**ICT159 Assignment 1**

***Muhammad Talha Ijaz***

***3541863***

Progress on the assignment needs to be demonstrated in the weeks leading up to the submission of the assignment. There are no marks given unless progress is demonstrated in-person. The assignment that is submitted may require personal defence/demonstration after submission. If this defence/demonstration is needed, you will be contacted. If you do not defend/demonstrate in-person, no marks from this assignment will be used in the final total mark for the unit.

1. **Assumptions (3%)**

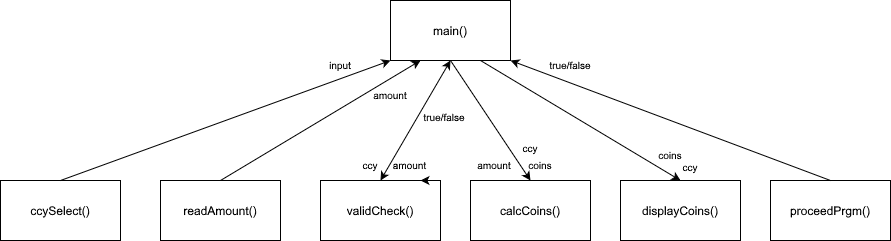
*All assumptions made other than those stated in the question that you make about the problem. There will virtually always be assumptions you are implicitly making so think about this very carefully. However, your assumptions cannot contradict the assignment specification. Also be careful that you do not put in unnecessary assumptions.*

* Program Input assumptions:
  + An integer variable will be used to store the amount and currency selection
  + The program cannot accept values greater than 95 or less than 1
  + The program cannot accept data types other than integer
  + The program will take in the currency selection one, and subsequently the amount once only
* Program Processing assumptions:
  + The program will prompt the user to enter again when an invalid input is entered
  + The program will use a formula that returns what coins will be returned for the entered value
  + The program will store the returned coins in an array
* Program Output assumptions:
  + The program will output the coins that will be returned for the input amount and selected currency
  + The program will give the user an option to exit or continue the program
* Any other assumptions:
  + Using a 2-D array to efficiently store the coins for different currencies
  + The program will be coded and compiled and run on a Windows platform

1. **Structure Chart (10%)**

*Structure chart for your program. Show parameter passing. If the structure chart is not implemented in code, you will get no marks.*

*Draw structure chart below. Do not show type information with parameter. Using meaningful names for parameters and show direction of parameter passing with arrow.*

**

*Fill in the following table which provides more detail about each parameter. Add rows as necessary.*

| **Module Name *(as shown on structure chart)*** | **Parameter name *(as shown on structure chart)*** | **Parameter type *(e.g. int, float, .etc)*** | **Parameter direction *(e.g. in, out, in/out)*** | **Purpose of Module/Parameter** | **Single Responsibility (Y/N)** |
| --- | --- | --- | --- | --- | --- |
| ccySelect() | input | int | out | it reads and returns the user’s choice of currency | y |
| readAmount() | amount | int | out | it reads and returns the user’s amount | y |
| validCheck() | ccy | int | in | passes by value the user’s choice of currency from main | y |
|  | amount | int | in | passes by value the user’s amount from main | y |
|  | true/false | int | out | returns either true/false depending on whether the entered amount is valid between the range if 1-95 and multiple of 5 for AU$ | y |
| calcCoins() | ccy | int | in | passes by value the user’s choice of currency from main | y |
|  | amount | int | in | passes by value the user’s amount from main | y |
|  | coins | int | in | an array that stores the returned coins for each cent depending on the amount entered for the chosen currency | y |
| displayCoins() | coins | int | in | an array that stores the returned coins for each cent depending on the amount entered for the chosen currency | y |
|  | ccy | int | in | passes by value the user’s choice of currency from main | y |
|  |  |  |  | the module prints out the returned coins for the amount entered | y |
| proceedPrgm() | true/false | int | out | takes in an input from the user and returns a true/false whether they want to continue to exit | y |
|  |  |  |  |  |  |

1. **Algorithm (15%)**

*Your algorithm written in a uniform fashion using a pseudo-code and adhering to the conventions required in the unit. Your algorithm should be presented at an appropriate level of detail sufficient to be easily implemented. Submit your high- level algorithm (where necessary) along with algorithms of your decompositions into modules as appropriate to the question.   
Algorithms that look like the code was written first and then word processed to look like an algorithm would receive no marks. This means no C library routines can be used. scanf and printf are just two examples of C library routines.*

START

DECLARE stop, amount, ccy: INTEGER

DECLARE coins: ARRAY[1:4] OF INTEGER

proceed ← 0

REPEAT

ccy ← ccySelect()

REPEAT

amount ← readAmount()

IF validCheck(amount, ccy) = 1 THEN

calcCoins(amount, ccy, coins

BREAK

ENDIF

UNTIL FALSE

displayCoins(ccy, coins)

stop ← proceedPrgm()continue the program

UNTIL stop = 1

PRINT(“Program Terminated.”)

END

FUNCTION ccySelect() RETURNS INTEGER

DECLARE input, type: INTEGER

WHILE TRUE

PRINT(“Please Enter a corresponding number to select —-----------from the following currencies”)

PRINT(“1. US$”)

PRINT(“2. AUS$”)

PRINT(“3. Euro”)

type ← INPUT(input)

IF type = 1 AND input >= 1 AND input <= 3 THEN

RETURN input

ELSE

PRINT("Invalid selection. Please try again.")

ENDIF

ENDWHILE

ENDFUNCTION

FUNCTION readAmount() RETURNS INTEGER

DECLARE amount : INTEGER

PRINT("Enter amount: ")

INPUT(amount)

RETURN amount

END FUNCTION

FUNCTION validCheck(input : INTEGER, ccy: INTEGER) RETURNS INTEGER

IF (ccy = 1) OR (ccy = 3) THEN

IF (input >= 1 AND input <= 95) THEN

PRINT(“Amount: “, input, “(\*\*\*Note: Decimals values to be rounded to floor\*\*\*”)

RETURN 1

ELSE

PRINT("Enter an integer value from 1-95 only.")

RETURN 0

ENDIF

ELSE IF ccy = 2 THEN

IF (input MOD 5 = 0) AND (input > 0 AND input < 96) THEN

RETURN 1

ELSE

PRINT("Value must be a multiple of 5 between 1-95 only")

RETURN 0

ENDIF

ENDIF

END FUNCTION

PROCEDURE calcCoins(input : INTEGER, ccy: INTEGER, coinCount[] : INTEGER)

DECLARE ccyCents: ARRAY[1:3, 1:4] OF INTEGER

DECLARE i : INTEGER

ccyCents ← [[50, 25, 10, 1], [50, 20, 10, 5], [20, 10, 5, 1]]

FOR i ← 1 TO 4

coinCount[i] ← input / ccyCents[ccy][i]

input ← input MOD ccyCents[ccy][i]

NEXT i

ENDPROCEDURE

PROCEDURE displayCoins(ccy: INTEGER, coinCount[] : INTEGER)

DECLARE ccyCents: ARRAY[1:3, 1:4] OF INTEGER

DECLARE i : INTEGER

ccyCents ← [[50, 25, 10, 1], [50, 20, 10, 5], [20, 10, 5, 1]]

FOR i ← 1 TO 4

IF coinCount <> 0 THEN

PRINT(coinCount[i], ccyCents[ccy][i], “cent —--------------------coin(s)”)

ENDIF

NEXT i

ENDPROCEDURE

FUNCTION proceedPrgm() RETURNS INTEGER

DECLARE input : INTEGER

PRINT(“Would you like to continue the program?”)

PRINT(“Enter 1 to continue or 2 to exit”)

WHILE TRUE

INPUT(input)

IF input = 1 THEN

RETURN 1

ELSE IF input = 2 THEN

RETURN 0

ELSE

PRINT(“Invalid input. Enter 1 to continue or 2 to exit.”)

ENDIF

ENDWHILE

ENDFUNCTION

1. **Test Table (15%)**

*A set of test data in tabular form with expected results and desk check results from your algorithm. Each test data must be justified – reason for selecting that data. No marks will be awarded unless justification for each test data is provided. A desk check is a manual check of the algorithm. So run through the algorithm one step at a time.*

Add rows to the following table as needed. Table can span more than one page. Each test id tests only one condition for the desk check. There should be no duplicated reasons listed in the second column.

| **Test id** | **Test description/justification – what is the test for and why this particular test.** | **Actual data for this test** | **Expected output** | **Actual desk check result** | **Desk check outcome – Pass/Fail** |
| --- | --- | --- | --- | --- | --- |
| 1 | Testing for a valid, upper bound value for USD currency to see if the program takes in the data. | 1, 95 | 1 x 50 cents,  1 x 25 cents,  2 x 10 cents | 1 x 50 cents,  1 x 25 cents,  2 x 10 cents | Pass |
| 2 | Testing for a valid, upper bound value for AU$ currency to see if the program takes in the data. | 2, 95 | 1 x 50 cents,  2 x 20 cents,  1 x 5 cents | 1 x 50 cents,  2 x 20 cents,  1 x 5 cents | Pass |
| 3 | Testing for a valid, upper bound value for Euro currency to see if the program takes in the data. | 3, 95 | 4 x 20 cents,  1 x 10 cents,  1 x 5 cents | 4 x 20 cents,  1 x 10 cents,  1 x 5 cents | Pass |
| 4 | Testing for a valid, lower bound value for USD currency to see if the program takes in the data. | 1, 1 | 1 x 1 cents | 1 x 1 cents | Pass |
| 5 | Testing for a valid, lower bound value for AU$ currency to see if the program takes in the data. | 2, 5 | 1 x 5 cent | 1 x 5 cent | Pass |
| 6 | Testing for a valid, lower bound value for Euro currency to see if the program takes in the data. | 3, 1 | 1 x 1 cent | 1 x 1 cent | Pass |
| 7 | Testing for an invalid value below lower bound for USD currency to see if the program handles the error | 1, 0 | Enter an integer value from 1-95 only. | Enter an integer value from 1-95 only. | Pass |
| 8 | Testing for an invalid value below lower bound for AU$ currency to see if the program handles the error | 2, 1 | Value must be a multiple of 5 between 1-95 only. | Value must be a multiple of 5 between 1-95 only. | Pass |
| 9 | Testing for an invalid value below lower bound for Euro currency to see if the program handles the error | 3, 0 | Enter an integer value from 1-95 only. | Enter an integer value from 1-95 only. | Pass |
| 10 | Testing for an invalid value above upper bound for USD currency to see if the program handles the error | 1, 96 | Enter an integer value from 1-95 only. | Enter an integer value from 1-95 only. | Pass |
| 11 | Testing for an invalid value above upper bound for AU$ currency to see if the program handles the error | 2, 96 | Value must be a multiple of 5 between 1-95 only. | Value must be a multiple of 5 between 1-95 only. | Pass |
| 12 | Testing for an invalid value above upper bound for Euro currency to see if the program handles the error | 3, 96 | Enter an integer value from 1-95 only. | Enter an integer value from 1-95 only. | Pass |
| 13 | Testing for a valid value for USD currency to see if the program takes in the data | 1, 53 | 1 x 50 cents,  3 x 1 cents | 1 x 50 cents,  3 x 1 cents | Pass |
| 14 | Testing for a valid value for AU$ currency to see if the program takes in the data | 2, 65 | 1 x 50 cents,  1 x 10 cents,  1 x 5 cents | 1 x 50 cents,  1 x 10 cents,  1 x 5 cents | Pass |
| 15 | Testing for a valid value for Euro currency to see if the program takes in the data | 3, 79 | 3 x 20 cents,  1 x 10 cents,  1 x 5 cents,  4 x 1 cents | 3 x 20 cents,  1 x 10 cents,  1 x 5 cents,  4 x 1 cents | Pass |
| 16 | Testing for an invalid negative value for USD currency to see if the program handles the error | 1, -1 | Enter an integer value from 1-95 only. | Enter an integer value from 1-95 only. | Pass |
| 17 | Testing for an invalid negative value for USD currency to see if the program handles the error | 2, -1 | Value must be a multiple of 5 between 1-95 only. | Value must be a multiple of 5 between 1-95 only. | Pass |
| 18 | Testing for an invalid negative value for USD currency to see if the program handles the error | 3, -1 | Enter an integer value from 1-95 only. | Enter an integer value from 1-95 only. | Pass |
| 19 | Testing for an invalid, negative input for selecting currency to see if the program handles the error | -1 | Invalid selection. Please try again. | Invalid selection. Please try again. | Pass |
| 20 | Testing for upper bound on selecting currency to see if it selects and continues the program | 3 | Enter Amount: | Enter Amount: | Pass |
| 21 | Testing for lower bound on selecting currency to see if it selects and continues the program | 1 | Enter Amount: | Enter Amount: | Pass |
| 22 | Testing for null input for amount for USD | 1, “” | Enter an integer value from 1-95 only. | Enter an integer value from 1-95 only. | Pass |
| 23 | Testing for null input for amount for AU$ | 2, “” | Value must be a multiple of 5 between 1-95 only. | Value must be a multiple of 5 between 1-95 only. | Pass |
| 24 | Testing for null input for amount for Euro | 3, “” | Enter an integer value from 1-95 only. | Enter an integer value from 1-95 only. | Pass |
| 25 | Testing for null input for selecting currency | “” | Invalid selection. Please try again | Invalid selection. Please try again | Pass |
| 26 | Testing a valid value for if the program will continue program | 1 | Please enter a corresponding number to select from the following currencies:  1. US$  2. AUS$  3. Euro | Please enter a corresponding number to select from the following currencies:  1. US$  2. AUS$  3. Euro | Pass |
| 27 | Testing a valid value for if the program will exit program | 2 | Program Terminated. | Program Terminated. | Pass |
| 28 | Testing an invalid value to check if the program will handle the error for continuing or exiting | -12 | Invalid input. Enter 1 to continue or 2 to exit. | Invalid input. Enter 1 to continue or 2 to exit. | Pass |
| 29 | Testing an invalid data type into currency selection to see if the program can handle the error | “abc” | Invalid selection. Please try again. | Invalid selection. Please try again. | Pass |
| 30 | Testing an invalid data type for amount to see if the program can handle the error | 1, “xyz” | Enter an integer value from 1-95 only. | Enter an integer value from 1-95 only. | Pass |
| 31 | Testing float input for amount to see if the program will handle the different data type | 1, 93.3 | Amount: 93 (Note: Decimals values to be rounded to floor) | Amount: 93 (Note: Decimals values to be rounded to floor) | Pass |

1. **Code (40%)**

*Name and purpose of functions in the source code files. Do not put actual source code here. Code exists as separate source code files that are submitted. Source code files (.c, .h) must be submitted separately in a zip file, and the source code must build (compile and link) to create an executable that operates correctly.*

*In the column, “Purpose of the function/subroutine”, you will need to ensure that the function/subroutine that you have written does only one thing: Single responsibility principle.* Find out what is meant by the “Single responsibility principle” or it relates to the cohesion principle.

High level functions (modules) like *main*() call other subroutine/functions (modules) to get the job done. These other subroutines/functions call lower level routines (designed using a structure chart) to do single tasks. As an example, a function call *sum*, only does adding items together. This function will not ask for user input and will not do any output of the result to screen or file.

*In the column “Single responsibility? (Y/N)” you need to ensure that the code implementation of the function/subroutine does only one thing (see footnote [1]), in which case the answer is Y for yes. If you find that the code function/subroutine is doing more than one thing, you must refactor the function/subroutine. Do that by first examining your algorithm and structure chart to determine how to further modularise your design.  
  
Make sure you use the code style required in the unit. No marks awarded if the table below is not filled in or if the source code does not build and run.*

Extend the rows in the following table as needed. Functions/subroutines (modules) need to match what is in the structure chart and algorithm. If there are a number of functions in the same file, you write the file name once in the *File name* column for the first function listed in the table. Successive functions (in the same file) do not need to list the file name repeatedly.

| **File name** | **Name of function/subroutine in file** | **Purpose of the function/subroutine** | **Single responsibility? (Y/N)** |
| --- | --- | --- | --- |
| main.c | main() | The main function is responsible for calling all the modules from the file “coinsCalc.h”, handling initialization of variables and the control flow of the modules. | N |
| coinsCalc.c | ccySelect() | The function displays 3 selections of currency and returns the user’s input | Y |
|  | readAmount() | The function takes in the user’s input for amount and returns it | Y |
|  | validCheck(input, ccy) | The function checks whether amount is in the required range, being 1-95 and a multiple of 5 for AU$, returns true/false depending on validation | Y |
|  | calcCoins(input, ccy, coinCount[]) | The subroutine calculates the number of coins for each cent depending on the amount and stores it into the array passed as parameter. | Y |
|  | displayCoins( coinCount[], ccy) | The subroutine |  |
|  | proceedPrgm() | The function prompts the user whether they want to continue or exit and return their input | Y |

1. **Results of Program Testing (10%)**

*Results of applying your test data to your final program (tabular form), including a sample printout of your program in operation.*

Add rows to the following table as needed. Table can span more than one page.

Each test id tests only one situation for the test run of the program. This table is a copy/paste of the desk check, except the actual output column shows the results of the actual program output. There should be no duplicated reasons listed in the second column (Test description)

| **Test id** | **Test description/justification – what is the test for and why this particular test.** | **Actual data for this test** | **Expected output** | **Actual program output when test is carried out** | **Program test outcome – Pass/Fail** |
| --- | --- | --- | --- | --- | --- |
| 1 | Testing for a valid, upper bound value for USD currency to see if the program takes in the data. | 1, 95 | 1 x 50 cents 1 x 25 cents 2 x 10 cents | 1 x 50 cent coin(s)  1 x 25 cent coin(s)  2 x 10 cent coin(s) | Pass |
| 2 | Testing for a valid, upper bound value for AU$ currency to see if the program takes in the data. | 2, 95 | 1 x 50 cents 2 x 20 cents 1 x 5 cents | 1 x 50 cent coin(s)  2 x 20 cent coin(s)  1 x 5 cent coin(s) | Pass |
| 3 | Testing for a valid, upper bound value for Euro currency to see if the program takes in the data. | 3, 95 | 4 x 20 cents,  1 x 10 cents,  1 x 5 cents | 4 x 20 cent coin(s)  1 x 10 cent coin(s)  1 x 5 cent coin(s) | Pass |
| 4 | Testing for a valid, lower bound value for USD currency to see if the program takes in the data. | 1, 1 | 1 x 1 cents | 1 x 1 cent coin(s) | Pass |
| 5 | Testing for a valid, lower bound value for AU$ currency to see if the program takes in the data. | 2, 5 | 1 x 5 cent | 1 x 5 cent coin(s) | Pass |
| 6 | Testing for a valid, lower bound value for Euro currency to see if the program takes in the data. | 3, 1 | 1 x 1 cent | 1 x 1 cent coin(s) | Pass |
| 7 | Testing for an invalid value below lower bound for USD currency to see if the program handles the error | 1, 0 | Enter an integer value from 1-95 only. | Enter an integer value from 1-95 only. | Pass |
| 8 | Testing for an invalid value below lower bound for AU$ currency to see if the program handles the error | 2, 1 | Value must be a multiple of 5 between 1-95 only. | Value must be a multiple of 5 between 1-95 only. | Pass |
| 9 | Testing for an invalid value below lower bound for Euro currency to see if the program handles the error | 3, 0 | Enter an integer value from 1-95 only. | Enter an integer value from 1-95 only. | Pass |
| 10 | Testing for an invalid value above upper bound for USD currency to see if the program handles the error | 1, 96 | Enter an integer value from 1-95 only. | Enter an integer value from 1-95 only. | Pass |
| 11 | Testing for an invalid value above upper bound for AU$ currency to see if the program handles the error | 2, 96 | Value must be a multiple of 5 between 1-95 only. | Value must be a multiple of 5 between 1-95 only. | Pass |
| 12 | Testing for an invalid value above upper bound for Euro currency to see if the program handles the error | 3, 96 | Enter an integer value from 1-95 only. | Enter an integer value from 1-95 only. | Pass |
| 13 | Testing for a valid value for USD currency to see if the program takes in the data | 1, 53 | 1 x 50 cents,  3 x 1 cents | 1 x 50 cent coin(s)  3 x 1 cent coin(s) | Pass |
| 14 | Testing for a valid value for AU$ currency to see if the program takes in the data | 2, 65 | 1 x 50 cents,  1 x 10 cents,  1 x 5 cents | 1 x 50 cent coin(s)  1 x 10 cent coin(s)  1 x 5 cent coin(s) | Pass |
| 15 | Testing for a valid value for Euro currency to see if the program takes in the data | 3, 79 | 3 x 20 cents,  1 x 10 cents,  1 x 5 cents,  4 x 1 cents | 3 x 20 cent coin(s)  1 x 10 cent coin(s)  1 x 5 cent coin(s)  4 x 1 cent coin(s) | Pass |
| 16 | Testing for an invalid negative value for USD currency to see if the program handles the error | 1, -1 | Enter an integer value from 1-95 only. | Enter an integer value from 1-95 only. | Pass |
| 17 | Testing for an invalid negative value for USD currency to see if the program handles the error | 2, -1 | Value must be a multiple of 5 between 1-95 only. | Value must be a multiple of 5 between 1-95 only. | Pass |
| 18 | Testing for an invalid negative value for USD currency to see if the program handles the error | 3, -1 | Enter an integer value from 1-95 only. | Enter an integer value from 1-95 only. | Pass |
| 19 | Testing for an invalid, negative input for selecting currency to see if the program handles the error | -1 | Invalid selection. Please try again. | Invalid selection. Please try again. | Pass |
| 20 | Testing for upper bound on selecting currency to see if it selects and continues the program | 3 | Enter Amount: | Enter Amount: | Pass |
| 21 | Testing for lower bound on selecting currency to see if it selects and continues the program | 1 | Enter Amount: | Enter Amount: | Pass |
| 22 | Testing for null input for amount for USD | 1, “” | Enter an integer value from 1-95 only. | No output | Failure |
| 23 | Testing for null input for amount for AU$ | 2, “” | Value must be a multiple of 5 between 1-95 only. | No output | Failure |
| 24 | Testing for null input for amount for Euro | 3, “” | Enter an integer value from 1-95 only. | No output | Failure |
| 25 | Testing for null input for selecting currency | “” | Invalid selection. Please try again | No output | Failure |
| 26 | Testing a valid value for if the program will continue program | 1 | Please enter a corresponding number to select from the following currencies:  1. US$  2. AUS$  3. Euro | Please enter a corresponding number to select from the following currencies:  1. US$  2. AUS$  3. Euro | Pass |
| 27 | Testing a valid value for if the program will exit program | 2 | Program Terminated. | Program Terminated. | Pass |
| 28 | Testing an invalid value to check if the program will handle the error for continuing or exiting | -12 | Invalid input. Enter 1 to continue or 2 to exit. | Invalid input. Enter 1 to continue or 2 to exit. | Pass |
| 29 | Testing an invalid data type into currency selection to see if the program can handle the error | “abc” | Invalid selection. Please try again. | Invalid selection. Please try again. | Pass |
| 30 | Testing an invalid data type for amount to see if the program can handle the error | 1, “xyz” | Enter an integer value from 1-95 only. | 1 x 1 cent coin(s) | Failure |
| 31 | Testing float input for amount to see if the program will handle the different data type | 1, 93.3 | Amount: 93 (Note: Decimals values to be rounded to floor) | Amount: 93 (Note: Decimals values to be rounded to floor)  1 x 50 cent coin(s)  1 x 25 cent coin(s)  1 x 10 cent coin(s)  8 x 1 cent coin(s) | Pass |

After completing the above test table, copy/paste printouts of your program in operation. Use the values (column 3) in your table to do the test runs. You can screen capture and paste below. Make sure you label each printout with the correct *Test id* and Test description.

| **Test id** | **Test description/justification – what is the test for and why this particular test.** | **Program Output** |
| --- | --- | --- |
| 1 | Testing for a valid, upper bound value for USD currency to see if the program takes in the data. |  |
| 2 | Testing for a valid, upper bound value for AU$ currency to see if the program takes in the data. |  |
| 3 | Testing for a valid, upper bound value for Euro currency to see if the program takes in the data. |  |
| 4 | Testing for a valid, lower bound value for USD currency to see if the program takes in the data. |  |
| 5 | Testing for a valid, lower bound value for AU$ currency to see if the program takes in the data. |  |
| 6 | Testing for a valid, lower bound value for Euro currency to see if the program takes in the data. |  |
| 7 | Testing for an invalid value below lower bound for USD currency to see if the program handles the error |  |
| 8 | Testing for an invalid value below lower bound for AU$ currency to see if the program handles the error |  |
| 9 | Testing for an invalid value below lower bound for Euro currency to see if the program handles the error |  |
| 10 | Testing for an invalid value above upper bound for USD currency to see if the program handles the error |  |
| 11 | Testing for an invalid value above upper bound for AU$ currency to see if the program handles the error |  |
| 12 | Testing for an invalid value above upper bound for Euro currency to see if the program handles the error |  |
| 13 | Testing for a valid value for USD currency to see if the program takes in the data |  |
| 14 | Testing for a valid value for AU$ currency to see if the program takes in the data |  |
| 15 | Testing for a valid value for Euro currency to see if the program takes in the data |  |
| 16 | Testing for an invalid negative value for USD currency to see if the program handles the error |  |
| 17 | Testing for an invalid negative value for USD currency to see if the program handles the error |  |
| 18 | Testing for an invalid negative value for USD currency to see if the program handles the error |  |
| 19 | Testing for an invalid, negative input for selecting currency to see if the program handles the error |  |
| 20 | Testing for upper bound on selecting currency to see if it selects and continues the program |  |
| 21 | Testing for lower bound on selecting currency to see if it selects and continues the program |  |
| 22 | Testing for null input for amount for USD |  |
| 23 | Testing for null input for amount for AU$ |  |
| 24 | Testing for null input for amount for Euro |  |
| 25 | Testing for null input for selecting currency |  |
| 26 | Testing a valid value for if the program will continue program |  |
| 27 | Testing a valid value for if the program will exit program |  |
| 28 | Testing an invalid value to check if the program will handle the error for continuing or exiting |  |
| 29 | Testing an invalid data type into currency selection to see if the program can handle the error |  |
| 30 | Testing an invalid data type for amount to see if the program can handle the error |  |
| 31 | Testing float input for amount to see if the program will handle the different data type |  |

1. **Self-Assessment (7%)**

*Self-assessment of how successful you were in achieving the requirements and a discussion of any problems you encountered. This write up is done in this document in the space provided below.*

Write the self-assessment here. Use as much space as needed.

I have been successfully able to implement the required logic and ensure my program is highly modular and my modules display high cohesion. Initially the main logic for returning the coins seemed very repetitive and unusually long. I then explored the choice of having a dedicated module for each currency to return coins, however I still felt it could be improved due to high repetitiveness. My solution was to implement a 2-dimensional array which can be passed into a general coin calculator formula that made use of a loop to iterate and calculate and eliminate the need for making unnecessarily long lines. Another problem I encountered was ensuring my modules were designed and plugged into the main such so it handles errors and persists till the user enters a valid input, but I did find issues with handling float inputs in my program also for which the program automatically rounds them off instead. However, most of the program was implemented nearly effortlessly and excellently outputs only the coins which will be returned, an improvement I found in the program which would normally output all 4 coins and their count, as opposed to my improvement which ignores coins with no count.

**You also need to submit a separate file called *evaluation.txt***. This file has two headings and you enter the required **summary as dot points** under the headings. The first heading is “**What works**” and the second heading is “**What does not work**”. Do not make any false claims. *A false claim here would mean that marks for this component would not be awarded. So make sure that you have tested your program thoroughly. Note that you may already not have marks for the code.*

The file *evaluation.txt* will also declare if you have checked each submitted file for viruses or malware. Name the tool and version number of the tool that you used to conduct the check. If the checks for viruses/malware are not made and the declaration is not shown in *evaluation.txt*, this assignment will not be marked and no marks will be given to you. Any delay that results from virus or malware will incur the specified daily penalty for the assignment. Advice on how to do a malware scan is under Unit Info or Essential Resources at the LMS site for this unit.

## Did you demonstrate progress on this assignment to your lecturer prior to submission?

**Y/N)** Y

Name of Lecturer: Hajrah Jahan

Demonstration Date:16/10/2024