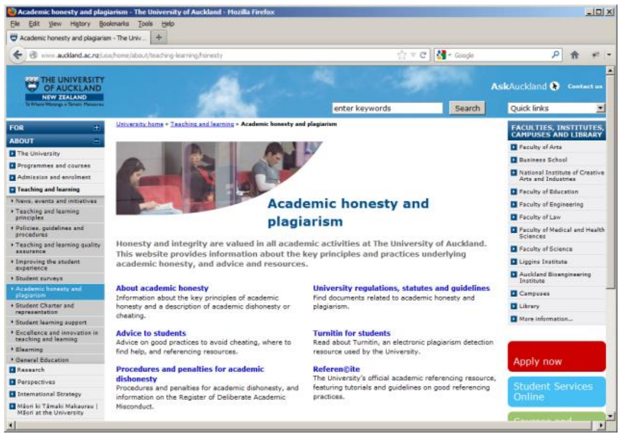
have I given <u>anyone</u> access to the source code
that I have written for this project?

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


Academic honesty and plagiarism - The University of Auckland - Mozilla Firefox
File Edit View History Bookmarks Tools Help
Academic honesty and plagiarism - The Univ...
www.auckland.ac.nz/uoa/home/about/teaching-learning/honesty
Google
THE UNIVERSITY OF AUCKLAND NEW ZEALAND
AskAuckland
Contact us
enter keywords
Search
Quick links
FOR
ABOUT
The University
Programmes and courses
Admission and enrolment
Teaching and learning
News, events and initiatives
Teaching and learning principles
Policies, guidelines and procedures
Teaching and learning quality assurance
Improving the student experience
Student surveys
Academic honesty and plagiarism
Student Charter and representation
Student learning support
Excellence and innovation in teaching and learning
Elearning
General Education
Research
Perspectives
International Strategy
Māori ki Tāmaki Makaurau | Māori at the University
FACULTIES, INSTITUTES, CAMPUSES AND LIBRARY
Faculty of Arts
Business School
National Institute of Creative Arts and Industries
Faculty of Education
Faculty of Engineering
Faculty of Law
Faculty of Medical and Health Sciences
Faculty of Science
Liggins Institute
Auckland Bioengineering Institute
Campuses
Library
More information...
Apply now
Student Services Online

University home » Teaching and learning » Academic honesty and plagiarism

Academic honesty and plagiarism

Honesty and integrity are valued in all academic activities at The University of Auckland. This website provides information about the key principles and practices underlying academic honesty, and advice and resources.

About academic honesty
Information about the key principles of academic honesty and a description of academic dishonesty or cheating.

University regulations, statutes and guidelines
Find documents related to academic honesty and plagiarism.

Advice to students
Advice on good practices to avoid cheating, where to find help, and referencing resources.

Turnitin for students
Read about Turnitin, an electronic plagiarism detection resource used by the University.

Procedures and penalties for academic dishonesty
Procedures and penalties for academic dishonesty, and information on the Register of Deliberate Academic Misconduct.

Referen©ite
The University's official academic referencing resource, featuring tutorials and guidelines on good referencing practices.

~Dr Paramvir Singh
Professional Teaching Fellow
School of Computer Science