ENGGEN 131 – Semester Two – 2017

C Programming Project

Deadline (Style marking): 11:59pm, Sunday 22nd October
Deadline (Correctness marking): 11:59pm, Friday 27th October

Worth: 12% of your final grade

<u>No late submissions accepted</u>

Introduction

Welcome to the final project for the ENGGEN131 course!

You have ten 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.

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

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.

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, 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!

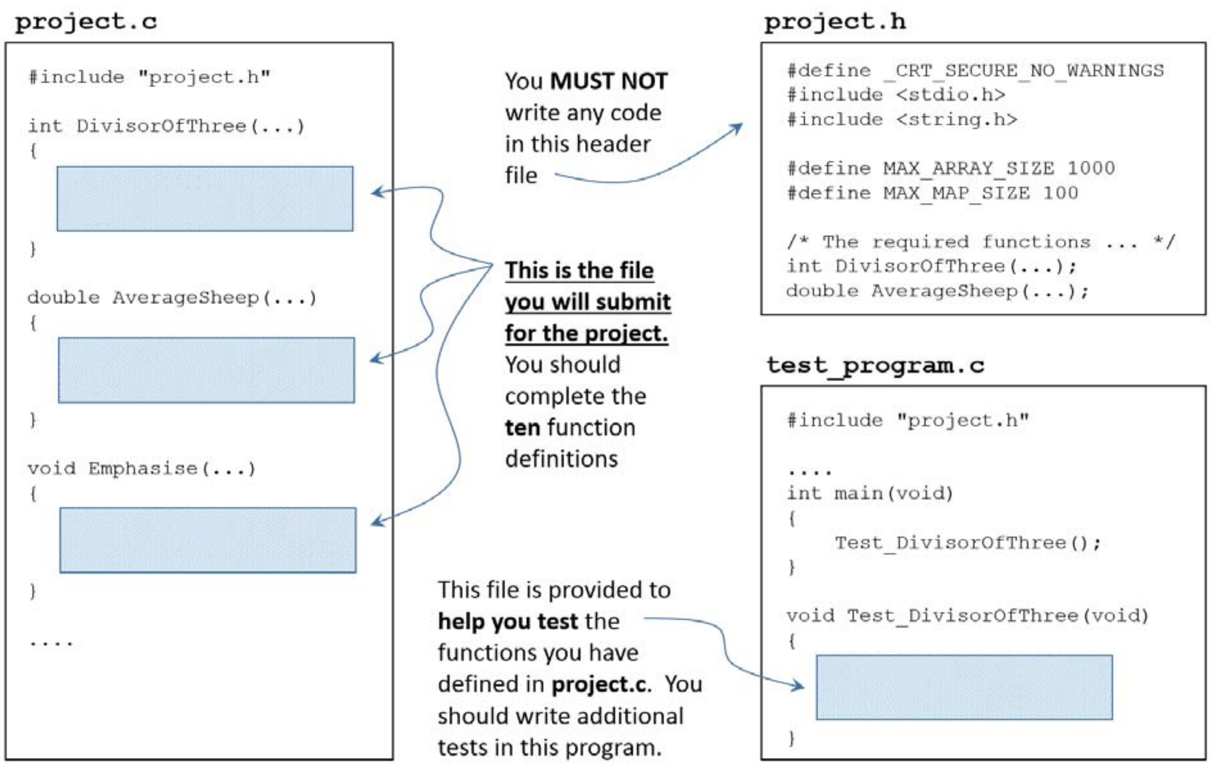
OK, now, on with the project…

Understanding the project files

There are *three files* that you will be working with when you are developing your solutions to the project tasks. The most important of these three files is **project.c**. This is the source 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_program.c**, containing a **main()** function has been provided to you to help you test the function definitions you write in **project.c**

The diagram below illustrates the relationship between the three files.



project.c

```
#include "project.h"

int DivisorOfThree(...)
{

}

double AverageSheep(...)
{

}

void Emphasise(...)
{

}

....
```

You **MUST NOT** write any code in this header file

This is the file you will submit for the project. You should complete the **ten** function definitions

This file is provided to **help you test** the functions you have defined in **project.c**. You should write additional tests in this program.

project.h

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

#define MAX_ARRAY_SIZE 1000
#define MAX_MAP_SIZE 100

/* The required functions ... */
int DivisorOfThree(...);
double AverageSheep(...);
```

test_program.c

```
#include "project.h"

....
int main(void)
{
    Test_DivisorOfThree();
}

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

Getting started

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

project.c	This is the source file that you will ultimately submit. In this source file you will find the ten 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.
project.h	This is the header file that contains the prototype declarations for the ten functions you have to write. You must not edit this header file in any way. Both source files (**project.c** and **test_program.c**) include this header file, and the automated marking program will use the provided definition of **project.h**. Modifying this header file in any way will be an error.
test_program.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 **project.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 **project.c** source file. In this source file you will find ten function definitions, however they are all implemented *incorrectly*. The prototype declarations are as follows:

```
int DivisorOfThree(int a, int b, int c);
double AverageSheep(int *counts);
void Emphasise(char* word);
int PrimeFactors(int n, int *factors);
void ConnectTwo(int maze[10][10]);
void DayTrader(int *prices, int numPrices, int *bestRun, int *bestRunIndex);
void Compress(int *input, int *output);
void AddOne(char *input, char *output);
void Histogram(char *result, int *values, int numValues);
void GoldRush(int *results, int rows, int cols, int
                                 map[MAX_MAP_SIZE][MAX_MAP_SIZE], int bonus);
```

You need to modify and correct the definitions of these ten functions.

Next, you should run the program in **test_program.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 project.c test_program.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 Project - Semester Two - 2017

Welcome to the minimal test program for Project Two.

This  test  program  provides  an  absolute minimal set of  test  cases
that you can use to automatically test the functions you  have  defined
for the project. Failing any of these tests is an indication that there
is an error in your implementation.  However, because this is a minimal
set of tests,  passing all of them is no guarantee that your  functions
are  defined  correctly.  It  is  up  to  you  to  test your code more
thoroughly, but hopefully this template will be a useful guide for you.

Good luck!

Task One:   DivisorOfThree() - not yet implemented
Task Two:   AverageSheep()   - not yet implemented
Task Three: Emphasise()      - not yet implemented
Task Four:  PrimeFactors()   - not yet implemented
Task Five:  ConnectTwo()     - not yet implemented
Task Six:   DayTrader()      - not yet implemented
Task Seven: Compress()       - not yet implemented
Task Eight: AddOne()         - not yet implemented
Task Nine:  Histogram()      - not yet implemented
Task Ten:   GoldRush()       - not yet implemented
```

Notice that initially the output from each test function is:

```
      not yet implemented
```

When you attempt to implement each function, you will get feedback on whether the function passes or fails the basic tests that are provided. There are three basic tests provided for each function (although you should create more tests of your own!). Therefore, ideally, you want to see this output for each function:

```
      PASS PASS PASS
```

What to submit

You **must not** modify **project.h**, although you can modify **test_program.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 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 **project.h** header file already includes <stdio.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

When you submit **project.c** for the Style marking deadline (Sunday 22nd October), this 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 a teaching assistant.

When you submit **project.c** for the Correctness marking deadline (Friday 27th October), this 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 ten 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 ten functions from the **project.c** source file. You can simply leave the initial code in the function definition if you choose not to implement it. All ten required functions must be present in the **project.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 (for Tasks 1-9), you can make use of this constant from **project.h**:

```
#define MAX_ARRAY_SIZE 1000
```

If you need to declare a 2-dimensional array in Task 10), you can make use of this constant from **project.h**:

```
#define MAX_MAP_SIZE 100
```

Comments

You will see in the template **project.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: "No remainders" (10 marks)

The *greatest common divisor* (or GCD) of a set of numbers is the largest positive integer that divides all numbers in the set without remainder. Define a function called **DivisorOfThree()** that is passed three integer inputs. The function must return the *GCD* of these three numbers. If any of the input numbers is less than or equal to zero, then the function must return -1.

Function prototype declaration:

```
int DivisorOfThree(int a, int b, int c)
```

Assumptions:

> You cannot assume the inputs are positive. If *any* input is negative, or equal to zero, then the function must return -1.

Example:

```
printf("GCD = %d\n", DivisorOfThree(1288, 759, 1173));
printf("GCD = %d\n", DivisorOfThree(760, 1960, 2720));
printf("GCD = %d\n", DivisorOfThree(100, 0, 1000000));
```

Expected output:

```
GCD = 23
GCD = 40
GCD = -1
```

Task Two: “Counting sheep” (10 marks)

A device has been installed on a gate to count the number of sheep that pass by every hour. The owner of the farm wants to compute the *average* number of sheep passing each hour. The device provides data in the form of an array, where each element in the array is the number of sheep that were counted in a one hour period. However, the device is not perfectly reliable and sometimes if fails. In such cases, the value -1 will appear in the data rather than an accurate hourly reading. You should *ignore* these erroneous values when calculating the average. A special value, 9999, is inserted at the end of the list of data values to indicate that there is no more data to process.

Define a function called **AverageSheep()** that is passed one input: an array of integers containing the count data. The function must calculate the *average* of all values in the array up to, but excluding, the value 9999 (which indicates the end of the data sequence). You should also ignore any values equal to -1 when performing your calculation. If there are **no valid values** with which to compute the average, then the function should return 0.0.

Function prototype declaration:

```
double AverageSheep(int *counts)
```

Assumptions:

You can assume the array will contain the value 9999 (to indicate the end of the data).

This special value, 9999, may appear anywhere (for example, it may be the first value in the array). If there are no valid data values with which to compute the average, then the function should return 0.0.

Example:

```
int sheep1[MAX_ARRAY_SIZE] = {25, 12, 18, 19, 9999};
int sheep2[MAX_ARRAY_SIZE] = {-1, 25, 12, 18, -1, 9999};
int sheep3[MAX_ARRAY_SIZE] = {-1, 22, 9999, -1, 25, 12};
int sheep4[MAX_ARRAY_SIZE] = {-1, -1, 9999, -1, 25, 12};

printf("Average = %f\n", AverageSheep(sheep1));
printf("Average = %f\n", AverageSheep(sheep2));
printf("Average = %f\n", AverageSheep(sheep3));
printf("Average = %f\n", AverageSheep(sheep4));
```

Expected output:

```
Average = 18.500000
Average = 18.333333
Average = 22.000000
Average = 0.000000
```

Task Three: "Emphasise" (10 marks)

Sometimes people use two underscore characters to emphasise certain words within a sentence. For example, consider the following sentence:

```
This is a _really_ fun project!
```

In this case, the two underscore characters are used to emphasise the word "really". Another way that this emphasis could have occurred is using capital letters (instead of the underscore characters):

```
This is a REALLY fun project!
```

The advantage of this second approach is that the string is shorter - it uses two fewer characters! Define a function called **Emphasise()** that is passed one string input. The string *will* contain *exactly* two underscore characters. The function should *modify* the input string by converting all **alphabetic** characters (i.e. 'a'-'z') *between* the two underscore characters to upper case. In addition, the underscore characters should *no longer appear* in the resulting string.

Function prototype declaration:

```
void Emphasise(char* word)
```

Assumptions:

You can assume there will be *exactly* two underscore characters somewhere in the input string. Only lowercase alphabetic characters ('a'-'z') should be capitalized.

Example:

```
char wordsA[MAX_ARRAY_SIZE] = "this is a _good_ question!";
char wordsB[MAX_ARRAY_SIZE] = "It is _over 9000_!";
char wordsC[MAX_ARRAY_SIZE] = "_Nothing to see here_";

Emphasise(wordsA);
Emphasise(wordsB);
Emphasise(wordsC);

printf("%s\n", wordsA);
printf("%s\n", wordsB);
printf("%s\n", wordsC);
```

Expected output:

```
this is a GOOD question!
It is OVER 9000!
NOTHING TO SEE HERE
```

Task Four: “Prime factors” (10 marks)

The *prime factors* of a positive integer are the prime numbers that divide that integer exactly. The *prime factorisation* of a positive integer is a list of all of the integer's prime factors. For example, the prime factorisation of 234 is: 2, 3, 3 and 13. Every integer has a unique prime factorisation.

Define a function called **PrimeFactors()** that is passed two inputs: an integer and an array of integers. The function should compute the prime factorisation of the first input, and store each of the prime factors (in increasing order) in consecutive elements of the array. The function should *return* the number of factors.

Function prototype declaration:

```
int PrimeFactors(int n, int *factors)
```

Assumptions:

You can assume the integer input, *n*, is greater than or equal to 2.

Example:

```
int numFactors, i;
int factors[MAX_ARRAY_SIZE];

printf("Factors of 567: ");
numFactors = PrimeFactors(567, factors);
for (i = 0; i < numFactors; i++) {
     printf("%d ", factors[i]);
}

printf("\nFactors of 5678901: ");
numFactors = PrimeFactors(5678901, factors);
for (i = 0; i < numFactors; i++) {
     printf("%d ", factors[i]);
}
```

Expected output:

```
Factors of 567: 3 3 3 3 7
Factors of 5678901: 3 3 17 37117
```

Task Five: “Connect Two” (10 marks)

For this task you need to calculate the shortest path between two points on a grid. The starting location will be labelled ‘1’ and the destination location will be labelled ‘2’. The shortest path will initially follow a diagonal (if necessary) followed by a vertical or horizontal path (if necessary).

Define a function called **ConnectTwo()** which is passed one input: a 2-dimensional array (10 rows and 10 columns) of integers. This 2-dimensional array represents a map. All elements in the array will be equal to zero, except for the starting location (which has the value 1) and the ending location (which has the value 2). Your function should determine the *shortest path* between these two locations and then *modify the array* by setting each element *on the path* to the value 3. The starting and ending locations should remain unchanged. The diagram below illustrates this (the input map is on the left, and the resulting modified map is on the right):

```
0000000100
0000000000
0000000000
0000000000
0000000000
0000000000
0000000000
0000000000
0002000000
0000000000
```

Element 1 represents the starting location and element 2 represents the destination

```
0000000100
0000003000
0000030000
0000300000
0003000000
0003000000
0003000000
0003000000
0002000000
0000000000
```

The shortest path between these two locations is illustrated using the value 3

Function prototype declaration:

```
void ConnectTwo(int maze[10][10])
```

Assumptions:

You can assume the input array dimensions are always 10x10, and that every element is initialised to zero except for the starting and ending locations (which will be initialised to 1 and 2 respectively).

<u>Example:</u>

```
     int i, j;
     int map[10][10] = {
          {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},
          {0, 0, 0, 2, 0, 0, 0, 0, 0, 0}
     };

     ConnectTwo(map);

     printf("\n");
     for (i = 0; i < 10; i++) {
          for (j = 0; j < 10; j++) {
               printf("%d", map[i][j]);
          }
          printf("\n");
     }
```

<u>Expected output:</u>

```
     1000000000
     0300000000
     0030000000
     0003000000
     0003000000
     0003000000
     0003000000
     0003000000
     0003000000
     0002000000
```

Task Six: “The Wolf of Wall Street” (10 marks)

Your friend is trying to make money on the share market and has asked you to help them analyse the price data for a number of stocks they are looking at. In particular, they are interested in finding “runs” in the data, where a run consists of an increasing sequence of prices (moving left to right). For example, consider the data below which shows 10 prices changing over time:

123, 120, 118, 119, 121, 126, 127, 130, 129, 132

The longest “run” in this sequence of data values begins with the value 118. Starting with the value 118, the next 5 values are in strictly increasing order (119, 121, 126, 127, 130). The length of this “run” is therefore 5. Your friend wants you to write a program which takes a sequence of prices as input and calculates two things: where the longest “run” begins in the data and the length of the longest “run”.

Define a function called **DayTrader()** which takes four inputs: an array of integers representing the price information, the length of the array (i.e. the number of prices), and two integer pointers which you should use to store the calculated output values. The first pointer records the address where you should store the *length* of the best “run”, and the second pointer records the address where you should store the *index position* of the *start* of the “run”.

Function prototype declaration:

```
void DayTrader(int *prices, int numPrices, int *bestRun,
                                          int *bestRunIndex)
```

Assumptions:

You can assume that the array will contain at least one price (i.e. the length of the array will be greater than 0).
Runs only consist of *strictly* increasing values (two consecutive equal values do not constitute a “run”)
If there are two or more “runs” of the *same length* in the array, then you must return the *smallest* index position (i.e. left most value) when reporting the start of the run (i.e. the `bestRunIndex`)

Example:

```
int pricesA[15] = {59, 60, 55, 23, 42, 44, 48, 50, 43, 45,
                                        43, 44, 47, 51, 52};

int pricesB[10] = {1, 2, 3, 3, 3, 4, 3, 4, 5, 6};

int pricesC[10] = {123, 120, 118, 119, 121, 126, 127, 130,
                                                  129, 132};

int bestRun, bestRunIndex;

DayTrader(pricesA, 15, &bestRun, &bestRunIndex);
printf("Best run = %d, best run index = %d\n", bestRun,
                                              bestRunIndex);

DayTrader(pricesB, 10, &bestRun, &bestRunIndex);
printf("Best run = %d, best run index = %d\n", bestRun,
                                              bestRunIndex);

DayTrader(pricesC, 10, &bestRun, &bestRunIndex);
printf("Best run = %d, best run index = %d\n", bestRun,
                                              bestRunIndex);
```

Expected output:

```
Best run = 4, best run index = 3
Best run = 3, best run index = 6
Best run = 5, best run index = 2
```

Task Seven: "Compression" (10 marks)

Compression is an important technique for reducing the amount of space required to store information. This is of particular importance when large amounts of data need to be transmitted from one place to another. Define a function called **Compress()** which implements a basic compression algorithm. The input data to be compressed is represented simply as an array of positive integers. The end of the input data is indicated by the special value -1.

Compress the data by counting how many times a value is repeated *consecutively*. You can then represent that data value with two values: the number of repetitions of the value and the value itself. For example, if the data consists of eight "10"s: 10, 10, 10, 10, 10, 10, 10, 10 then this can be represented in compressed form as just two values: 8, 10.

Of course, this algorithm is not very effective if values aren't repeated consecutively in the data. In the worst case, when there is no repetition, the compressed data will be double the size of the original data! You should store the compressed data in the second input to the function, which is also an array of integers. *Don't forget to place the value -1 at the end of the compressed data!*

Function prototype declaration:

```
void Compress(int *input, int *output)
```

Assumptions:

The input array will always contain the special value -1 (indicating the end of the data).
The input array will consist of *at least one* value to be compressed (i.e. before the -1).
Ensure that you add the value -1 to the end of the compressed data in the output array.

Example:

```
int input[MAX_ARRAY_SIZE] =
          {7,7,7,7,7,3,4,4,4,7,0,0,0,0,0,0,0,0,0,0,0,0,-1};
int output[MAX_ARRAY_SIZE];
int i;

Compress(input, output);

i = 0;
while (output[i] != -1) {
     printf("%d ", output[i]);
     i++;
}
```

Expected output:

```
5 7 1 3 3 4 1 7 12 0
```

Task Eight: "Arbitrary Incrementing" (10 marks)

In C, the `int` type is limited to (typically) 32 bits. The following example, where 1 is added to the largest positive integer value, illustrates *integer overflow*:

```
int value = 2147483647;
int result = value + 1;
printf("%d + 1 = %d", value, result);
```

In this case, the output would be:

```
2147483647 + 1 = -2147483648
```

If we want to represent arbitrarily large integers, then we could use an *array* where each element in the array represents a digit in the number. In fact, we could use strings for this purpose, and have the characters in the string represent the digits. We could then define special functions to perform arithmetic. As an example, consider the following code:

```
char value[MAX_ARRAY_SIZE] = "2147483647";
char output[MAX_ARRAY_SIZE];
AddOne(value, output);
printf("%s + 1 = %s", value, output);
```

In this case, we use the string "2147483647" to represent a large integer value. The **AddOne()** function is then used to compute a new string where the characters represent the integer that is one larger than the original. The output from the code above is:

```
2147483647 + 1 = 2147483648
```

This apparently solves the integer overflow problem! However, there are some downsides - arithmetic performed in this way is *much* slower than using the `int` type.

We can now perform this basic arithmetic on really large numbers:

```
char value[MAX_ARRAY_SIZE] = "123456789123456789123456789";
char output[MAX_ARRAY_SIZE];
AddOne(value, output);
printf("%s + 1 = %s", value, output);
```

and the output is:

```
123456789123456789123456789 + 1 = 123456789123456789123456790
```

For this task, you should define the **AddOne()** function.

The **AddOne()** function takes two inputs: the first is a string (i.e. an array of characters) which represents an arbitrarily large number. The second is also a string into which you should store the output of the function. The output is a string which represents the number one larger than the input number.

Function prototype declaration:

```
void AddOne(char *input, char *output)
```

Assumptions:

You can assume that the input string represents a valid positive integer (greater than or equal to 1).
You **cannot** assume that the length of the output string will equal the length of the input string - for example, this will clearly not be the case if all of the characters in the input string are equal to ‘9’.

Example:

```
AddOne("12345", output);
printf("Result = %s\n", output);

AddOne("9999999999999", output);
printf("Result = %s\n", output);

AddOne("1999999999999", output);
printf("Result = %s\n", output);
```

Expected output:

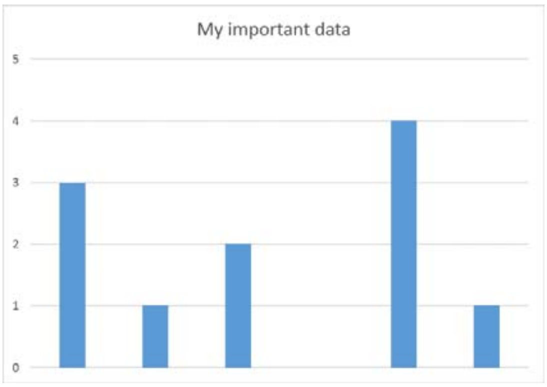
```
Result = 12346
Result = 10000000000000
Result = 2000000000000
```

Task Nine: “Textual histogram”	(10 marks)

Let’s say that we have the following 6 data values representing frequencies of some measurement:

3, 1, 2, 0, 4, 1

and we now would like to plot these on a histogram. We could do this easily using many graphical plotting programs such as Excel:



We could also represent the same data using a textual representation, where the bars are represented by “X” characters:

```
    X
X   X
X X X
XXX XX
```

And to make this look a little nicer, we could surround the bars with a border of ‘*’ characters:

```
********
*    X *
*X   X *
*X X X *
*XXX XX*
********
```

Define a function called **Histogram()** that takes an array of integers representing the data to be plotted, and generates a string (representing the histogram) in precisely the format described above. Please take note of the following:

- each line of text in the string ends with a new line (‘\n’) *except* for the very last line
- there must be *no extra space characters* anywhere at the beginning or end of a line

The **Histogram()** function should take three input values. The second and third input values represent the data to be plotted. This is stored as an array of integers, and the number of elements in the array. The first input to the function is the string into which you should store the resulting histogram.

Function prototype declaration:

```
void Histogram(char *result, int *values, int numValues);
```

Assumptions:

You can assume the input array will consist of *at least* one value greater than 0.

Example:

```
int values1[10] = {1, 0, 3, 1, 2, 4, 5, 6, 2, 2};
int values2[3] = {1, 0, 1};
char formatted[MAX_ARRAY_SIZE];
char example[MAX_ARRAY_SIZE] = "*****\n*X X*\n*****";

Histogram(formatted, values1, 10);
printf("%s\n\n", formatted);

Histogram(formatted, values2, 3);
printf("%s\n", formatted);
if (strcmp(example, formatted) == 0) {
    printf("This matches EXACTLY and is correct");
}
```

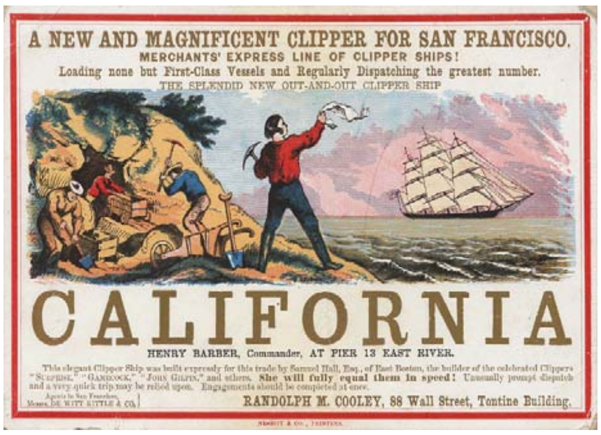
Expected output:

```
************
*       X  *
*      XX  *
*     XXX  *
*  X  XXX  *
*  X XXXXXX*
*X XXXXXXXX*
************

*****
*X X*
*****
This matches EXACTLY and is correct
```

Task Ten: “Gold Rush!” (8 marks)

The year is 1849 and you have just arrived in California to make your fortune in the gold rush. You have your pick axe, your kerosene lamp, a box of dynamite and your laptop. You need to stake your claim over the land that contains the largest gold deposits. You have access to a number of prospecting maps (which are grid based) which contain information about the quality of the gold deposits in the land. Fortunately, this data is digitized so you can easily read it into your program as a 2-dimensional array.


A NEW AND MAGNIFICENT CLIPPER FOR SAN FRANCISCO.
MERCHANTS' EXPRESS LINE OF CLIPPER SHIPS!
Loading none but First-Class Vessels and Regularly Dispatching the greatest number.
THE SPLENDID NEW OUT-AND-OUT CLIPPER SHIP
CALIFORNIA
HENRY BARBER, Commander, AT PIER 13 EAST RIVER.
She will fully equal them in speed!
RANDOLPH M. COOLEY, 88 Wall Street, Tontine Building.

An example of a prospecting map is the following 15 x 15 map:

```
1 2 2 0 0 0 0 0 0 0 0 0 0 0 0
0 4 3 0 0 0 0 9 9 8 0 0 0 0 0
0 2 0 3 3 0 0 9 9 0 0 0 0 0 0
0 0 0 0 0 4 6 9 9 6 0 0 0 0 0
0 0 0 0 0 0 9 0 8 0 0 6 0 0 0
0 0 9 9 9 9 0 0 0 0 7 7 8 8 0
0 0 9 9 9 9 0 0 0 0 0 7 0 0 0
0 0 9 9 9 9 0 1 1 1 2 2 2 2 2
0 0 0 9 9 0 0 0 0 0 0 0 0 3 0
0 0 0 4 4 0 0 0 0 0 0 0 5 6 0
0 0 0 0 9 9 9 0 0 9 0 0 0 5 0
0 0 1 2 9 9 9 0 0 0 9 0 0 4 2
0 0 0 0 9 9 9 0 0 0 9 9 9 0 0
9 9 0 0 0 0 1 0 0 0 0 9 0 0 0
9 0 0 0 0 0 2 2 0 0 0 0 0 0 0
```

The values (which are digits between 0 and 9) in each cell of the map represent the type of minerals present in the corresponding locations. The value 0 indicates that there are no deposits of any value at the location, and values between 1 and 8 indicate various kinds of common minerals which are of little interest to you. The value you are interested in is the value 9 as this represents the presence of gold! To help you quickly determine which maps are most promising, you want to write a program to compute how much gold (i.e. cells with a value of 9) is present in a given map.

You also want to compute one other value - that is, how much *pure gold* is present in the map. A cell contains pure gold if it contains gold <u>and</u> if *all eight* of the cells directly adjacent to it (in any direction: up, down, left, right or any diagonal) *also* contain gold. By this definition, cells on the border of the map cannot contain pure gold.

The 15 x 15 map shown above contains 4 separate regions of gold (which are not connected to each other). The table below illustrates these four regions. In the column labelled "Prospecting map", the map is shown and cells containing *gold* are highlighted in bold. Cells containing *pure gold* are also underlined. In the column labelled "Region", the four non-connected regions are shown on separate rows. The column labelled "Gold" shows how much gold is contained in the region and the column labelled "Pure gold" shows the amount of pure gold

Prospecting map	***Region***	***Gold***	***Pure gold***
	9 9	21	2
	9 9		
1 2 2 0 0 0 0 0 0 0 0 0 0 0 0	9 9		
0 4 3 0 0 0 0 **9** **9** 8 0 0 0 0 0	9		
0 2 0 3 3 0 0 **9** **9** 0 0 0 0 0 0	9 9 9 9		
0 0 0 0 0 4 6 **9** **9** 6 0 0 0 0 0	9 9 9 9		
0 0 0 0 0 0 **9** 0 8 0 0 6 0 0 0	9 9 9 9		
0 0 **9** **9** **9** **9** 0 0 0 0 7 7 8 8 0	9 9		
0 0 **9** $\underline{\mathbf{9}}$ $\underline{\mathbf{9}}$ **9** 0 0 0 0 0 7 0 0 0	9 9 9	9	1
0 0 **9** **9** **9** **9** 0 1 1 1 2 2 2 2 2	9 9 9		
0 0 0 **9** **9** 0 0 0 0 0 0 0 0 3 0	9 9 9		
0 0 0 4 4 0 0 0 0 0 0 0 5 6 0	9	6	0
0 0 0 0 **9** **9** **9** 0 0 **9** 0 0 0 5 0	9		
0 0 1 2 **9** $\underline{\mathbf{9}}$ **9** 0 0 0 **9** 0 0 4 2	9 9 9		
0 0 0 0 **9** **9** **9** 0 0 0 **9** **9** **9** 0 0	9		
9 **9** 0 0 0 0 1 0 0 0 0 **9** 0 0 0	9 9	3	0
9 0 0 0 0 0 2 2 0 0 0 0 0 0 0	9		

Define a function called **GoldRush()** that takes five inputs:

- A one-dimensional array of integers, called **results**, into which you will store the results computed by your function
- An integer, called **rows**, which indicates how many rows are present in the map
- An integer, called **cols**, which indicates how many columns are present in the map
- A two-dimensional array of integers, called **map**, which contains the map data
- An integer called **bonus** which is only used if you have attempted the bonus tasks described later (for this task, this input will always have the value 0)

This function should compute the total amount of gold in the map, and the total amount of pure gold in the map. Using the example map shown above, the total amount of gold is 39 and the total amount of pure gold is 3.

The function is void (i.e. it does not return a value). To return the results that you have computed, you should use the **results** array (the first input to the function). You should store the total amount of *gold* in **results[0]** and the amount of *pure gold* in **results[1]**.

Function prototype declaration:

```
void GoldRush(int *results, int rows, int cols,
            int map[MAX_MAP_SIZE][MAX_MAP_SIZE], int bonus);
```

Assumptions:

You can assume the map will have at least two rows and two columns, and will have no more than MAX_MAP_SIZE rows and MAX_MAP_SIZE columns

Example:

```
int i, j;
int results[MAX_ARRAY_SIZE];
int map[MAX_MAP_SIZE][MAX_MAP_SIZE] = {
     {1,2,2,0,0,0,0,0,0,0,0,0,0,0,0},
     {0,4,3,0,0,0,0,9,9,8,0,0,0,0,0},
     {0,2,0,3,3,0,0,9,9,0,0,0,0,0,0},
     {0,0,0,0,0,4,6,9,9,6,0,0,0,0,0},
     {0,0,0,0,0,0,9,0,8,0,0,6,0,0,0},
     {0,0,9,9,9,9,0,0,0,0,7,7,8,8,0},
     {0,0,9,9,9,9,0,0,0,0,0,7,0,0,0},
     {0,0,9,9,9,9,0,1,1,1,2,2,2,2,2},
     {0,0,0,9,9,0,0,0,0,0,0,0,0,3,0},
     {0,0,0,4,4,0,0,0,0,0,0,0,5,6,0},
     {0,0,0,0,9,9,9,0,0,9,0,0,0,5,0},
     {0,0,1,2,9,9,9,0,0,0,9,0,0,4,2},
     {0,0,0,0,9,9,9,0,0,0,9,9,9,0,0},
     {9,9,0,0,0,0,1,0,0,0,0,9,0,0,0},
     {9,0,0,0,0,0,2,2,0,0,0,0,0,0,0}
};

GoldRush(results, 15, 15, map, 0);
printf("Searching for gold!\n");
printf("  Total gold = %d\n", results[0]);
printf("  Pure gold  = %d\n", results[1]);
```

Expected output:

```
Searching for gold!
  Total gold = 39
  Pure gold  = 3
```

Bonus Task I: "Gold regions" (1 mark)

The last input to the **GoldRush()** function is an integer called **bonus**. For Task Ten, described above, this input will always be 0. For the first bonus task, the value of this input will be set to 1.

For the first bonus task, you should count the *total amount of gold* present in *each* non-connected region of gold on the map. You should report your results by storing them in the **results** array, in *decreasing* order (so the largest region of gold is reported first). Once you have stored the results for each region, in consecutive elements of the array, you must then store the value 0 in the next element of the array to indicate there are no more results.

Example:

```
int i, j;
int results[MAX_ARRAY_SIZE];
int map[MAX_MAP_SIZE][MAX_MAP_SIZE] = {
     {1,2,2,0,0,0,0,0,0,0,0,0,0,0,0},
     {0,4,3,0,0,0,0,9,9,8,0,0,0,0,0},
     {0,2,0,3,3,0,0,9,9,0,0,0,0,0,0},
     {0,0,0,0,0,4,6,9,9,6,0,0,0,0,0},
     {0,0,0,0,0,0,9,0,8,0,0,6,0,0,0},
     {0,0,9,9,9,9,0,0,0,0,7,7,8,8,0},
     {0,0,9,9,9,9,0,0,0,0,0,7,0,0,0},
     {0,0,9,9,9,9,0,1,1,1,2,2,2,2,2},
     {0,0,0,9,9,0,0,0,0,0,0,0,0,3,0},
     {0,0,0,4,4,0,0,0,0,0,0,0,5,6,0},
     {0,0,0,0,9,9,9,0,0,9,0,0,0,5,0},
     {0,0,1,2,9,9,9,0,0,0,9,0,0,4,2},
     {0,0,0,0,9,9,9,0,0,0,9,9,9,0,0},
     {9,9,0,0,0,0,1,0,0,0,0,9,0,0,0},
     {9,0,0,0,0,0,2,2,0,0,0,0,0,0,0}
};

GoldRush(results, 15, 15, map, 1);
printf("Searching for gold!\n");
printf("  Total gold regions = ");
i = 0;
while (results[i] != 0) {
     printf("%d ", results[i]);
     i++;
}
```

Expected output:

```
Searching for gold!
  Total gold regions = 21 9 6 3
```

Bonus Task II: "Pure gold" (1 mark)

The last input to the **GoldRush()** function is an integer called **bonus**. For the second bonus task, the value of this input will be set to 2.

For the second bonus task, you should count the *total amount of pure gold* present in *each* non-connected region of gold on the map. You should report your results by storing them in the **results** array, in *decreasing* order (so the largest region of pure gold is reported first). Once you have stored the results for each region, in consecutive elements of the array, you must then store the value 0 in the next element of the array to indicate there are no more results.

Example:

```
int i, j;
int results[MAX_ARRAY_SIZE];
int map[MAX_MAP_SIZE][MAX_MAP_SIZE] = {
     {1,2,2,0,0,0,0,0,0,0,0,0,0,0,0},
     {0,4,3,0,0,0,0,9,9,8,0,0,0,0,0},
     {0,2,0,3,3,0,0,9,9,0,0,0,0,0,0},
     {0,0,0,0,0,4,6,9,9,6,0,0,0,0,0},
     {0,0,0,0,0,0,9,0,8,0,0,6,0,0,0},
     {0,0,9,9,9,9,0,0,0,0,7,7,8,8,0},
     {0,0,9,9,9,9,0,0,0,0,0,7,0,0,0},
     {0,0,9,9,9,9,0,1,1,1,2,2,2,2,2},
     {0,0,0,9,9,0,0,0,0,0,0,0,0,3,0},
     {0,0,0,4,4,0,0,0,0,0,0,0,5,6,0},
     {0,0,0,0,9,9,9,0,0,9,0,0,0,5,0},
     {0,0,1,2,9,9,9,0,0,0,9,0,0,4,2},
     {0,0,0,0,9,9,9,0,0,0,9,9,9,0,0},
     {9,9,0,0,0,0,1,0,0,0,0,9,0,0,0},
     {9,0,0,0,0,0,2,2,0,0,0,0,0,0,0}
};

GoldRush(results, 15, 15, map, 2);
printf("Searching for gold!\n");
printf("  Total pure gold regions = ");
i = 0;
while (results[i] != 0) {
     printf("%d ", results[i]);
     i++;
}
```

Expected output:

```
Searching for gold!
  Total pure gold regions = 2 1
```

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 project.c and test_program.c source files and the unedited project.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!

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
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.

Paul Denny
October 2017