ENGGEN 131 – Semester Two, 2016

# C Programming Project

**Deadline**: 11:59pm, Monday 17th 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. If you like, you may 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 will get partial credit for tasks solved partially). Each function is worth 10% of the marks for the project.

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. 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, or even suspension from Engineering) are simply not worth the risk.

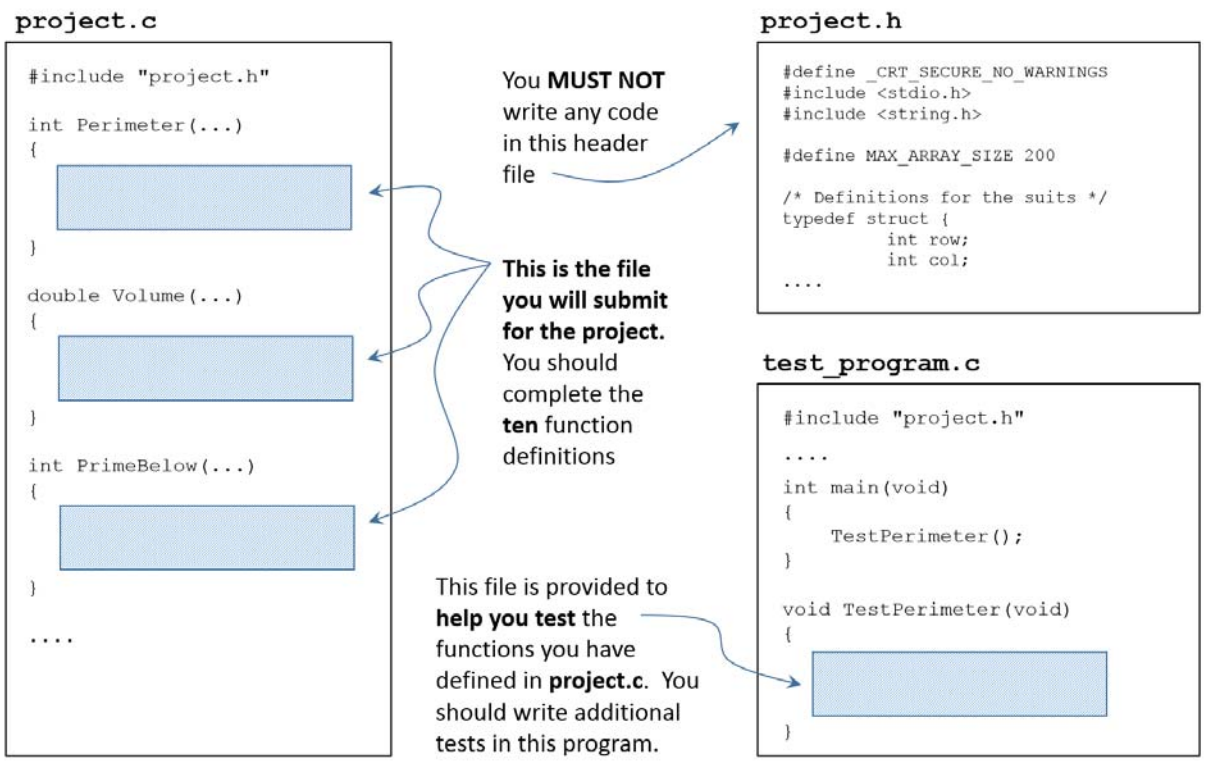
| Acceptable | Unacceptable |
|---|---|
| • Describing problems you are having to someone else, either in person or on Piazza, without revealing any code you have written<br>• Asking for advice on how to solve a problem, where the advice received is general in nature and does not include any code<br>• 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)<br>• 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<br>• Writing code on paper or at a computer, and sharing that code in any way with anyone else<br>• Giving or receiving any amount of code from anyone else in any form<br>• Code sharing = NO |

## Understanding the project files

There are three files that you will be working with. The most important of these three files is **project.c**. This is the source file that <u>you will submit for marking</u>. 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 help you test the function definitions you write in **project.c**

The diagram below illustrates the relationship between these three source files.



project.c

```
#include "project.h"

int Perimeter(...)
{

}

double Volume(...)
{

}

int PrimeBelow(...)
{

}

....
```

project.h

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

#define MAX_ARRAY_SIZE 200

/* Definitions for the suits */
typedef struct {
        int row;
        int col;
....
```

test_program.c

```
#include "project.h"
....
int main(void)
{
    TestPerimeter();
}

void TestPerimeter(void)
{

}
```

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.

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. One very simple example has been included here 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 Perimeter(int width, int height);
double Volume(int radius);
int PrimeBelow(int upper);
void Abbreviate(char* word);
void Strikeout(char *hide, char *phrase);
int KthLargest(int k, int *values, int numValues);
Rectangle BoundingRectangle(Rectangle r1, Rectangle r2);
void Codeword(char *result);
int TallestVine(int seedA, int seedB, int days);
int MakeMove(int cave[10][10], char move);
```

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 *.c
```

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

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

```
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!

Testing Perimeter()...
  - FAIL (10, 25)
  - FAIL (20, -1)

Testing Volume()...
  - FAIL (10)
  - FAIL (-5)

Testing PrimeBelow()...
  - FAIL (10)
  - FAIL (47)

Testing Abbreviate()...
  - FAIL (wordA)
  - FAIL (wordB)

Testing Strikeout()...
  - FAIL (wordA)
  - FAIL (wordB)

Testing KthLargest()...
  - FAIL (valuesA)
  - FAIL (valuesB)

Testing BoundingRectangle()...
  - FAIL (c1)
  - FAIL (c2)

Testing Codeword()...
  - This function cannot be tested automatically

Testing TallestVine()...
  - FAIL (501, 491, 8)
  - FAIL (20, 1000, 73)

Testing MakeMove()...
  - FAIL (MakeMove)
```

Notice that initially, because all of the function definitions are incorrect, all of the tests fail (hence the output “FAIL”).

When you correct the function definitions, the output from this program should change to PASS for each test.

## 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 program to crash or hang regardless of the input. 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

Your submitted source file, **project.c**, 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.

## 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 **project.h**:

```
/* Any arrays that are passed as inputs to functions are */
/* guaranteed to be no larger than this constant */
#define MAX_ARRAY_SIZE 200
```

You must not declare any arrays to be larger than this constant. For example, you could declare a temporary array (inside one of your function definitions) like this:

```
int temp[MAX_ARRAY_SIZE];
```

## 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 solving (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:** “Around the outside”

Define a function called **Perimeter()** that is passed two integer inputs representing the *width* and *height* of a field. The function must return the *perimeter* of the field, that is, the length of the outside boundary. The output must be an integer. If either input value is negative, the function should return 0.

Function prototype declaration:

```
int Perimeter(int width, int height)
```

Assumptions:

You cannot assume the inputs are positive. If either input is negative, you must return 0.

Example:

```
printf("Perimeter = %d\n", Perimeter(10, 25));
printf("Perimeter = %d\n", Perimeter(0, 100));
printf("Perimeter = %d\n", Perimeter(20, -1));
```

Expected output:

```
Perimeter = 70
Perimeter = 200
Perimeter = 0
```

## **Task Two:** “Sphere”

Define a function called **Volume()** that is passed one integer input representing the *radius* of a sphere. The function must return the *volume* of the sphere. The following formula gives the volume of a sphere in terms of its radius:

$$V = \frac{4}{3}\pi r^3$$

The output of the function must be a double. If the input value is negative, the function should return 0.0.

Function prototype declaration:

```
double Volume(int radius)
```

Assumptions:

You cannot assume the input is positive. If the input is negative, you must return 0.0. The math library is not available to you, so you cannot use the pow() function.

You may use the provided value for PI:

```
double PI = 3.141592654;
```

Example:

```
printf("Volume = %f\n", Volume(10));
printf("Volume = %f\n", Volume(123));
printf("Volume = %f\n", Volume(-5));
```

Expected output:

```
Volume = 4188.790205
Volume = 7794781.463028
Volume = 0.000000
```

**Task Three:** “Primes”

Define a function called **PrimeBelow()** that is passed one integer input representing an *upper bound*. The function must return the *largest* prime number that is *less than* this upper bound. As the smallest prime number is 2, if the input to the function is 2 or less, the function should return -1 to indicate an error.

Function prototype declaration:

```
int PrimeBelow(int upper)
```

Assumptions:

> You cannot assume the input is greater than 2. If the input to the function is 2 or less, then you must return -1.

Example:

```
printf("Prime = %d\n", PrimeBelow(10));
printf("Prime = %d\n", PrimeBelow(47));
printf("Prime = %d\n", PrimeBelow(2));
```

Expected output:

```
Prime = 7
Prime = 43
Prime = -1
```

**Task Four:** "No vowels"

Define a function called **Abbreviate()** that is passed one string input. The function should *modify* this input string by removing all of its vowels.

Function prototype declaration:

```
void Abbreviate(char* word);
```

Assumptions:

You can assume that all of the characters in the input string are in lower case (although there may be spaces, digits and punctuation characters, there will not be any upper case characters). There are only 5 vowels: a, e, i, o and u.

Example:

```
char wordA[MAX_ARRAY_SIZE] = "facetiously";
char wordB[MAX_ARRAY_SIZE] = "sequoia";
char wordC[MAX_ARRAY_SIZE] = "wry";
char phrase[MAX_ARRAY_SIZE] = "meet you at the movies
                                                   tonight";

Abbreviate(wordA);
Abbreviate(wordB);
Abbreviate(wordC);
Abbreviate(phrase);

printf("%s\n", wordA);
printf("%s\n", wordB);
printf("%s\n", wordC);
printf("%s\n", phrase);
```

Expected output:

```
fctsly
sq
wry
mt y t th mvs tnght
```

**Task Five:** “Censor”

Define a function called **Strikeout()** that is passed two string inputs. The function should *modify* the second input string by replacing all occurrences of the first input string that it contains by the asterisk character (*).

Function prototype declaration:

```
void Strikeout(char *hide, char *phrase);
```

Assumptions:

> You can assume that the length of the first input string is less than or equal to the length of the second input string.

Example:

```
char wordA[MAX_ARRAY_SIZE] = "nail";
char phraseA[MAX_ARRAY_SIZE] = "get the nail out";

char wordB[MAX_ARRAY_SIZE] = "hello";
char phraseB[MAX_ARRAY_SIZE] = "hello world";

char wordC[MAX_ARRAY_SIZE] = "cat";
char phraseC[MAX_ARRAY_SIZE] = "cat dog cat cat ant cat";

char wordD[MAX_ARRAY_SIZE] = "finish";
char phraseD[MAX_ARRAY_SIZE] = "finish";

Strikeout(wordA, phraseA);
Strikeout(wordB, phraseB);
Strikeout(wordC, phraseC);
Strikeout(wordD, phraseD);

printf("%s\n", phraseA);
printf("%s\n", phraseB);
printf("%s\n", phraseC);
printf("%s\n", phraseD);
```

Expected output:

```
get the **** out
***** world
*** dog *** *** ant ***
******
```

## **Task Six:** “Orderly”

Define a function called **KthLargest()** that is passed three inputs - an integer k, an array of integers, and the number of values in the array. The function should return the k$^{th}$ largest value in the array. If there is no k$^{th}$ largest value in the array (for example, if the value k is larger than the number of distinct values in the array), then the function should return -1 to indicate this.

Function prototype declaration:

```
int KthLargest(int k, int *values, int numValues);
```

Assumptions:

You can assume that the input array contains at least one element. You can also assume that the first input, k, is greater than or equal to 1.

Example:

```
int valuesA[12] = {4, 3, 7, 6, 3, 1, 4, 2, 6, 7, 6, 7};
int valuesB[5] = {3, 3, 999, 3, 3};

printf("A1 = %d\n", KthLargest(1, valuesA, 12));
printf("A2 = %d\n", KthLargest(2, valuesA, 12));
printf("A3 = %d\n", KthLargest(3, valuesA, 12));

printf("B1 = %d\n", KthLargest(1, valuesB, 5));
printf("B2 = %d\n", KthLargest(2, valuesB, 5));
printf("B3 = %d\n", KthLargest(3, valuesB, 5));
```

Expected output:

```
A1 = 7
A2 = 6
A3 = 4
B1 = 999
B2 = 3
B3 = -1
```

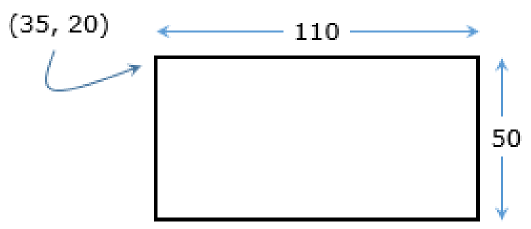
**Task Seven:** "Bounding"

The `Rectangle` structure is defined as follows:

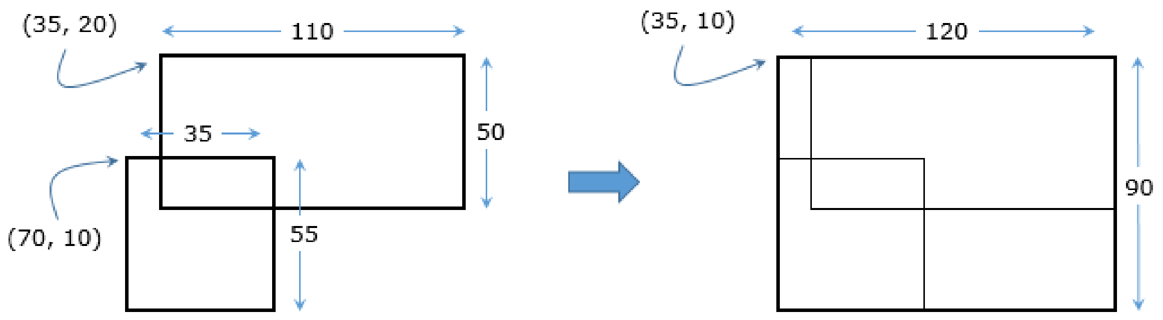
```
typedef struct {
     int row;
     int col;
     int width;
     int height;
} Rectangle;
```

The first two fields represent the row (down from the top) and column (across from the left) of the top left corner of a rectangle, and the latter two fields represent the width and height of the rectangle. For example, the code on the left below would initialise the `Rectangle` shown on the right:

```
Rectangle a;

a.row = 35;
a.col = 20;
a.width = 110;
a.height = 50;
```



For this exercise, you need to define a function called **BoundingRectangle()** which takes two `Rectangle`s as inputs, and returns a third `Rectangle` which is the *smallest possible* `Rectangle` that *entirely bounds* (i.e. encloses) the two input `Rectangle`s. The diagram below shows two `Rectangle`s (their coordinates are shown on the left hand side of the diagram), and on the right hand side of the diagram the bounding `Rectangle` is shown.



Function prototype declaration:

```
Rectangle BoundingRectangle(Rectangle r1, Rectangle r2);
```

Assumptions:

You can assume that all fields of a `Rectangle` structure, that is the row, col, width and height, are all greater than 0.

Example:

```
Rectangle a, b, c, tiny;

a.row = 35;
a.col = 20;
a.width = 110;
a.height = 50;

b.row = 70;
b.col = 10;
b.width = 35;
b.height = 55;

tiny.row = 45;
tiny.col = 45;
tiny.width = 5;
tiny.height = 5;

c = BoundingRectangle(a, b);
printf("%d %d %d %d \n", c.row, c.col, c.width, c.height);

c = BoundingRectangle(a, tiny);
printf("%d %d %d %d \n", c.row, c.col, c.width, c.height);
```

Expected output:

```
35 10 120 90
35 20 110 50
```

## **Task Eight:** “Secret code word”

This task is a little different to the rest. Please pay <u>very careful attention</u> to these instructions.

Define a function called **Codeword()** that is passed a string (i.e. an array of characters) as input. The function must modify the string by setting it to a particular word (a very simple way of achieving this is by calling the standard strcpy() function as shown in the example below). Please note, this task requires <u>just one line of code</u> (as shown in the example below). The key is for you to work out *exactly what word* to use. For example, if you think the correct word is “mycodeword” then you can solve this task with the following function definition:

```
void Codeword(char *result)
{
     strcpy(result, "mycodeword");
}
```

Please make sure that you do not add any space characters to the start or end of this word.

<u>Function prototype declaration:</u>

```
void Codeword(char *result);
```

<u>Assumptions:</u>

The word is a real word in the English language, is all lower case, and contains no spaces.

<u>How to discover the secret word</u>

To discover the secret word, you need to download an image file from a secret webserver. It’s so secret, that no one knows about it (shhhh!) Here is the link to the secret webserver:

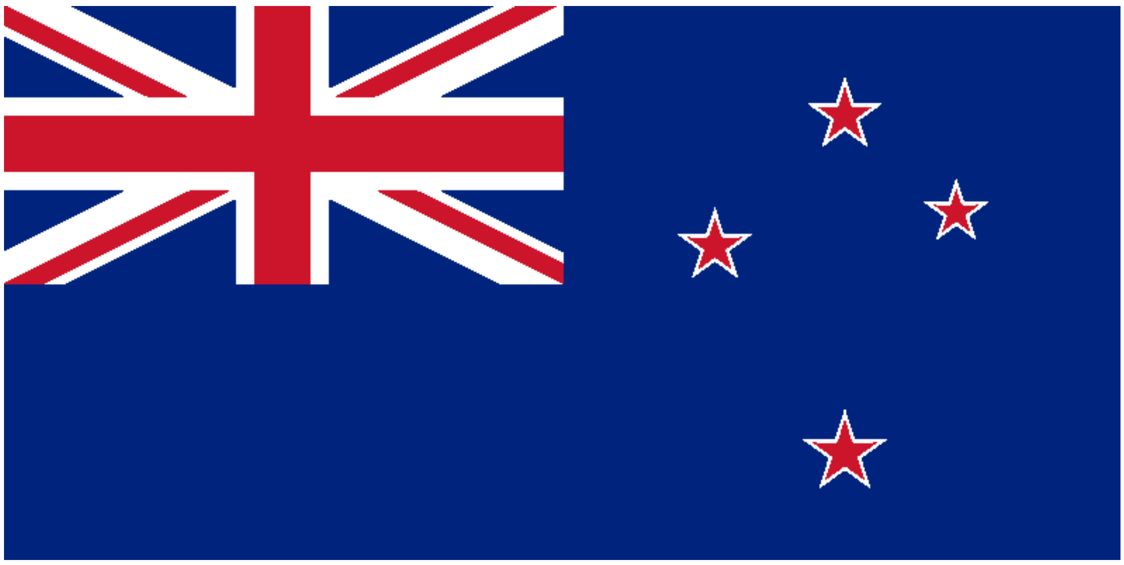
http://redsox.tcs.auckland.ac.nz/CWS/CourseWorkService.svc/cwm?cid=Steganography&type=bmp

This link also appears on Canvas.

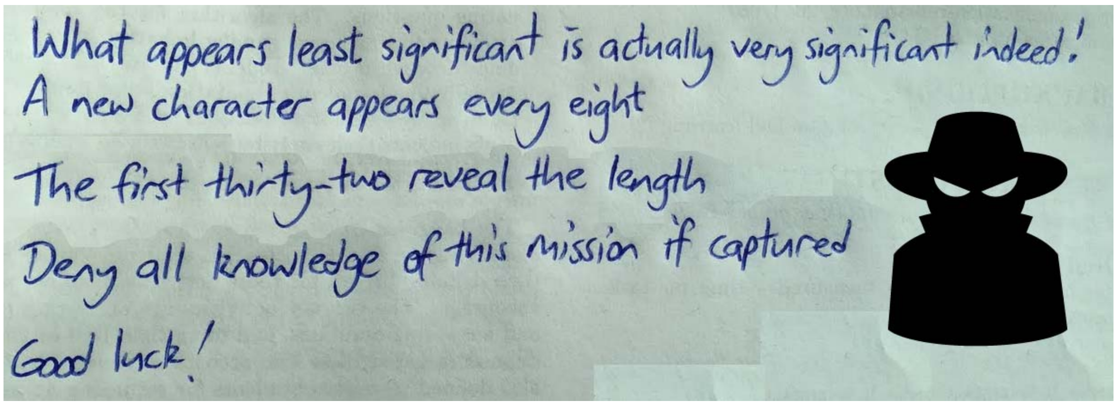
Save the image file to disk. Make sure that your username appears in the file. For example, if your University of Auckland username is “abcd001” then the secret image file you download will be named: “Steganography-abcd001.bmp”.

<u>The image looks like a flag, where is the secret word?</u>

At first glance, the image looks like the New Zealand flag:



The secret word is actually hidden inside this image file. You meet a shady spy-looking character who hands you the following piece of paper:

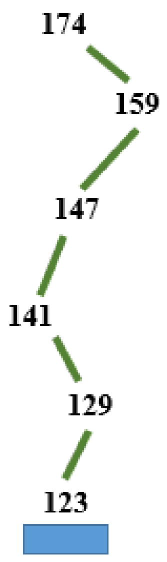


What appears least significant is actually very significant indeed!
A new character appears every eight
The first thirty-two reveal the length
Deny all knowledge of this mission if captured
Good luck!

These appear to be some cryptic hints. Good luck!

**Task Nine:** "Digital vines"

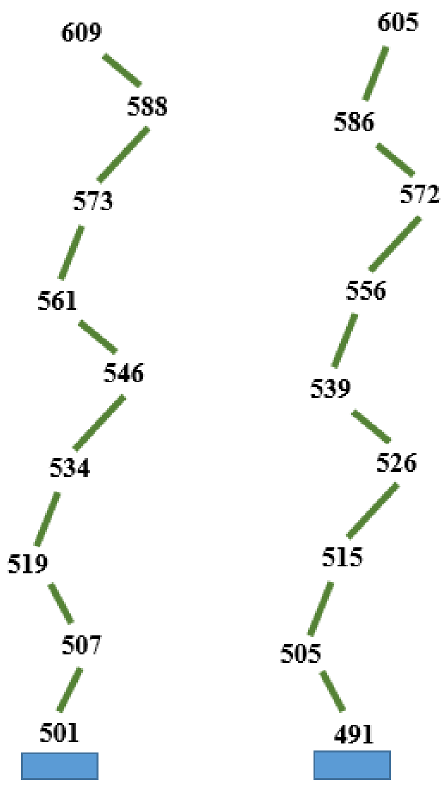
A *digital vine* begins as a seed (a positive integer) and grows each day by an amount equal to the sum of its digits. For example, the diagram shown on the right illustrates the growth of the digital vine that begins with the seed 123 over 5 days.



In this case, the vine reaches a height of 174 after 5 days.

When two digital vines grow side by side over a period of days, one of two outcomes is possible:

- If the two vines contain entirely different values, then the two vines grow independently and do not join
- If one vine generates the same value that has already been generated in the other vine, then the two vines immediately join and fuse at that value, no further growth occurs, and all growth above that value disappears

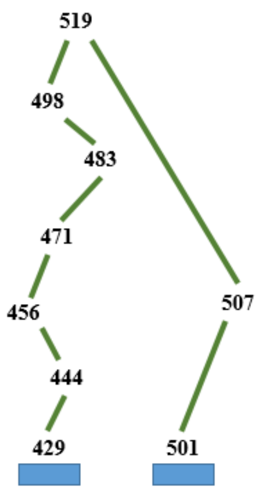


For example, in the diagram on the left, two vines grow side by side over a period of 8 days. The vine on the left has the seed value 501 and the vine on the right has the seed value 491. In this case, the vines do not join because they consist of entirely different values over the period of growth.

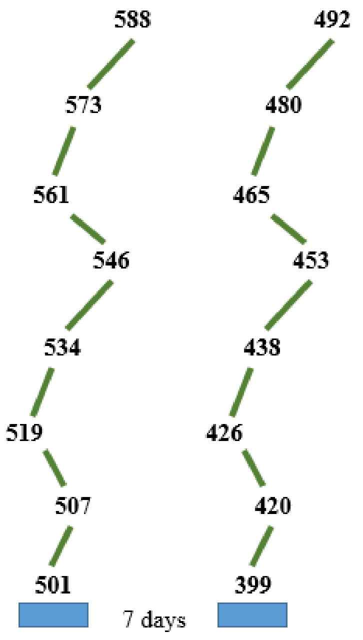
In this case, the tallest vine (the one on the left, starting with a seed value of 501) reaches a height of 609.

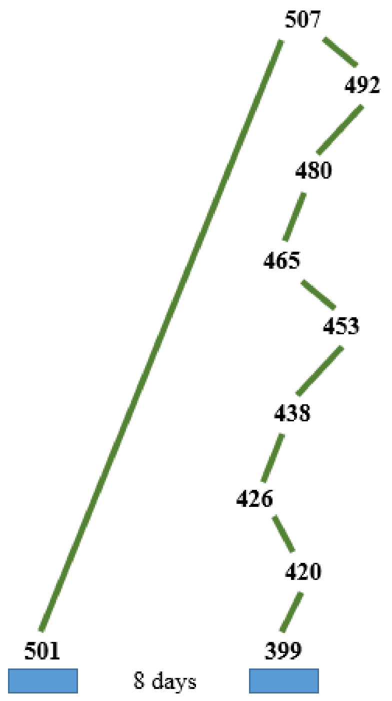
The next example shows the result of two vines growing over the same period of time (8 days). The vine on the left has a seed value of 429 and the vine on the right has a seed value of 501. This time the vines join at a value (the value 519). Even though one vine reaches this value sooner (after 2 days) than the other vine (after 6 days), the point at which they join signals the end of the growth of the two vines.

The point at which they have joined (in this case, 519) therefore represents the tallest point that the vines will grow.



Note that the length of time the vines are given to grow is critical. Consider two vines growing side by side with seed values of 501 and 399. If the vines are given 7 days to grow, then the vines do not join and the tallest point they will reach is 588 (this is reached by the vine on the left hand side which starts with the seed value of 501). This is illustrated in the diagram on the left below. On the other hand, if the vines are given 8 days to grow, then they do actually join. In this case, the tallest point they will reach is the value at which they join - in this case 507. This is illustrated in the diagram on the right below.





Define a function called **TallestVine()** that takes two input seeds and a period of time (in days) as input and calculates the tallest point the two vines will reach after growing for the specified period of time.

Function prototype declaration:

```
int TallestVine(int seedA, int seedB, int days);
```

Assumptions:

You can assume that both seed values are positive integers and that the number of days is greater than 0.

Example:

```
printf("Here are the examples from the specification:\n");
printf("Tallest vine = %d\n", TallestVine(501, 491, 8));
printf("Tallest vine = %d\n", TallestVine(429, 501, 8));
printf("Tallest vine = %d\n", TallestVine(501, 399, 7));
printf("Tallest vine = %d\n", TallestVine(501, 399, 8));

printf("\nThese two vines only meet after 73 days:\n");
printf("Tallest vine = %d\n", TallestVine(20, 1000, 72));
printf("Tallest vine = %d\n", TallestVine(20, 1000, 73));

printf("\nThese two vines never join, after any time:\n");
printf("Tallest vine = %d\n", TallestVine(480, 481, 5000));
```

Expected output:

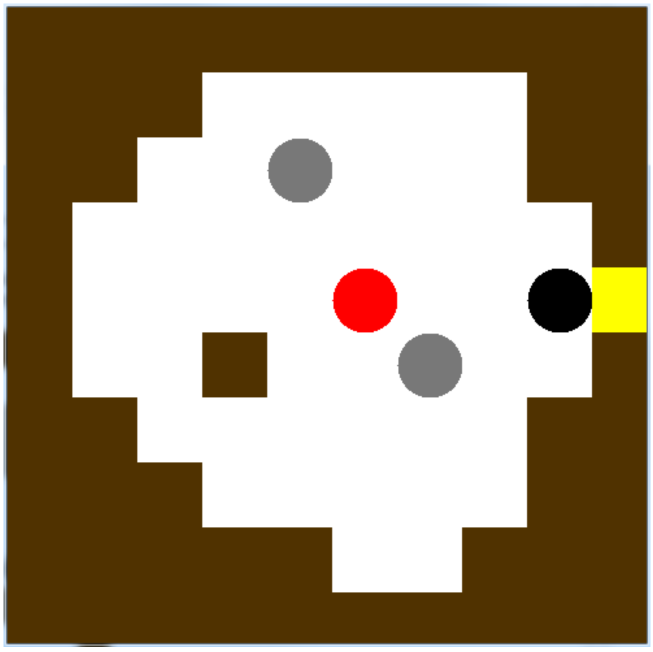
```
Here are the examples from the specification:
Tallest vine = 609
Tallest vine = 519
Tallest vine = 588
Tallest vine = 507

These two vines only meet after 73 days:
Tallest vine = 1972
Tallest vine = 1003

These two vines never join, after any time:
Tallest vine = 107598
```

Limitation of arrays for this task:

It may be advisable to avoid relying on arrays in your solution. If you are using arrays, you are limited to creating arrays with a maximum length of MAX_ARRAY_SIZE (this is the constant defined in project.h, which is set to 200). As you can see in the last example above, the number of days of growth may exceed this constant.

**Task Ten:** "Cave escape"

*Cave escape* is a 2-dimensional puzzle game where the goal is to escape from a cave by pushing boulders into holes that block the exit. In the diagram shown below, the caver (trying to escape the cave) is represented by the red circle. The gray coloured circles are the boulders that the caver can push. The black circle is the hole which is currently blocking the yellow exit, and into which the boulders can be pushed.



To push a boulder, the caver must position themselves next to the boulder and can only push the boulder in the direction of their movement. When the caver pushes a boulder, the boulder will continue to roll in a straight line until it hits an obstacle (either part of the cave or another boulder).

To escape the cave, the caver must push a boulder into the hole which will fill it in allowing them to access the yellow exit.

Your job is to complete the implementation of the game-logic for the movement. You must write a function called **MakeMove()** which is given two inputs: a 2-dimensional array of integers representing the cave, and the position in which the caver is to move. The 10x10 2-dimensional array that represents the cave illustrated in the diagram above is shown below:

```
int cave[10][10] = {
     {1, 1, 1, 1, 1, 1, 1, 1, 1, 1},
     {1, 1, 1, 0, 0, 0, 0, 0, 1, 1},
     {1, 1, 0, 0, 2, 0, 0, 0, 1, 1},
     {1, 0, 0, 0, 0, 0, 0, 0, 0, 1},
     {1, 0, 0, 0, 0, 3, 0, 0, 4, 5},
     {1, 0, 0, 1, 0, 0, 2, 0, 0, 1},
     {1, 1, 0, 0, 0, 0, 0, 0, 1, 1},
     {1, 1, 1, 0, 0, 0, 0, 0, 1, 1},
     {1, 1, 1, 1, 1, 0, 0, 1, 1, 1},
     {1, 1, 1, 1, 1, 1, 1, 1, 1, 1},
};
```

*Note that the values in the array have the following meanings:*

0 = empty space
1 = wall of cave
2 = boulder
3 = caver
4 = hole
5 = exit

The second input to the **MakeMove()** function is a character. This will have one of four possible values: 'w', 'a', 's' or 'd'. These characters represent the directions up, left, down and right respectively.

Rules:

- The caver can not move into or through the walls of the cave.
- When a boulder rolls into a hole, both the hole and the boulder disappear (i.e. the value 4 changes to 0 and the boulder disappears)
- To push a boulder, the caver must move onto the square containing the boulder, and the boulder must be able to roll away, in the same direction, from the square that it was in (i.e. if the caver moves “up” onto a square with a boulder, then the boulder must have at least some empty space above it in which to roll)
- A boulder, once pushed, will continue to move in a straight line until it hits an obstacle (either a hole, another boulder or a cave wall)
- You can assume that the boulder will never be pushed through the exit (in other words, you don’t need to handle the case that a boulder moves onto a square with the value 5)

Function prototype declaration:

```
int MakeMove(int cave[10][10], char move);
```

What you have to do:

You must implement the **MakeMove()** function such that for any input cave/array, and any given move, the array will be updated correctly to reflect that move.

Assumptions:

You can assume that all elements around the border of the two-dimensional array representing the cave (i.e. all elements in rows 0 and 9 and columns 0 and 9) will either contain the value 1 or the value 5.

Examples:

In the examples that follow, the cave is represented in ASCII with the following character mappings:

- 0 = empty space = ' '
- 1 = wall of cave = '#'
- 2 = boulder = 'O'
- 3 = caver = 'X'
- 4 = hole = '*'
- 5 = exit = ' ' (next to the hole)

Expected output:

The examples below illustrate how the cave would change following the described actions.

| Starting array (i.e. input) | Move | Ending array (i.e. output) |
|---|---|---|
| `##########` | | `##########` |
| `###     ##` | | `###     ##` |
| `##  O   ##` | | `##  O   ##` |
| `#        #` | | `#        #` |
| `#    X  * ` | ‘s’ = down | `#       * ` |
| `#  #  O  #` | | `#  # XO  #` |
| `##      ##` | | `##      ##` |
| `###     ##` | | `###     ##` |
| `#####  ###` | | `#####  ###` |
| `##########` | | `##########` |
| `##########` | | `##########` |
| `###     ##` | | `###   O ##` |
| `##  O   ##` | | `##  O   ##` |
| `#        #` | | `#        #` |
| `#       * ` | ‘w’ = up | `#       * ` |
| `#  #  O  #` | | `#  #  X  #` |
| `##    X ##` | | `##      ##` |
| `###     ##` | | `###     ##` |
| `#####  ###` | | `#####  ###` |
| `##########` | | `##########` |
| `##########` | | `##########` |
| `###     ##` | | `###     ##` |
| `##  O   ##` | | `##  O   ##` |
| `#        #` | | `#        #` |
| `#       * ` | ‘a’ = left | `#       * ` |
| `#  #  OX #` | | `#  #O X  #` |
| `##      ##` | | `##      ##` |
| `###     ##` | | `###     ##` |
| `#####  ###` | | `#####  ###` |
| `##########` | | `##########` |
| `##########` | | `##########` |
| `###     ##` | | `###     ##` |
| `##      ##` | | `##      ##` |
| `#        #` | | `#        #` |
| `#  XO   * ` | ‘d’ = right | `#   X     ` |
| `#  #O    #` | | `#  #O    #` |
| `##      ##` | | `##      ##` |
| `###     ##` | | `###     ##` |
| `#####  ###` | | `#####  ###` |
| `##########` | | `##########` |

Return value:

The **MakeMove()** function must return an integer.  Only one of the three values need be returned:

| Return value | Situation |
|---|---|
| 2 | the player exited the cave (i.e. moved onto the square marked 5, in this the exit is replaced by the value 3 to indicate the caver has reached the exit) |
| 1 | the player fell into the hole (i.e. moved onto the square marked 4, in this case the player disappears) |
| 0 | the player did not exit the cave or fall into a hole on this move |

Resources for Task Ten

You have been provided with a program called `caveescape.c` which you can use to help test your **MakeMove()** function.  All of the functionality for the game, except for the **MakeMove()** function, is provided to you in this program.

# 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 2015 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 2015 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 2015 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:<br><br>`cl /W4 *.c`<br><br>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, or that compiles but generates warnings.

## 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

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

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.

**ReferenCite**
The University's official academic referencing resource, featuring tutorials and guidelines on good referencing practices.

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

Paul Denny
*September 2016*