**Feedback for Deliverable**

# ICT159 Ass1 2018.1

| ↓ | □ | ↓ | ○ | ↓ | Δ | ↓ |
|---|---|---|---|---|---|---|
| **Did not** achieve outcomes at all. | **Sub-standard** achievement of outcomes. | **Flawed** achievement of outcomes. | **Half** achieved the outcomes. | **Competent** achievement of outcomes. | **Superior** achievement of outcomes. | **Outstanding** achievement of outcomes. |

Time taken to receive this feedback: **2 days, 21 hrs, 47 mns.**

| Student Number | 33170193 | Student Name | Chong:Jin Cherng | Mark: TBA |
|---|---|---|---|---|

1

**Assumptions**:

**General Feedback for Deliverable 1**

There will always be assumptions, no matter how general or obvious they might be. For example, you might assume that the data is being read from the keyboard and the results printed to the screen.
Also, because of the use of scanf() you will be forced to assume that the value being entered is of the correct data type.
You should be aware in general that, within the limitations of scanf(), your programs should be as robust as possible. However, some students may get around this by specifying it as an assumption, for example *Assume that the money entered will be positive*.
This is fine as an assumption although it may be regarded as a minor (but recognised) weakness in the actual algorithm.

Learning Attributes section:

*Basic Comprehension and analysis Skills - What are valid assumptions?.*

**Specific Feedback from Marker for deliverable 1**

Good Day Jin.
Feedback for you for ass1.

- Really nice thinking about your assumptions, Jin.

2

**Algorithm**:
Your algorithm should be written in a uniform fashion (pseudo-code or a similar style).
Your algorithm should be presented at an appropriate level of detail, sufficient to be easily implemented.

**Algorithm**: Correctness

**General Feedback for Deliverable 2**

The algorithm must be complete, consistent and unambiguous. The more relaxed nature of structured English makes it too easy to produce algorithms that dont meet these requirements so become more precise. In particular :

Read in the user's input
Convert to lower case
Add up the numbers
Validate the data

Wherever possible in algorithms, students should be specifying names to identify the data they are working with.
So the first step is slightly ambiguous in the sense that no name as been assigned to the data read in.
The second and third are worse because they dont specify what is being converted/which numbers are being added up - this sort of vagueness becomes much harder to allow to exist if a name has been specified for the data being processed.
The last line is the worst of all because it now refers to *the data* which may or may not be the input that was read in. However, even if it did refer to a specific piece of data it is clearly ambiguous and incomplete in that it doesn't even suggest how the data should be validated. At this point it is no longer an algorithm

3

but still english!!

Learning Attributes section:

*Basic Algorithm Development Skills .*

**Specific Feedback from Marker for deliverable 2**

- In function ChkMoney, what is the default value of chkValid?????
- You do not show that the parameters of function Change are pass by reference, so how do they change the parameter values?
- Your function parameter list is long and it is easy to incorrectly pass in the wrong parameter in the wrong place (ie the cents vs the dollars).
- Q2: You do not do error checking of cents being greater than 95 or less than 5.
- Look at this:

If(gtMoney < 0) Then
Output "Warning!!! You've entered a negative value! Not only does it not make sense but the program may give you wrong error."
EndIf
perhaps "wrong answer" would be better wording.

4

**Algorithm**: Correctness - Question 1

5

**Algorithm**: Correctness - Question 2

6

**Algorithm**: Style (indentation, variable names, etc)

**General Feedback for Deliverable 3**

Other things to be careful of include:
Consistent use of *variables* etc. That is, identifying the name of particular value and using this name in each part of the algorithm.
Declare variables in the algorithm just as they need to be with the code.
Use descriptive variable names and in lower case.
Use consistent and proper indenting to show what part of the algorithm is part of which structure - this is critical!
The indentation style that is used must be that taught in the lecture notes.
Consistency between the high level algorithm and the low-level (where appropriate).
A clean flow of logic through the algorithm such that, wherever possible, structures have one entry point and one exit point.
**No** use of effective *go to* statements, either explicit or implicit.
The algorithm must use the three structures (sequence, selection, iteration).
The same applies to the use of **break** and **continue** statements in loops, and these should be avoided wherever possible.
We want an algorithm which aims to be as simple as possible while properly solving the problem -- that is, no unnecessary complexity!

Have a look at http://xkcd.com/1513/ for some comments on style!

Learning Attributes section:

*Adopting and following professional standards*

7 **Algorithm**: Style - Question 1

8 **Algorithm**: Style - Question 2

9 **Algorithm**: Efficiency
You algorithm must be a correct and efficient solution to the problem.

**General Feedback for Deliverable 4**
You algorithm must be a correct and efficient solution to the problem.

Learning Attributes section:
*Basic Computing Analysis skills*

**Specific Feedback from Marker for deliverable 4**
- Can you think how you might remove the issue of the long parameter list?

10 **Algorithm**: Efficiency - Question 1

11 **Algorithm**: Efficiency - Question 2

12 **Test data and Test Table**:

13 **Test data and Test Table**: Selection of inputs

**General Feedback for Deliverable 5**
The proper testing of all appropriate boundary conditions (i.e., before, on and immediately after the boundary) and also for special cases
(e.g., negatives, zero values etc.)

Learning Attributes section:
*Problem Analysis, Testing and Reporting.*

**Specific Feedback from Marker for deliverable 5**
- Your testing is extensive, but there are many test values that contribute little to the testing
output - they duplicate existing tests. Jin: what other tests would show the problems in this
algorithm?

14 **Test data and Test Table**: Selection of inputs - Question 1

15 **Test data and Test Table**: Selection of inputs - Question 2

16 **Test data and Test Table**: Desk Checking

**General Feedback for Deliverable 6**
You should have desk-checked the test data against the algorithm to ensure that the results are actually correct for your algorithm.

Learning Attributes section:

Learning Attributes section:

*Problem Analysis, Testing and Reporting.*

17 **Test data and Test Table**: Desk Checking - Question 1

18 **Test data and Test Table**: Desk Checking - Question 2

19 **Program Code**:
Remember to lay your code out neatly, use meaningful names for your variables and comment your code where appropriate.

20 **Program Code**: Indentation and Layout

**General Feedback for Deliverable 7**

Indentation is an issue that has been discussed all semester and it is a major part of good program style.
A well-indented program is easier to understand and analyse, particularly for a person who didn't write it. So someone should be able to look at the program and see pretty easily its general structure. If this is true then the indentation is probably at least reasonable!
Obviously there are different styles of indentation but students are required to use the style that is taught in lecture notes etc.
The biggest problem with indentation seems to be with **if** statements where the true clause and false clause MUST line up.
An indentation style where the **else** and the following **if** are on the same line is not good.

Learning Attributes section:

*Adopting and following professional standards*

**Specific Feedback from Marker for deliverable 7**

- Use consistent indentation for the code.

21 **Program Code**: Indentation and Layout - Question 1

22 **Program Code**: Indentation and Layout - Question 2

23 **Program Code**: Quality

**General Feedback for Deliverable 8**

Does the code does match the algorithm? It is not enough to simply produce a semi-working algorithm and then write a program from this which then needs extensive work to get running properly. If the two do not match in any significant way then this is not good quality.

Learning Attributes section:

*Basic Computing Coding skills*

24 **Program Code**: Quality - Question 1

25 **Program Code**: Quality - Question 2

26

**Program Code**: Other Code Style

**General Feedback for Deliverable 9**

The stylistic conventions relating to the code are those that have been given throughout the semester during lectures.
Constants should be in all upper case (and ideally used wherever appropriate), and variables should be in lower case.
Function names should in general be mixed case starting with a capital letter for each word.

Identifier names should be concise, descriptive and self-explanatory wherever possible, although throw-away temporary variables are permitted.
Comments should be used where appropriate but not over-used. Commenting every line is counter-productive and the comments should be clearly thought out in order to convey important information to an informed reader of the code. If a piece of code doesn't really require a comment since it is self-explanatory then that is OK, but if a comment should be there to explain something and is not, then that is not OK.
It is also important that comments do not interfere with the overall indentation structure, so the comment should be lined up with the line of code it corresponds to.

Have a look at http://xkcd.com/1513/ for some comments on style!

Learning Attributes section:

*Basic Computing Maintenance skills*

**Specific Feedback from Marker for deliverable 9**

- Add small comments around the C-code segments to aid in readability.

27 **Program Code**: Other Code Style - Question 1

28 **Program Code**: Other Code Style - Question 2

29 **Code on disk?**:
It is expected that the code is on the disk as required.

YES NO

30 **Code on disk: compiling and running**:

**General Feedback for Deliverable 10**

A soft copy (on CD) of your source code **in the root folder**.
Did the code compile and run from this source ?

Learning Attributes section:

*Basic Computing Development skills*

31 **Code on disk**: compiling and running - Question 1

32 **Code on disk**: compiling and running - Question 2

**Code on disk: correctness**:

33

**General Feedback for Deliverable 11**

Does the program work correctly given the various inputs? Is it partially working? Is it a logic error or a minor compilation problem?
How does the program behave in relation to the assumptions made by the you? For example, if the program is unable to deal with input of an incorrect type and this is not clearly stated as a limitation in the assumptions then you will be penalised for this section.

Learning Attributes section:

*Code Analysis and Reporting*.

**Specific Feedback from Marker for deliverable 11**

- I found it difficult to see how much change as given because everything is on one long line!

34 **Code on disk**: correctness - Question 1

35 **Code on disk**: correctness - Question 2

36 **Program testing outputs**:

**General Feedback for Deliverable 12**

This should confirm that the program implements the algorithm correctly and indicate on the test table that this is valid.

Learning Attributes section:

*Code Analysis, Testing and Reporting*.

**Specific Feedback from Marker for deliverable 12**

- Extensive testing!

37 **Self Assessment**:

**General Feedback for Deliverable 13**

Self assessment of how successful you were in achieving the requirements and a discussion of any problems you encountered.
Did you give a brief rundown of how you went about the process?
Or does this include a brief discussion on different possible approaches and why they ultimately chose what they did?
Did you show your understanding of the problems you had as you did this assignment?

Learning Attributes section:

*Life Long Learning*.

**Specific Feedback from Marker for deliverable 13**

- Jin, excellent reflection!
Think about this assessment feedback as well.

**Comments**

**General Feedback to all students**

Some students' algorithms were much too imprecise. By this stage - week 8 -
you should be able to create precise pseudo-code that does not have ambiguous
english sentences in it. For example, what does:
read the input
mean? Is it just a vague description, does it mean to read the input into a variable
called input, or is the input just put somewhere else "magically?

Some students should have used constants to represent the value of the denominations
**not** variables. The value of a 50c coin is fixed and constant - it does not vary.
CONSTANT INTEGER fiftyCentCoin = 50
CONSTANT INTEGER twentyCentCoin = 20
etc
and then pass these constants into your modules.

Change existing code to meet new requirements. Don't keep old and inappropriate variable
names from previous bits of example code.

Some indentation of the nested if-then-else code was not what is required by the
lecture notes. Similarly for do-while loops, etc.

The use of braces {} are for C, not necessarily for pseudo-code.
And **algorithms** would not have scanf "%lf" type statements.

Some students had redundant checks in their coin testing code.

What about negative amounts of change?

Some students did not test their algorithm and **then** their code properly.

Follow what the assignment specification says! Don't arbitrarily choose to do
what might be convenient or easier for you.... that is a good way to lose marks.
The specification said:
*Note that for this problem the principle of code reuse is particularly important and a significant number of marks are allocated to this. You should attempt to design your solution such that it consists of a relatively small number of functions that are as general in design as possible and you should have* ***one function in particular that can be reused (called repeatedly) in order to solve the majority of the problem.***
*If you find that you have developed a large number of functions which each perform a similar task (or have a lot of repeated code) then attempt to analyse your design to generalise the logic so that it may be reused.*
Many students just ignored this part of the assignment!
The simplest way to solve it is probably using integer division and then subtract away the
result each time (or use modulus to get the remainder). This should then be abstracted into
a single module which takes the change amount and the coin denomination and calculates how
many of that coin are required. Since there are four different types of coins this module
can be called four times with different inputs, namely each denomination of coin.
This is a great abstraction of the problem, particularly since it allows this function to
be re-used for the dollar amounts when doing part (b).

A good high level algorithm should probably be something like:
Get Amount
Calculate Change
Output Results

However, some students may have separate modules for each denomination of coins and usually
this will mean that these modules will output the results as they go.
Outputting of results from within the same function that does the calculation is not good
as this indicates low cohesion. And having many functions which solve the same problem
means that you have not properly considered how to generalise this code in order to maximise its reuse.

It is acceptable to either print out the results as you go (i.e., from the main function when
returned from their calculating function) or to have a separate module for printing
(although this may be a bit messy as there will be lots of parameters).
Best is to make a general printing module and simply call this to do the work.

One thing to be careful about in Q2 is the passing of data between functions.
Variables should only be passed when this is required and then only passed by reference
**when the variable needs to be changed**.
Some chose to use **return** values in preference to pass by reference and this is fine.
A good rule of thumb is that if only one value is returned from a function, use **return**
rather than a single pass by reference.

---

No Specific Outcomes file: specific_outcomes.html

---