**Introduction**

The purpose of the report is to document the findings of the investigation into the quality of a piece of software developed for texture assessment.

A texture assessment is carried out by processing data collected by a surface texture instrument in order to generate statistical or frequency spectrum of the data collected.

The program calculates parameters and spectral data from the above instrument. In particular:

*Load data from file –* The data is stored in a text file. It is assumed that it was captured using a texture assessment device

*Calculate the frequency spectrum data –* The algorithm performs calculations of the Fourier transform

*Save the spectral data –* The data is stored in text format file

*Compute parameters –* The parameters for the surface texture are computed

The report will focus on the accuracy and quality of algorithms for delivery of these parameters.

**Quality Requirements**

The quality of software depends on how well it meets the requirements. There are two types of requirements; functional and non-functional.

Functional requirements detail the operations that the software needs to perform such as, in the case of this task, store data in text format or calculate the Fourier transform of the parameters.

Non-functional parameters would include things that support delivery of functional requirements such as readability of the code and availability of support service for the end user.

There is a degree to which quality is necessary that depends on the use case. Only the minimum standard of quality is required but it is sensible to exceed this standard. However there is a limit to how much quality is feasible to achieve given time and resource constraints. For example, in relation to the code example given for this task, this will be determined by whether the program is intended for use by an individual or a small group of users or the wider market. The former may not require the same standard of quality as the latter.

The quality requirements for this code would be:

Functional:

* The program must load data from file

*Handled by load.c*

* Calculate frequency spectrum of data in form of a Fourier transform

*Handled by fourier.*

* Save the data in .txt format

*Handled by load.c*

* Calculate surface texture parameters

*Performed in fourier.c and fft.c, however the function blocks overlap in functionality*

Non-functional:

* Maintainability of code

*Some variables are defined in a way that can cause errors down the line*

* Reusability of code

*The code is generally well commented*

* End user support

*No evidence available for this*

* Compatibility with the gathered data

*The code uses doubles, the data provided is type float*

**Testing Requirements**

Testing is a practice of running a program with direct intention of finding a previously undiscovered fault. Depending on the complexity of the code and availability of resources, it may be difficult to carry out a full test of the code. For example, the time/cost implications may be too severe or the code may require a piece of hardware that is unavailable. Also, testing becomes more difficult with increase of complexity of the system.

Therefore testing should be carried out whenever it is feasible to do so. Furthermore it is a process that requires strategy and planning as, in order to deliver a robust piece of code, many tests would be required. It may thus be necessary, at least from cost/time investment perspective, to automate testing as much as possible.

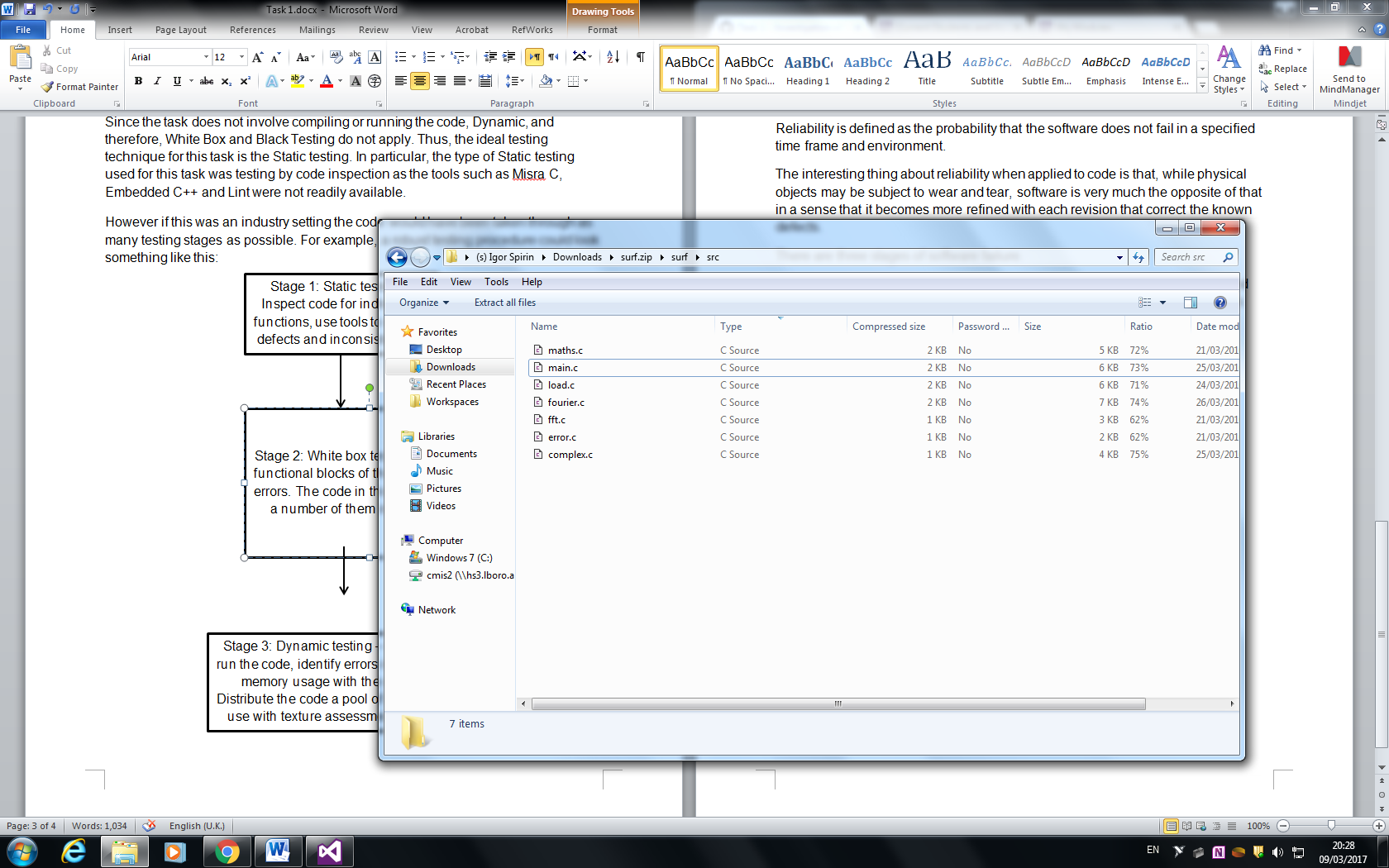
The aim of testing is to indicate presence of errors and reduce defects. However testing has some limitations as the extent of testing is determined by various standards - in-house, national and international.

There are two types of testing, static and dynamic. Static testing focuses mainly on error prevention through code inspection or automatic tests by various tools or computers. The tools include MISRA C, Embedded C++, and Lint. The code is not compiled or run during static testing.

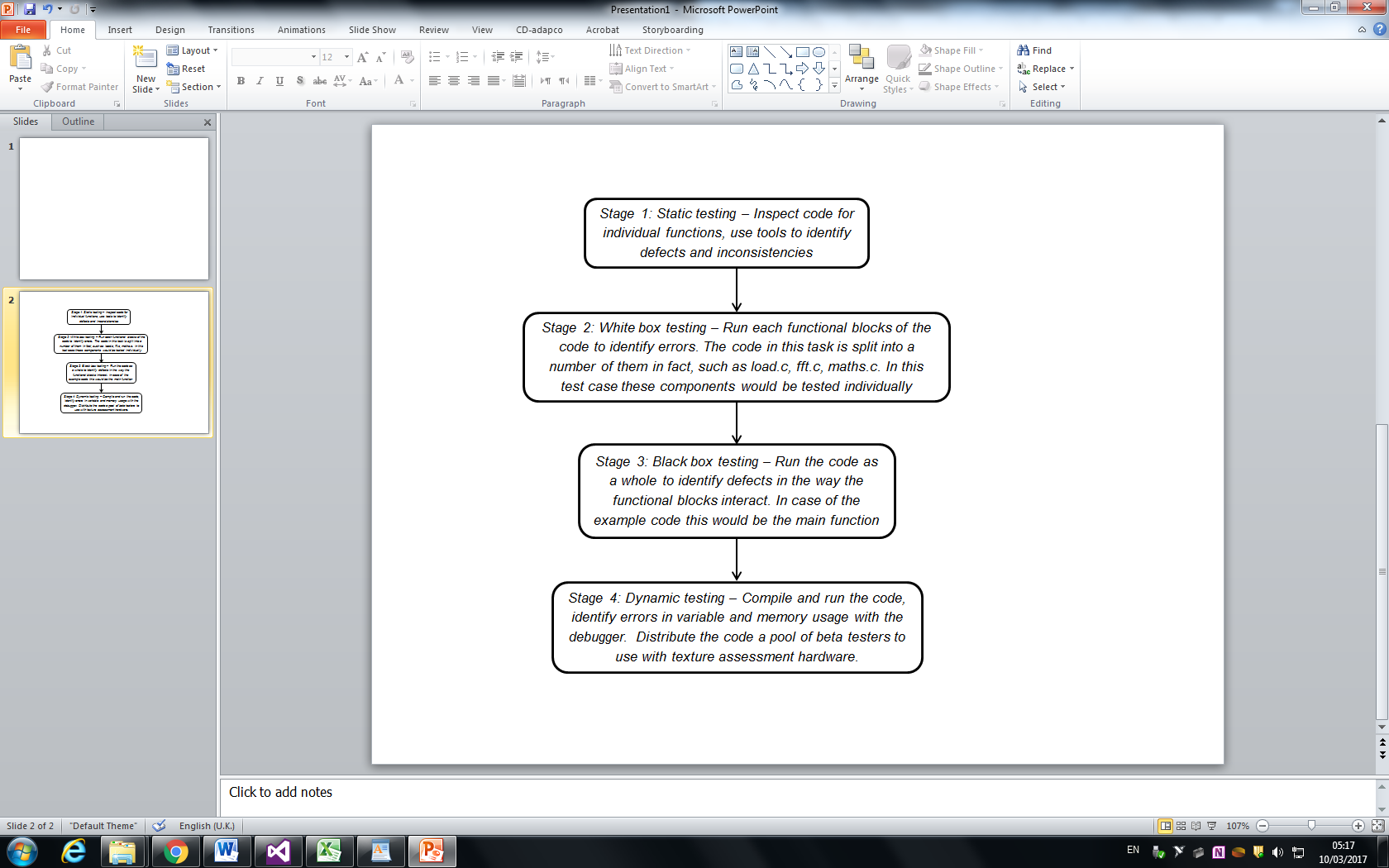
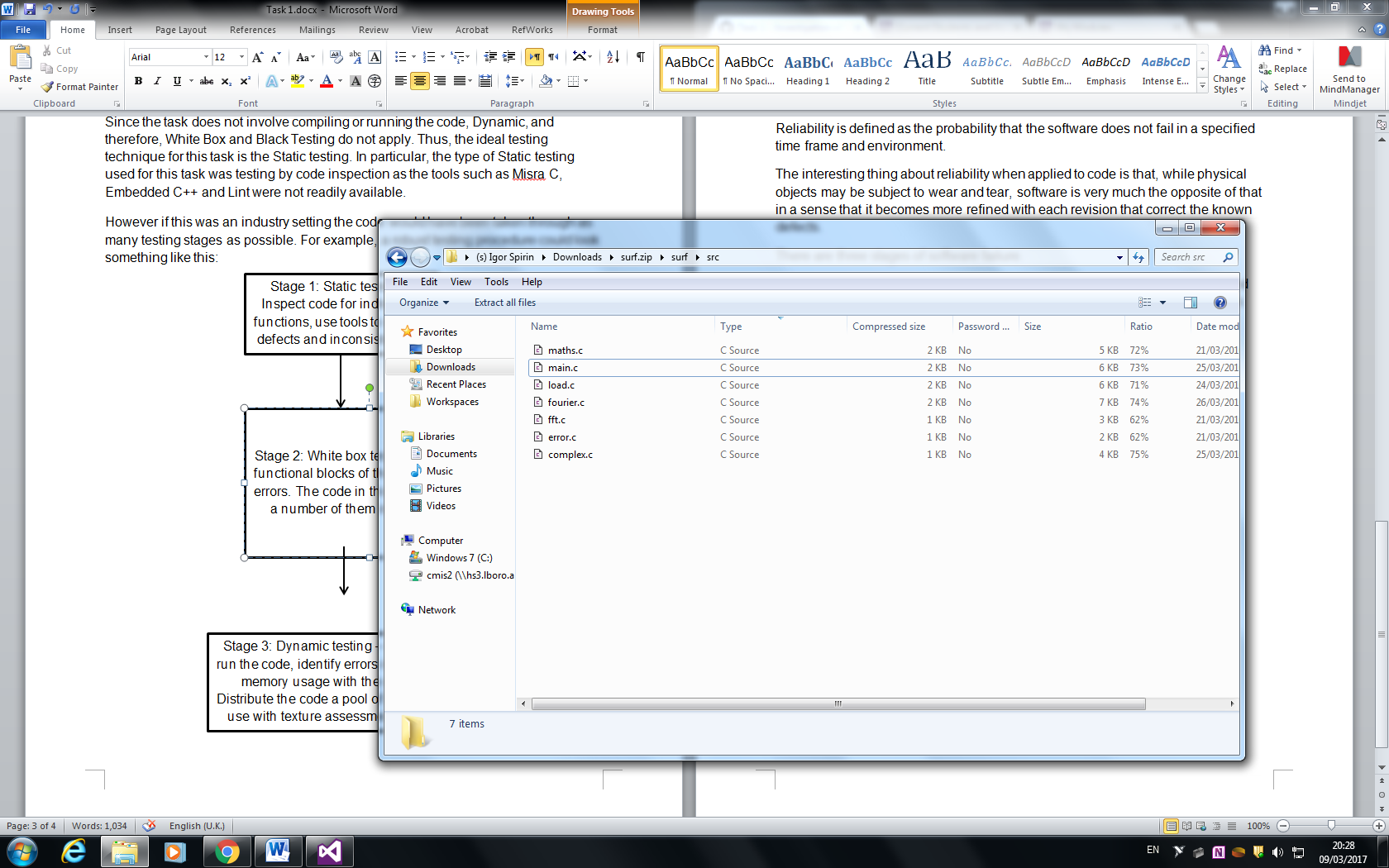
Dynamic testing is different from static testing in requiring the code to be compiled and run. Manual testing would include use by ‘beta’ testers or end users. It can also be tested automatically through heap checking which involves checking whether there is sufficient memory for the dynamic variables, that no garbage data is collected and stack checking where actual memory use is compared to the defined value.

Furthermore there is white box testing which focuses on individual functional blocks and black box testing which focuses on the operation of the code as a whole on the interface level.

Since the task does not involve compiling or running the code, Dynamic, and therefore, White Box and Black Testing are not feasible. Thus, the ideal testing technique for this task is the Static testing. In particular, testing by inspection.

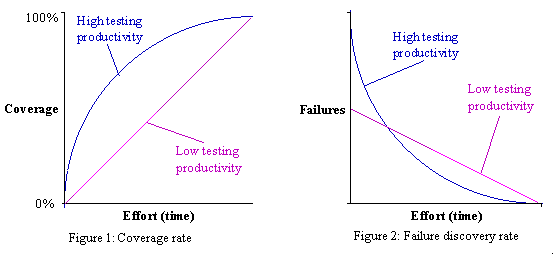
However if this was an industry setting the code would have been taken through as many testing stages as possible. For example, a robust testing procedure could look something like this:

**Test individually**



**Test main.c**

As mentioned previously the success of testing is measured in the number of defects found relative to the time put into testing. Thus:



The relationship between the test coverage and time investment should not be linear in a successful test. For instance, the example code may not support a certain type of document. This is ‘low hanging fruit’ error that can be found quickly by testing across several document types.

Figure 2 shows the relationships between detected failures and effort. Ideally the relationship should be exponential.

Simple Linear Reliability model can be used to explain the value added.

Assuming failure rate can fall to zero (doesn’t happen in real world) the graph shows that the failure rate falls as more faults are discovered.

The model is based on the formula:

Where is failures this week, is initial rate of failure and is failures recorded thus far. Total number of predicted failures is given by .

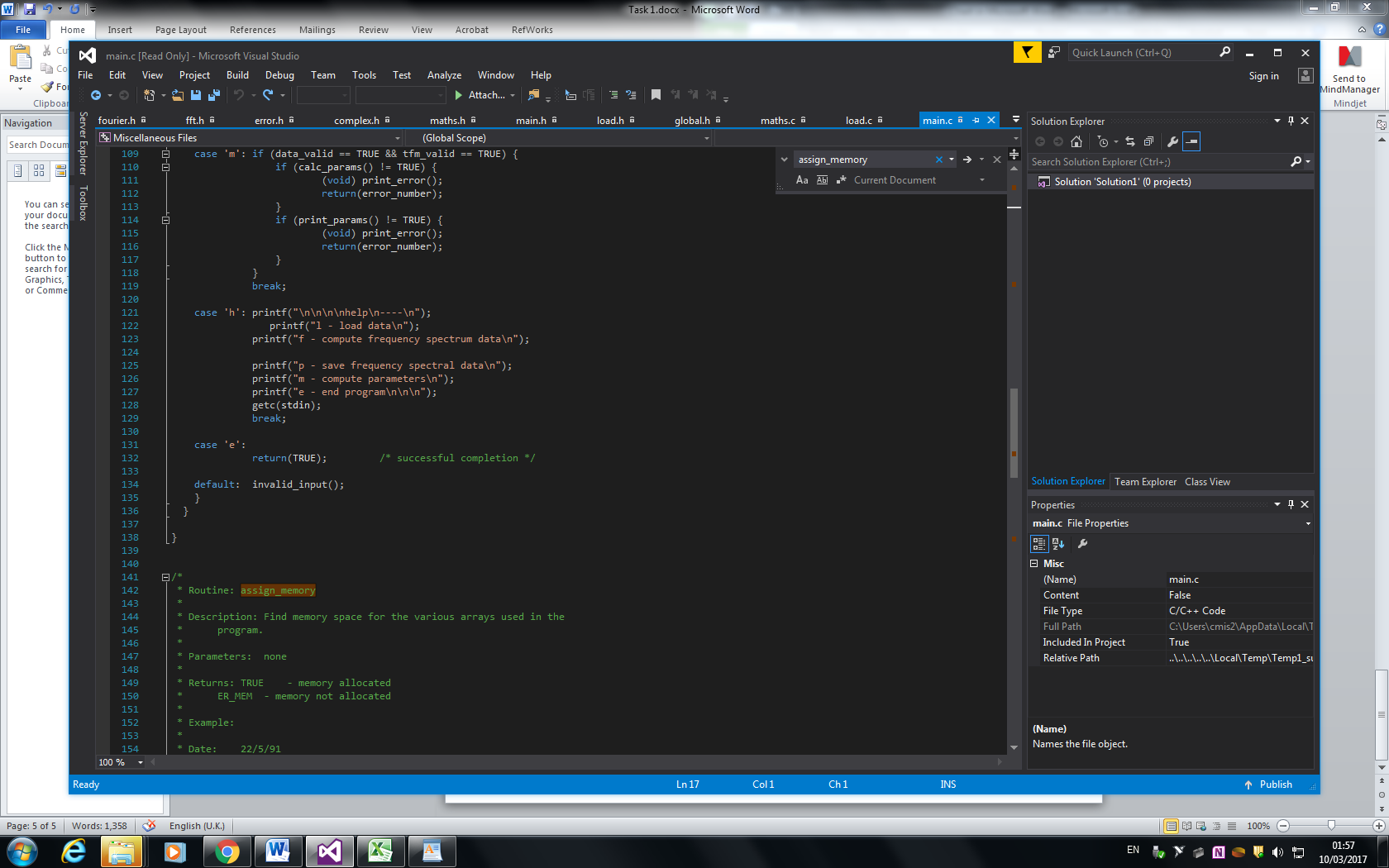
**Reliability**

Reliability is defined as the probability that the software does not fail in a specified time frame and environment. Physical objects lose reliability over time but software becomes more refined over time with each revision that corrects the known defects.

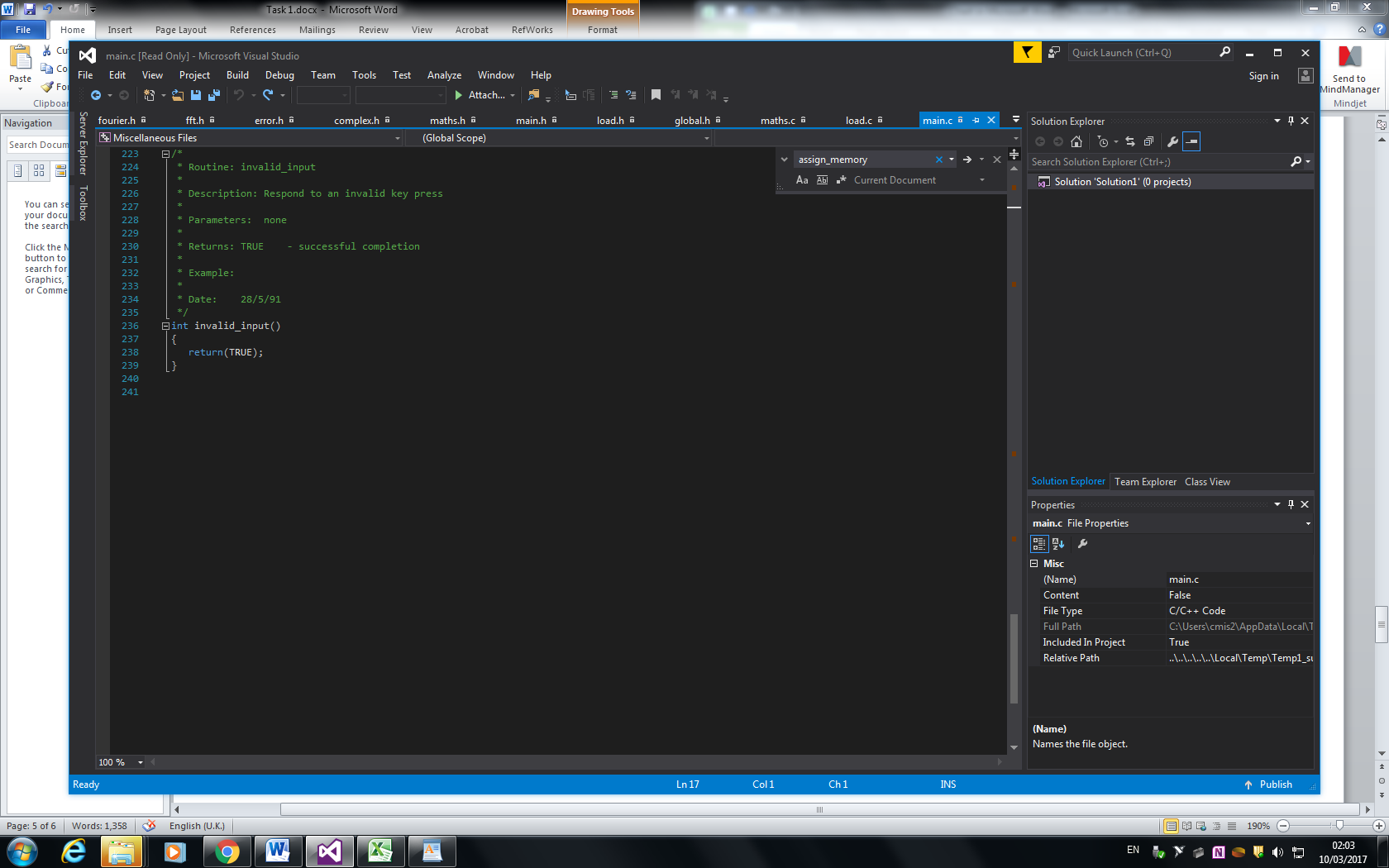
*Fault* is a defect that can propagate to the subsequent software components and cause errors down the line. *Errors* occur when the actual state of the software is different from the defined state, causing failure. *Failure* then occurs when the component ceases to perform its function.

Some things that can affect reliability in the example code:

1. main.c (line 132) will return TRUE and exit from ‘e’ command. If a wrong input is entered, the code will return a TRUE as well (lines 236-239). The code should output an error message upon detection of a wrong user input instead of exiting.

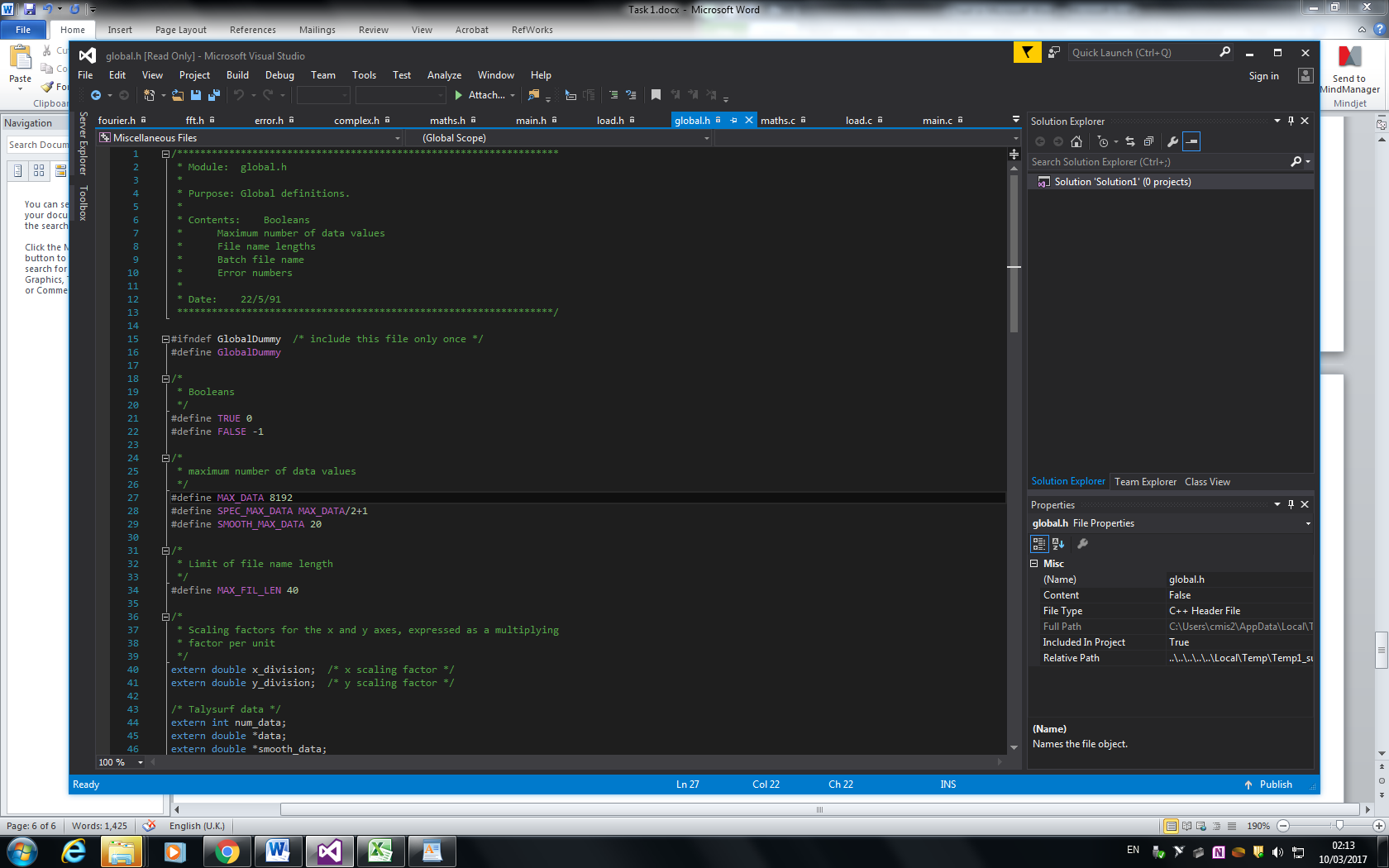


**Exit condition return**



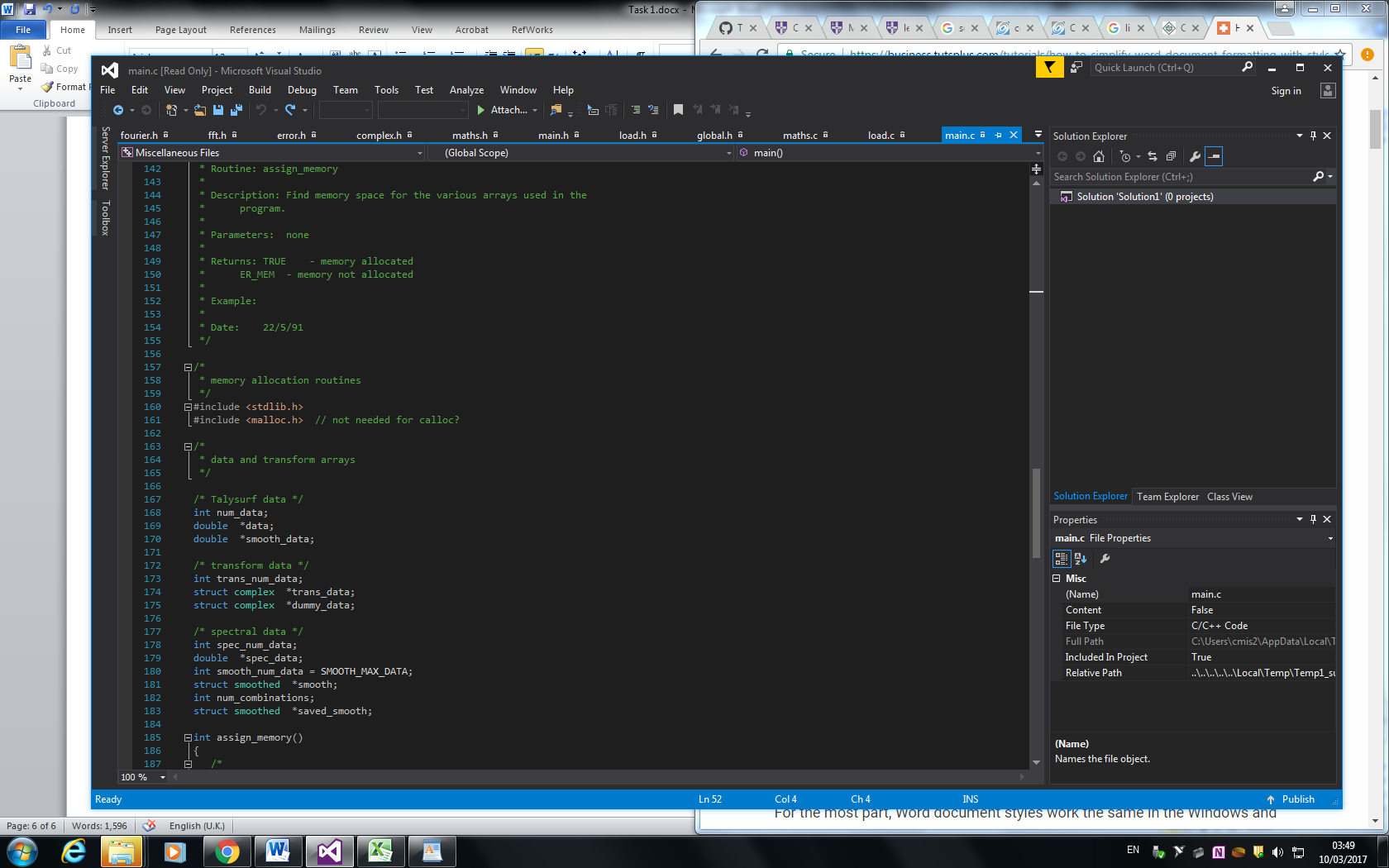
**Invalid input return**

1. In Global.h (lines 21, 22), TRUE and FALSE conditions are redefined as 0 and -1 respectively. From future development perspective it can be a cause of severe issues as these conditions are generally expected to be 1 and 0 respectively. For example, the next developer can create a while(TRUE) loop, but it will not work because in context of the code it means while(0).



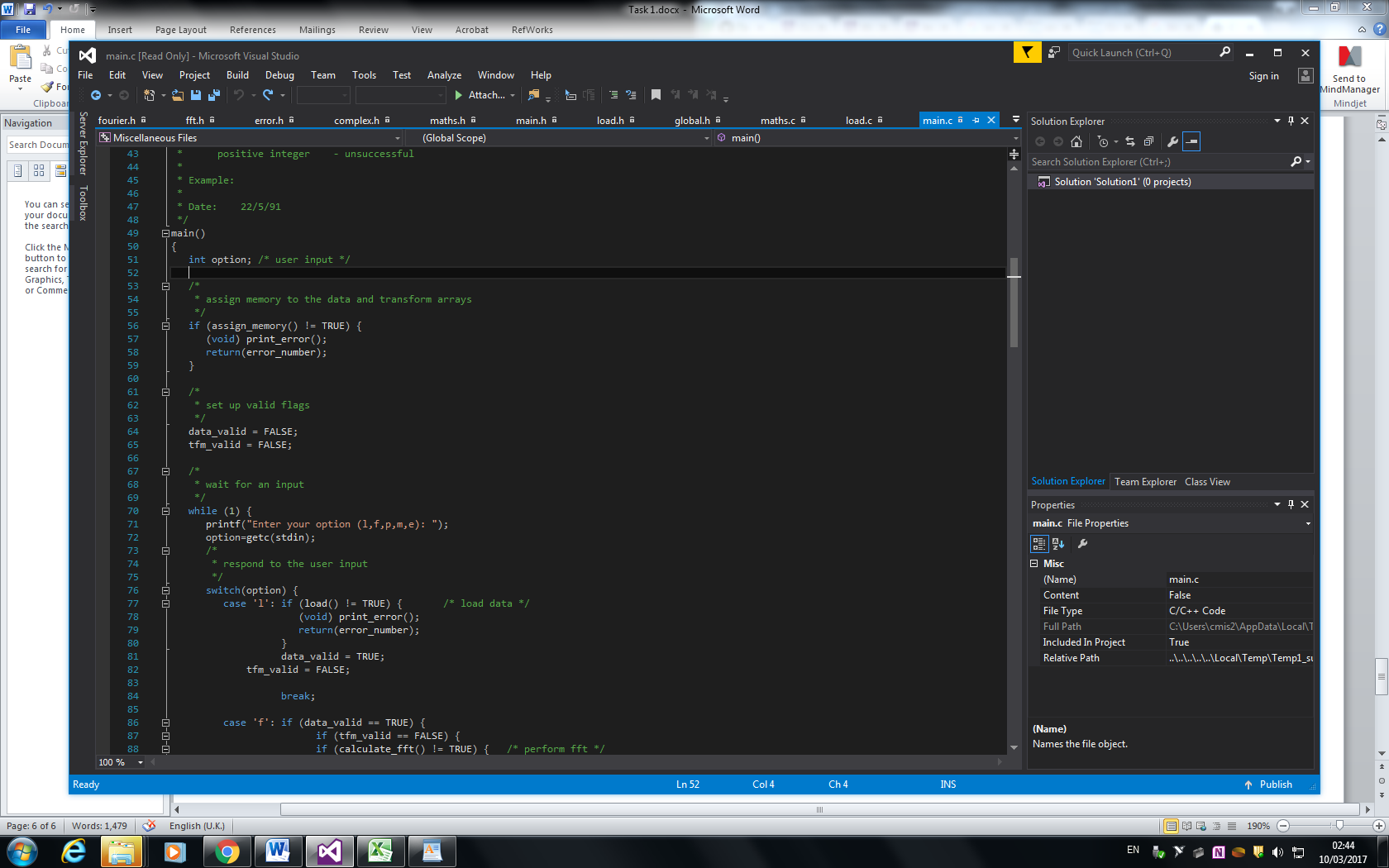
**Bad practice**

1. #include for <stdlib.h> and <malloc.h> in main.c (lines 160 & 161)hh are contained in the middle of the code instead of being included at the start with the other #includes. This is not an issue that can affect performance but is not considered to be good practice.



**Bad practice**

1. main.c allocates memory for the texture assessment (lines 56-58). This way of memory allocation can produce errors when the allocated memory amount is exceeded.



Since memory isn’t deallocated afterwards, this method can produce errors when the functionality of the code expands.

1. The file containing the code did not appear to contain a README file of any sort. Meaning that beyond the comments in the code, there was little in form of a description about the functionality of each function block.

**Conclusion**

An example code for Talysurf texture assessment device was examined in context of software testing, quality and reliability. It was discovered that the code does meet the functional requirements and is acceptable in terms of non-functional requirements.

Some potential sources of faults were discovered stemming from the way some variables were declared, as well as handling of user input and allocating memory.