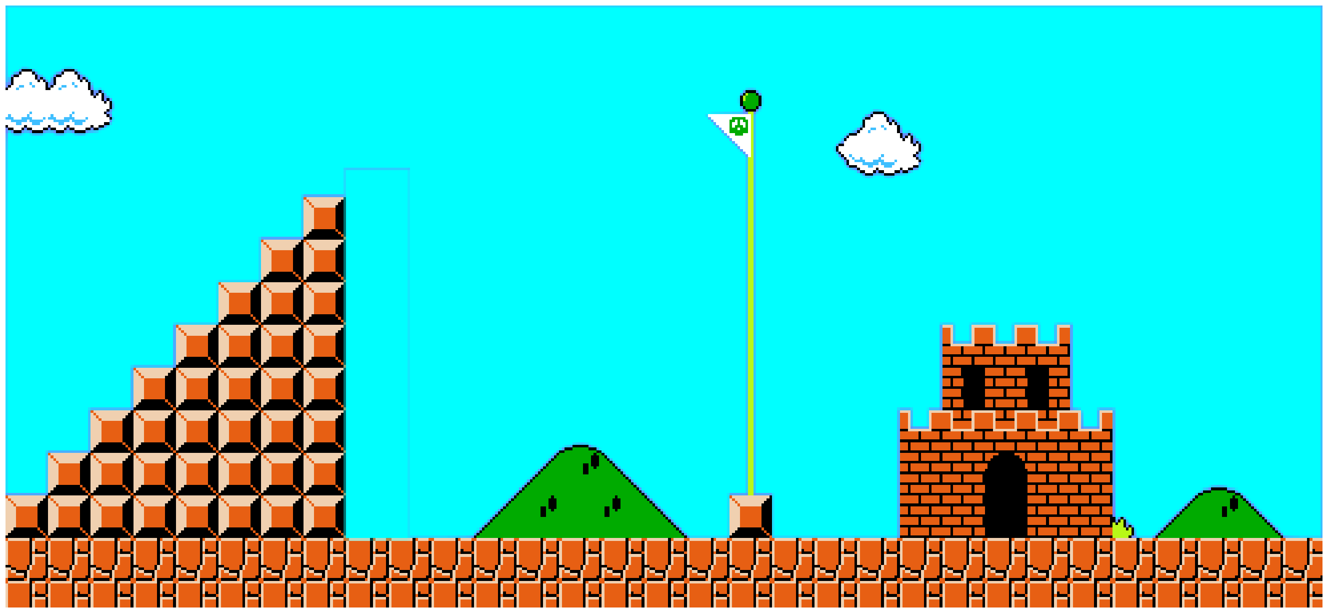
ASSIGNMENT # 1 – VARIABLES, FUNCTIONS, CONDITIONALS AND LOOPS IN C

PART A: MARIO'S WORLD – EXIT STAIR

In some of the worlds of Mario's adventure, Mario must ascend right-aligned pyramid of blocks, a la the below.



Let’s recreate that pyramid in C, albeit in text, using hashes (#) for bricks, a la the below. Of course, using text-based # block, ours will not be equal in height and width

```
       #
      ##
     ###
    ####
   #####
  ######
 #######
########
```

The program we’ll write will be called mariostair.c.

Furthermore, we will allow the user to decide just how tall the pyramid should be by frst prompting them for a positive integer between, say, 1 and 8, inclusive. Please understand the word 'inclusive'.

Now, here's how the program might work if the user inputs 4 when prompted:

```
$ ./mariostair
Height: 4
   #
  ##
 ###
####
```

Here's when user inputs 8 when prompted:

```
$ ./mariostair
Height: 8
       #
      ##
     ###
    ####
   #####
  ######
 #######
########
```

And here's when value entered is 1:

```
$ ./mariostair
Height: 1
#
```

If the user doesn’t, in fact, input a positive integer between 1 and 8, inclusive, when prompted, the program should re-prompt the user until they cooperate. This will be achieved by using cs50 library, e.g. get_int() function, and the do_while statement as has been shown in the lecture:

```
$ ./mariostair
Height: -1
Height: 0
Height: a
Height: 10
Height: 50
Height: 4
   #
  ##
 ###
####
```

Alrighty! What else with the requirements apart from the technicality requirements mentioned above then? Well first, in this assignment we need to write the pseudocode first!. Your pseudocode can be written in the vscode too by creating mariostair.txt file. Inside, just write a simple plain English sentence reflecting your pseudocode.

No custom function is required for this code but if you are bold enough, do one.

That's it! All the best and make us proud!

HINTS:

Try to print out the following structure, instead of empty whitespace before the hash-block, replace the empty whitespace with dots first. For example,

```
$./mariostair
Height: 8
.......#
......##
.....###
....####
...#####
..######
.#######
########
```

What does this mean? From the lecture, we know that we have to use two loops, which one of the loops, inner loop is nested inside the first outer loop. The outer loop is responsible to print for the height or row. For example, using for outer loop, when i = 0, that is to say that we are now printing the first row, which the second inner loop will be initiated such that it will print column block. However, you must have some conditional inside the second inner for loop such that in the first row, it will print out 7 dot blocks and right after just 1 hash block. Of course this is achieved using the j variable, which starts to print from j = 0 until j less than height (this is because the number of height represents the number of rows and number of column). Thus, your condition must make full use of the i and j variable. For example:

```
     i = 0: print first row.
     Then inner loop with j variable will start to print from j = 0 until j
     = 7, which is less than height where height = 8. Thus, from j = 0 to j
     = 6, it will print 7 dot blocks and when j = 7, it will print 1 hash
     block, which you will obtain on the first row as .......#.
     Thus, the formula to be used for the conditional:
          if (j < height – 1 - i), print dot.
          Else, print #.
```

After finishing first row printing of all the columns with dots and hash, meaning j = 8, which is no longer less than height, the inner loop will be exited, to which, we will have printf("\n") statement to move the cursor to next row.

By then i takes new value due to i++ (i = i + 1), which i now equals 1 since previously i starts with 0. This means, we are now ready to print the column block on the second row, which our second inner loop will do the job for us. Of course, with the conditional formula we used, from j = 0 to j = 5, it will print 6 dot block and when j = 6 to 7, it will print 2 hash blocks. This whole process will continue until we get our mariostair of size height! Once successfully print dots and hashes that, just replace dots with whitespaces.

DISCLOSURE REQUIREMENT: If you find any codes from external source, please provide full web address of the codes used.

PART B:

1. Identify your unique number based on your full name without bin, binti, A/L or A/P. For example, Mulia bin Minhat -> Mulia Minhat. Request from your lecturer USING class team general forum (PSST: not private chat msg system), submitting your stated full name and wait for a response from the lecturer. A number will be assigned to you based on the name submitted.
2. Based on that number, your assignment problem for part B is as below:

GREEDY ALGORITHMS:

According to the National Institute of Standards and Technology (NIST), a greedy algorithm is one “that always takes the best immediate, or local, solution while finding an answer. Greedy algorithms find the overall, or globally, optimal solution for some optimization problems, but may find less-than-optimal solutions for some instances of other problems.” What does this means? Well, take a look at your relevant problem below

NUMBER 1-3: "CHANGE"
When making change during cash transaction, odds are you want to minimize the number of coins you’re dispensing for each customer, lest you run out (or annoy the customer!). Fortunately, computer science has given cashiers everywhere ways to minimize numbers of coins due to: greedy algorithms.

Now, here we are talking about returning the coins to the customers and further, we have the following coin system in Malaysia: consisting of 50 sen, 20 sen, 10 sen and 5 sen. Here, we will assume that we have infinite amount of coins just to make things easy. Thus, to minimize the number of coins return to a customer, we use the most largest value, which is 50 sen so that the balance of change will be lesser such that lesser amount of coin will be given back to the customer. Thus here, we take a big chunk out of the change, implying the greedy process to solve the coin minimization problem.

Thus, your job is to create a change.c code that will count the minimum number of coins based on coin system of 50 sen, 20 sen, 10 sen an 5 sen, returned to the customer based on his or her total change from 0 sen to 1 ringgit, inclusive incremented by 5 sen, that also will serve as input to the program.

For example, if the change is 45 sen:

```
$./change.c
Change owed: 45
3
```

The answer 3 above is due to the fact that 2 coins for 20 sen and 1 coin for 5 sen are required to be returned to the customer, which 2 coins + 1 coins gives us 3 coins total!

Let's take a look at another example if the change is 1 ringgit or equivalent to 100 sen:

```
$./change.c
Change owed: 100
2
```

Again, the answer 2 above reflects the use of 2 coins of 50 sen. We could return 5 coins of 20 sen but then again, the latter approach does not reflect greedy algorithm where in this case, we want to take the big chunk of the balance by using the highest coin denominator, which is 50 sen. Remember, our goal is to minimize number of coins.

Of course, the last example is when the customer paid the exact change, thus no change needed to be returned to customer. For example:

```
$./change.c
Changed owed: 0
0
```

Now, if the input is not based on incremental of 5, for example, 41 sen, 73 sen, the code will re-prompt the user to re-enter the correct change value. Furthermore, if the changed owed is less than 0 and also more than 100 is entered, then the same situation of asking the user to re-enter the correct change value. For example:

```
$./change.c
Change owed: -1
Change owed: 150
Change owed: 41
Change owed: 73
Change owed: 25
2
```

The value 2 indicates 2 coins (1 coin of 20 sen + 1 coin of 5 sen) are returned back to the customer.

Implementation Details:
In change.c, fortunately we've implemented most (but not all!) of a program that prompts the user for the number of cents that a customer is owed and then prints the smallest number of coins with which that change can be made. Indeed, main is already implemented for you. But notice how main calls several functions that aren't yet implemented! One of those functions, get_cents , takes no arguments (as indicated by void) and returns an int . The rest of the functions all take one argument, an int , and also return an int . All of them currently return 0 so that the code will compile. But you'll want to replace every TODO and return 0; with your own code. Specifically, complete the implementation of those functions as follows:

Implement:
1) **get_sen** function in such a way that the function prompts the user for a number of cents using get_int and then returns that number as an int. If the user inputs a negative int or more than 100 or non-divisible by 5 value, your code should prompt the user again. (But you don't need to worry about the user inputting, e.g., a string, or a float, as get_int will take care of that for you.) Odds are you'll find a do_while loop of help.
2) **calculate_50s** function in such a way that the function calculates (and returns as an int) how many quarters a customer should be given if they're owed some number of cents. For instance, if the change is less than 50 sen, then calculate_50s should return 0. If change is from 50 sen up until 100 sen(or anything in between, then calculate_50s should return 1. Only when the change equal 100, then calculate_50s returns 2.
3) **calculate_20s** function in such a way that the function calculates the same for 20 sen as done in calculate_50s.
4) **calculate_10s** function in such a way that the function calculates the same for 10 sen as done in calculate_50s.
5) **calculate_5s** function in such a way that the function calculates the same for 5 sen as done in calculate_50s.

Note that, all these custom functions are defined before the main driver code or int main() function, which we do not have to prototype them. Furthermore, unlike functions that only have side effects, i.e. void-type, return-type functions that return a value should do so explicitly with return <some_value/expression> statement replacing the default statement of return 0!

Take care not to modify the distribution code itself, only replace the given TODOs and the subsequent return value! Note too that, recalling the idea of abstraction, each of your calculate functions should accept any value of cents, not just those values that the greedy algorithm might suggest. If the change is 85, for example, calculate_10s should return 8.

NUMBER 4-6: "PARCELS"
When sending customer's package who ordered more than 1 item from your website, most likely we want to reduce the number of packages to be sent to minimize the overall postage cost as more items in general, means higher parcel cost. At the same time, we do not want to use 1-size fit-for-all parcel. After all, profit is the goal. Fortunately, computer science has given sellers everywhere ways to minimize numbers of parcel due: greedy algorithms.

Now, here we are talking about sending parcels to the customers and further, we have the following parcel capacity or size based on the number of items ordered by customers, for example, we have 10-item, 5-item, 2-item and 1-item parcel size. We will assume that we have infinite amount of parcels just to make things easy. Thus, to minimize the number of parcels sent to a customer, we use the most largest value or largest parcel size, so that the balance of items needs to be delivered back to a customer will be lesser such that will requires lesser amount of parcel. Thus here, we take a big chunk out of the problem, implying the greedy process to solve the number of parcel minimization problem.

Thus, your job is to create a parcel.c code that will count the minimum number of parcels based on the number of parcel's capacity to store items, for example, 10-item, 5-item, 2-item and 1-item parcel capacity, sent to the customer depending on his or her total order number that varies from 1-item up to 50 items per order, and this serves as input to the program.

For example, if the ordered item is 26:
```
$./parcel.c
Items ordered: 26
4
```

The value of 3 represents 2 10-item parcels, 1 parcel of 5-item and 1 parcel for single item.

Let's take a look again if the ordered item is 50:
```
$./parcel.c
Items ordered: 26
5
```

Above value of 5 represents a total of 5 parcels of 10-item.

If items ordered is just 1:
```
$./parcel.c
Items ordered: 1
1
```

Above shows only 1 parcel for 1-item is needed.

Now, if the input is less than zero and more than 50, then the user will be prompted again on the number of items ordered. For example:

```
$./parcel.c
Items ordered: -1
Items ordered: 150
Items ordered: 51
Items ordered: 39
8
```

The value 5 indicates 5 parcels (3 parcels of 10-item + 1 parcel of 5-item + 4 parcel of 1 or single item) are sent to the customer. Now, we have to acknowledge that this situation where greedy algorithm may not be providing optimal solution since we can seen 4 parcels of 10-item rather than sending 8 parcels of different sizes of package.

Implementation Details:
In parcel.c, fortunately we've implemented most (but not all!) of a program that prompts the user for the number of items that a customer has placed an order and then prints the smallest number of total parcels with which that parcel delivery can be made. Indeed, main is already implemented for you. But notice how main calls several functions that aren't yet implemented! One of those functions, get_order , takes no arguments (as indicated by void) and returns an int. The rest of the functions all take one argument, an int, and also return an int. All of them currently return 0 so that the code will compile. But you'll want to replace every TODO and return 0; with your own code. Specifically, complete the implementation of those functions as follows:

Implement:
1) **get_order** function in such a way that the function prompts the user for a number of items ordered using get_int and then returns that number as an int . If the user inputs a negative int or more than 50 , your code should prompt the user again. (But you don't need to worry about the user inputting, e.g., a string, or a float , as get_int will take care of that for you.) Odds are you'll find a do_while loop of help.
2) **calculate_10i** function in such a way that the function calculates (and returns as an int) how many 10-item parcel a customer should be sent if they have ordered some number of items. For instance, if items ordered is less than 10, e.g. 4 or 9 , then calculate_10i should return 0. If items ordered are from 30 to any number in between 30 to 40, then calculate_10i should return 3.
3) **calculate_5i** function in such a way that the function calculates the same for 5-item parcel as done in calculate_10i.
4) **calculate_2i** function in such a way that the function calculates the same for 2-item parcel as done in calculate_10i.
5) **calculate_1i** function in such a way that the function calculates the same for single or 1-item parcel as done in calculate_10i.

Note that, all these custom functions are defined before the main driver code or int main() function, which we do not have to prototype them. Furthermore, unlike functions that only have side effects, i.e. void-type, return-type functions that return a value should do so explicitly with return <some_value/expression> statement replacing the default statement of return 0!

Take care not to modify the distribution code itself, only replace the given TODOs and the subsequent return value! Note too that, recalling the idea of abstraction, each of your calculate functions should accept any value of cents, not just those values that the greedy algorithm might suggest. If total items is 25, for example, calculate_2i should return 12.

NUMBER 7-9: "FERRY"
When a disaster struck a small town, usually the main goal of every authority is to remove the civilians from danger zone as soon as possible. This means that the authority has to ferry the civilians in a large number as much as possible, in a way as fast as possible too. In this scenario, different types of automobiles are used to ferry the civilians out of town. A bus that can ferry up to 20 people, a van of 10 people, a car of 5 people and a motorcycle of 1 people. Of course, cost is an issue which reducing number of transport will reduce the transportation cost in general. Fortunately, computer science has given authorities everywhere ways to minimize numbers of number due: greedy algorithms.

As a start, we will assume that we have an infinite amount of transports just to make things easy. Thus, to minimize the number of transport to ferry civilians , we use the largest mode of transport, so that the balance of people left in the town to be transported will be reduced extensively. Thus here, we take a big chunk out of the problem, implying the greedy process to solve the number of transport minimization problem.

Thus, your job is to create a ferry.c code that will count the minimum number of transport based on the number of people that needs to be transported, for example, a bus of 20 people, a van of 10 people, a car of 5 people, and a motorcycle ferrying 1 person, can be sent to the disaster zone to evacuate a specific number of civilians and that number of civilians becomes the input to the program.

For example, if the number of civilians is is 125:
```
$./ferry.c
Civilians ferried: 125
7
```

The total value of 8 represents 6 buses of 20 pax capacity and 1 car of 5 people.

Let's take a look again if the number of civilians is 72:
```
$./ferry.c
Civilians ferried: 72
6
```

Above value of 6 transport in total represents 3 buses of 20 pax capacity, 1 van of 10 pax capacity and 2 motorcycles carrying 1 person each.

If items ordered is just 16:
```
$./ferry.c
Civilians ferried: 1
3
```

Above shows only 3 transports are needed consisting of 1 van of 10 pax, 1 car of 5 pax and 1 single motorcycle of 1 person.

Now, if the input is less than zero, then the user will be prompted again on the number of items ordered. Furthermore, rather regrefully, your team can only handle evacuation up to 500 civilians in total only. Thus, if more, user will be prompted again too, for example:

```
$./ferry.c
Civilians ferried: -1
Civilians ferried: 501
Civilians ferried: 1000
Civilians ferried: 263
16
```

The total value of 16 indicates 13 buses are used to transport 260 civilians with additional 3 motorcycles for ferrying 3 people. Now, this is one of the example or situation where greedy algorithm may not be optimal we need to spend more manpower riding motorcycle to transport out 3 people. A car would have been better especially if the three people are from the same house.

Implementation Details:
In ferry.c, fortunately we've implemented most (but not all!) of a program that prompts the user for the number of civilians to be ferried and then prints the smallest number of transport which the civilians can be evacuated. Indeed, main is already implemented for you. But notice how main calls several functions that aren't yet implemented! One of those functions, get_civilians, takes no arguments (as indicated by void) and returns an int. The rest of the functions all take one argument, an int, and also return an int. All of them currently return 0 so that the code will compile. But you'll want to replace every TODO and return 0; with your own code. Specifically, complete the implementation of those functions as follows:

Implement:
1) **get_civilians** function in such a way that the function prompts the user for a number of civilians to be ferried out from disaster zone using get_int and then returns that number as an int. If the user inputs a negative int or more than 500, your code should prompt the user again. (But you don't need to worry about the user inputting, e.g., a string, or a float , as get_int will take care of that for you.) Odds are you'll find a do_while loop of help.
2) **calculate_bus** function in such a way that the function calculates (and returns as an int) how many buses are needed based on the number of civilians to be evacuated.
3) **calculate_van** function in such a way that the function calculates the same for 5-item parcel as done in calculate_10i.
4) **calculate_car** function in such a way that the function calculates the same for 2-item parcel as done in calculate_10i.
5) **calculate_motorcycle** function in such a way that the function calculates the same for single or 1-item parcel as done in calculate_10i.

Note that, all these custom functions are defined before the main driver code or int main() function, which we do not have to prototype them. Furthermore, unlike functions that only have side effects, i.e. void-type, return-type functions that return a value should do so explicitly with return <some_value/expression> statement replacing the default statement of return 0!

Take care not to modify the distribution code itself, only replace the given TODOs and the subsequent return value! Note too that, recalling the idea of abstraction, each of your calculate functions should accept any value of cents, not just those values that the greedy algorithm might suggest. If total items is 250, for example, calculate_motorcycle should return 250.

All these assignments are inspired by a Harvard University computer sciences program.