The University of Queensland
School of Electrical Engineering and Computer Science

CSSE7231 – Semester 1, 2025
Assignment 1 (Version 1.0)

Marks: 75
Weighting: 15%
Due: 3:00pm Friday 28 March, 2025

Introduction

The goal of this assignment is to give you practice in C programming. You will build on this ability in the remainder of the course (and subsequent programming assignments will be more difficult than this one). You are to create a program called `uqexpr` (shorthand for "UQ Expression") that will evaluate expressions and assignment operations entered on `stdin` or from a file. The assignment will also test your ability to code to a particular programming style guide and to use a revision control system appropriately.

Student Conduct

This is an individual assignment. You should feel free to discuss **general** aspects of C programming and the assignment specification with fellow students, including on the discussion forum. In general, questions like "How should the program behave if ⟨this happens⟩?" would be safe, if they are seeking clarification on the specification.

You must not actively help (or seek help from) other students or other people with the actual design, structure and/or coding of your assignment solution. It is **cheating to look at another person's assignment code** and it is **cheating to allow your code to be seen or shared in printed or electronic form by others**. All submitted code will be subject to automated checks for plagiarism and collusion. If we detect plagiarism or collusion, formal misconduct actions will be initiated against you, and those you cheated with. That's right, if you share your code with a friend, even inadvertently, then **both of you are in trouble**. Do not post your code to a public place such as the course discussion forum or a public code repository. (Code in private posts to the discussion forum is permitted.) You must assume that some students in the course may have very long extensions so do not post your code to any public repository until at least three months after the result release date for the course (or check with the course coordinator if you wish to post it sooner). Do not allow others to access your computer – you must keep your code secure. Never leave your work unattended.

You must follow the following code usage and referencing rules for **all code committed to your SVN repository** (not just the version that you submit):

Code Origin	Usage/Referencing
Code provided by teaching staff this semester Code provided to you in writing **this semester** by CSSE7231 teaching staff (e.g., code hosted on Blackboard, found in `/local/courses/csse2310/resources` on `moss`, posted on the discussion forum by teaching staff, provided in Ed Lessons, or shown in class).	**Permitted** May be used freely without reference. (You must be able to point to the source if queried about it – so you may find it easier to reference the code.)
Code you wrote this semester for this course Code you have personally written this semester for CSSE7231 (e.g. code written for A1 reused in A3) – provided you have not shared or published it.	**Permitted** May be used freely without reference. (This assumes that no reference was required for the original use.)

Code Origin	Usage/Referencing
Unpublished code you wrote earlier Code you have personally written in a previous enrolment in this course or in another UQ course or for other reasons and where that code has not been shared with any other person or published in any way.	**Conditions apply, references required** May be used provided you understand the code AND the source of the code is referenced in a comment adjacent to that code (in the required format – see the style guide). If such code is used without appropriate referencing then this will be considered misconduct.
Code from `man` pages on `moss` Code examples found in `man` pages on `moss`. (This does not apply to code from `man` pages found on other systems or websites unless that code is also in the `moss` `man` page.)	
Code and learning from AI tools Code written by, modified by, debugged by, explained by, obtained from, or based on the output of, an artificial intelligence tool or other code generation tool that you alone personally have interacted with, without the assistance of another person. This includes code you wrote yourself but then modified or debugged because of your interaction with such a tool. It also includes code you wrote where you learned about the concepts or library functions etc. because of your interaction with such a tool. It also includes where comments are written by such a tool – comments are part of your code.	**Conditions apply, references & documentation req'd** May be used provided you understand the code AND the source of the code or learning is referenced in a comment adjacent to that code (in the required format – see the style guide) AND an ASCII text file (named `toolHistory.txt`) is included in your repository and with your submission that describes in detail how the tool was used. (All of your interactions with the tool must be captured.) The file must be committed to the repository at the same time as any code derived from such a tool. If such code is used without appropriate referencing and without inclusion of the `toolHistory.txt` file then this will be considered misconduct. See the detailed AI tool use documentation requirements on Blackboard – this tells you what must be in the `toolHistory.txt` file.
Code copied from sources not mentioned above Code, in any programming language: • copied from any website or forum (including StackOverflow and CSDN); • copied from any public or private repositories; • copied from textbooks, publications, videos, apps; • copied from code provided by teaching staff only in a previous offering of this course (e.g. previous assignment one solution); • written by or partially written by someone else or written with the assistance of someone else (other than a teaching staff member); • written by an AI tool that you did not personally and solely interact with; • written by you and available to other students; or • from any other source besides those mentioned in earlier table rows above.	**Prohibited** May **not** be used. If the source of the code is referenced adjacent to the code then this will be considered code without academic merit (not misconduct) and will be removed from your assignment prior to marking (which may cause compilation to fail and zero marks to be awarded). Copied code without adjacent referencing will be considered misconduct and action will be taken. This prohibition includes code written in other programming languages that has been converted to C.
Code that you have learned from Examples, websites, discussions, videos, code (in any programming language), etc. that you have learned from or that you have taken inspiration from or based any part of your code on but have not copied or just converted from another programming language. This includes learning about the existence of and behaviour of library functions and system calls that are not covered in class.	**Conditions apply, references required** May be used provided you do not directly copy code AND you understand the code AND the source of the code or inspiration or learning is referenced in a comment adjacent to that code (in the required format – see the style guide). If such code is used without appropriate referencing then this will be considered misconduct.

You must not share this assignment specification with any person (other than course staff), organisation, website, etc. Uploading or otherwise providing the assignment specification or part of it to a third party including online tutorial and contract cheating websites is considered misconduct. The university is aware of many of these sites and many cooperate with us in misconduct investigations. You are permitted to post small extracts of this document to the course Ed Discussion forum for the purposes of seeking or providing clarification on this specification.

In short – **Don't risk it!** If you're having trouble, seek help early from a member of the teaching staff. Don't be tempted to copy another student's code or to use an online cheating service. Don't help another CSSE2310/7231 student with their code no matter how desperate they may be and no matter how close your

relationship. You should read and understand the statements on student misconduct in the course profile and on the school website: `https://eecs.uq.edu.au/current-students/guidelines-and-policies-students/student-conduct`.

Specification

The `uqexpr` program evaluates expressions and assignment operations entered on `stdin` or from a file.

Expressions

An expression is a combination of variables, numbers and operators that are evaluated to a single number. For example, `sin(3.14/variable)` is an expression that equals approximately 0 if `variable = 1`, and 1 if `variable = 2`.

Assignment Operations

Assignment operations have a single equals sign (`=`) and a single variable on the left-hand side (LHS) of the equals sign and an expression on the right-hand side (RHS) of the equals sign. The expression is evaluated and the variable is assigned the result. For example, `E = m*c^2` is an assignment operation. If `m = 2` and `c = 3*10^8`, then E is assigned the value `18*10^16`.

Command Line Arguments

Your `uqexpr` program is to accept command line arguments as follows:

`./uqexpr [--sigfigures` *`numsigfigs`*`] [--forloop` *`string`*`] [--def` *`string`*`] [`*`inputfilename`*`]`

The square brackets (`[]`) indicate optional arguments. The *italics* indicate placeholders for user-supplied value arguments. `uqexpr` expects zero or more option arguments beginning with "`--`" to follow the command name (with an associated value argument after `--def`, `--forloop` and `--sigfigures`). Option arguments can be in any order. A single optional filename for the input file to be read will follow this – if it is absent then expressions and assignment operations will be read from `stdin`. It is acceptable to run `uqexpr` with no command line arguments. It is also possible to provide *`inputfilename`* as an argument without any option arguments.

Some examples of how the program might be run include the following[1]:

```
./uqexpr

./uqexpr ./input_file.txt

./uqexpr --sigfigures 5 toEvaluate.txt

./uqexpr --def ThisIsAVariable=3.14 --forloop ThisIsALoopVariable,5,-1,-5

./uqexpr --sigfigures 7 --forloop x,0,1,5 --def a=-1 --forloop \
loop,10,-1,-1 --def b=3.14 expr.txt
```

The meaning of the arguments is as follows. More details on the expected behaviour of the program are provided later in this document.

- `--def` – this argument specifies the initial value for a given variable. If this argument is present then it must be followed by a string containing a variable name and value pair. A variable name should have an equals sign (`=`) after it, followed by a number to initially set the variable to. For example, to initialise the variable `hello` to the value 1, then the string that follows `--def` should be `hello=1`. The `--def` argument can be specified multiple times on the command line to define multiple variables.
- `--forloop` – this argument specifies that the following variable can be looped through in expressions and assignment operations. Loop variables should have a name and range (i.e. start, increment and end values) separated by commas (`,`). For example, to set the variable `x` to loop from 10 to 5 in increments of -0.25, then the string that follows `--forloop` should be `x,10,-0.25,5`. The `@loop` command can be used to execute loops. The initial value of a newly defined loop variable should be its start value. The `--forloop` argument can be specified multiple times on the command line to define multiple loop variables. Refer to the Range Command and Advanced Functionality sections for more details about the use of loop variables.

[1] This is not an exhaustive list and does not show all possible combinations of arguments. The examples assume the existence of the filenames.

- `--sigfigures` – this single digit argument specifies the maximum number of significant figures of numerical values printed to `stdout`. For example, if `--sigfigures 5` is specified, then printed numbers will have at most five significant figures. If `--sigfigures` is not specified, then the default maximum number of significant figures (`3`) will be used. `--sigfigures` can only be specified once on the command line. You can print out a floating point number to a specified maximum number of significant figures using the `printf()` or `fprintf()` format specifier `%.*g`. The first argument is the maximum number of significant figures and the second argument is the value to print.
- *`inputfilename`* – a filename specified at the end of the command line is the name of an input file containing expressions and assignment operations to be evaluated. If present, it is always specified last. If a filename is not specified, then expressions and assignment operations will be read from `stdin`. The *`inputfilename`* string cannot begin with "`--`". If the user wants to specify a file whose name does start with "`--`" then they should prefix the name with "`./`".

Before doing anything else, your program must check the command line arguments for validity. If the program receives an invalid command line then it must print the following message:

```
Usage: ./uqexpr [--sigfigures 2..9] [--forloop string] [--def string] [inputfilename]
```

to standard error (with a following newline), and exit with an exit status of 12.

Refer to the Command Line and Variable Checking section for information on how to determine the other rules of command line checking.

Checking whether an input file exists or can be opened is not part of the usage checking (other than checking that *`inputfilename`* is not an empty string). Checking the contents/format of the strings following `--forloop` and/or `--def` is not part of the usage checking (other than checking that the strings are not empty). These are checked after command line validity as described below.

File Checking

If an input filename is provided on the command line, your program must check that it can be opened for reading. If it cannot be opened for reading, then your program must print the message:

```
uqexpr: unable to read from input file "filename"
```

to standard error (with a following newline) where *`filename`* is replaced by the name of the file from the command line. The double quotes must be present. Your program must then exit with an exit status of 19.

Variable Checking

If the usage check and input file check (if required) described above are successful, then your program must check the value of arguments that follow `--def` and/or `--forloop`. The requirements for valid variables are:

- A string after `--def` must be in the form `name=value`, i.e. variable names and values should be separated by an equals sign `(=)`. The value is treated as a double precision floating point number.
- A string after `--forloop` must be in the form `name,start,increment,end`, i.e. the start value, increment value and end value are specified after the loop variable name and are comma (`,`) separated. These values should be treated as numbers of type `double`. Refer to the Hints section for a function that can be used to convert a string to `double`. The increment value is used in each loop iteration to update the loop variable and therefore cannot be 0. The increment value should be positive if the start value is less than the end value. The increment value should be negative if the start value is greater than the end value. It is permissible for the start value to equal the end value, in which case the increment value can be any number other than 0.
- Variable names must contain only letters (a to z, A to Z) and must have a length between 1 and `20` letters inclusive.
- See the Command Line and Variable Checking section below for more information.

If any of the above requirements are found to be violated whilst checking variables, then your program must print the message:

```
uqexpr: invalid variable(s) specified on the command line
```

to standard error (with a following newline). Your program must then exit with an exit status of 4.

After confirming the validity of all variables, if it is discovered that any variables have the same name, then your program must print the message:

```
uqexpr: one or more variables are duplicated
```

to standard error (with a following newline). Your program must then exit with an exit status of 18.

Command Line and Variable Checking

We have deliberately omitted some requirements for checking the arguments and variables specified on the command line. Please run the demo program (`demo-uqexpr`) and observe the behaviour of the program to answer the following questions.

- Are variable names case sensitive?
- What range of values is permitted to follow the `--sigfigures` option on the command line? (Hint: see the usage error message above.)
- What happens if the *`inputfilename`* string begins with "`--`"?
- What happens if an empty string is given as a command line argument (in any position)?
- What happens if the `--sigfigures` argument is given more than once on the command line?
- What happens if an argument is given after any *`inputfilename`*?
- What happens if an option argument (`--def`, `--forloop` or `--sigfigures`) is not followed by an option value argument?
- What happens if an unexpected argument appears on the command line?

Program Behaviour

Program Startup

If the above checks are successful, your program must print the following message to standard output (`stdout`), with a newline at the end of each line:

```
Welcome to uqexpr!
This program was written by s4828041.
```

Your program must then print the initialised variables that were identified and their value, each separated by a newline. The value should be formatted using the maximum number of significant figures specified by the `--sigfigures` option argument. If `--sigfigures` was not specified on the command line, then the default value (`3`) should be used. For example, if `--def hello=1 --def world=2` is specified on the command line, this message must be printed:

```
Variables:
hello = 1
world = 2
```

If no initialised variables were specified, then this message must be printed instead:

```
There are no variables.
```

Your program must then print the loop variables that were identified, each separated by a newline. The current value, start, increment and end values should be formatted using the maximum number of significant figures specified by the `--sigfigures` option argument. If `--sigfigures` was not specified on the command line, then the default value (`3`) should be used. For example, if `--forloop x,0,1,10 --forloop y,10,-0.123456789,5 --sigfigures 5` is specified on the command line, this message must be printed:

```
Loop variables:
x = 0 (0, 1, 10)
y = 10 (10, -0.12346, 5)
```

If no loop variables were specified, then this message must be printed instead:

```
No loop variables were found.
```

After variables are reported, the following message must be printed if no input file was specified on the command line, i.e. lines are to be read from `stdin`:

```
Submit your expressions and assignment operations to be evaluated.
```

Expression and Assignment Operation Validation and Evaluation

Your program must then repeatedly read lines from the input file (or `stdin` if no input file was specified). Each line is terminated by a newline character or pending EOF. Lines beginning with a hash symbol (`#`) are considered comments and ignored. The line (excluding any terminating newline) will then be evaluated for its validity. If the line entered is not a valid expression or assignment operation, your program must print the following message (terminated by a newline) to `stderr` and attempt to read another line from the input file or `stdin`:

```
Error in command, expression or assignment operation detected
```

The rules for valid expressions are:

- No equals sign (`=`) present in the line read.
- Able to be compiled by the `tinyexpr` library without error. See the Provided Libraries section below for more details about the `tinyexpr` library. The expression should be compiled using all validated loop and non-loop variables.

The rules for valid assignment operations are:

- One equals sign (`=`) present in the line read.
- The RHS of the assignment operation (i.e. the part after the equals sign) must be able to be compiled by the `tinyexpr` library without error.
- The LHS of the assignment operation should contain a single variable name and no other variable names, numbers or operators. Whitespace characters are permissible before and after the variable name. If a variable name is valid and is not an existing variable, then it is to be defined as a new non-loop variable and its value is to be set to the result after the RHS is evaluated by the `tinyexpr` library. Refer to the Hints section for more details.

If an expression or assignment operation is found to be valid, it should then be evaluated by the `tinyexpr` library. For expressions, your program must print the following message:

```
Result = value
```

to standard output (`stdout`) with a following newline, where *`value`* is replaced by the result after evaluating the expression and is formatted according to the `--sigfigures` option argument.

For assignment operations, your program must evaluate the expression on the RHS of the assignment operation and then store the result in the memory location of the variable on the LHS. Your program must print the following message:

```
var = value
```

to standard output (`stdout`) with a following newline, where *`var`* is replaced by the variable name and *`value`* is replaced by the result after evaluating the expression and is formatted according to the `--sigfigures` option argument. Assignment operations entered by the user can assign values to variables that are dependent on the values of other variables, e.g. `y = 2 + x` and `x = 2 * x` are valid assignment operations.

Your program must then attempt to read another expression or assignment operation from the file or `stdin` and repeat the above process.

Exiting the Program

If end-of-file (EOF[2]) is detected on `stdin` when your program goes to read a line, your program must print the following message to `stdout` (with a terminating newline):

```
Thanks for using uqexpr!
```

and exit with exit status 0.

[2]EOF is "end of file" – the stream being read (standard input in this case) is detected as being closed. Where standard input is connected to a terminal (i.e. a user interacting with a program) then the user can close the program's standard input by pressing Ctrl-D (i.e. hold the Control key and press D). This causes the terminal driver which receives that keystroke to close the stream to the program's standard input. Note that an expression or assignment operation might be terminated by EOF rather than newline. Such an entry must be treated as if the line was followed by a newline. EOF will be detected on the subsequent attempt to read a line.

Print Command

The `@print` command can be used to print the name of all non-loop variables and their current values and then all loop variables with their current values and range to `stdout`. There should be no leading or trailing characters (including whitespace) before or after the command, respectively. For example, suppose the user enters the following command:

```
./uqexpr --sigfigures 2 --forloop y,0,1,10 --forloop x,10,-0.25,5 \
--def hello=5 --def HeLLo=3.14
```

If the `@print` command is specified by the user via `stdin` or the input file, the following lines should be printed to `stdout`:

```
Variables:
hello = 5
HeLLo = 3.1
Loop variables:
y = 0 (0, 1, 10)
x = 10 (10, -0.25, 5)
```

Note that "`There are no variables.`" and "`No loop variables were found.`" lines should be printed instead of "`Variables:`" and "`Loop variables:`" if no non-loop or loop variables have been defined, respectively.

Range Command

The `@range` ***var*** command allows the user to define additional loop variables or redefine the range of existing loop variables. There should be no leading or trailing characters (including whitespace) before or after the command, respectively. There should be a single space between `@range` and ***var***. ***var*** is replaced by a string that follows the same format as the string that follows `--forloop` on the command line and should be subject to the same checks, e.g. variable names must be letters only. For example, to set the variable `y` to loop from 10 to 5 in increments of -0.25, then the command entered via `stdin` or within the input file should be:

```
@range y,10,-0.25,5
```

Assuming that `--sigfigures 2` was specified on the command line, the following line should be printed to `stdout`, containing the name and range of `y`:

```
y = 10 (10, -0.25, 5)
```

Note that since the range of existing loop variables can be redefined, it is permissible for existing loop variable names to be specified in a `@range` ***var*** command. It is also valid to convert an existing non-loop variable to a loop variable[3]. No loop variables are (re)defined if any errors are found in the ***vars*** argument. If the command is valid, the value of the variable is set to be the start value of the range. If the command entered is invalid, your program must print the following message (terminated by a newline) to `stderr` and attempt to read another line from the input file or `stdin`:

```
Error in command, expression or assignment operation detected
```

Advanced Functionality

This section describes more advanced functionality which involves implementing loop variable looping. It is recommended that you not implement this until other functionality is working. Loop variables can be looped through using the command `@loop` ***var expression***, where ***var*** is replaced by the name of a loop variable and ***expression*** is replaced by an expression or assignment operation. There should be no leading characters (including whitespace) before the command. There should be a single space between `@loop`, ***var*** and ***expression***. The specified loop variable must be looped through based on its start, increment and end values and the specified expression or assignment operation must be evaluated for each loop iteration. At the start of a loop, the loop variable is initialised to the start value. ***expression*** is evaluated and the loop value is incremented. This process repeats until the loop variable reaches its final value, which should be $\leq$ the end value if the increment is positive or $\geq$ the end value if the increment is negative. For example, if the string that follows `--forloop` on the command line is `y,10,-2.2,5`, then all expressions/assignment operations specified in the ***expression*** section of the `@loop y` ***expression*** command must be evaluated when $y = 10.0$, $y = 7.8$ and $y = 5.6$. It is recommended that you run the demo program (`demo-uqexpr`) to see how looping through a loop variable should work.

[3] It is not possible to convert a loop variable back to a non-loop variable.

For assignment operations specified in the `@loop` command, your program must evaluate the expression on the RHS of the assignment operation and then store the result in the memory location of the variable on the LHS. For both expressions and assignment operations specified in the `@loop` command, your program must print their respective evaluation messages described previously with the following string appended to the end of each message before the newline:

`when` ***`var`*** `=` ***`value`***

where ***`var`*** is replaced by the loop variable name and ***`value`*** is replaced by its respective value for that iteration and is formatted according to the `--sigfigures` option argument.

If any errors are detected while evaluating the `@loop` command (e.g. non-existent loop variable or invalid expression), your program must print the following message (terminated by a newline) to `stderr` and attempt to read another line from the input file or `stdin`:

```
Error in command, expression or assignment operation detected
```

Other Requirements

Your program must free all allocated memory before exiting, including if it exits due to a usage, file or variable error. (Your program does not have to free memory if it exits due to a signal, e.g. the interrupt signal generated by pressing Ctrl-C.)

For any aspects of behaviour not described in this specification, your program must behave in the same manner as `demo-uqexpr`. If you're unsure about some aspect, then you can run the demo program to check the expected behaviour.

We will not test for unexpected system call or library failures in an otherwise correctly implemented program (e.g. if `malloc()` fails due to insufficient available resources). We will not test your program by defining variables that have the same name as `tinyexpr` functions or constants, e.g. `sin`, `cos`, `tan`, `sqrt`, `e` and `pi`. Your program can behave however it likes in these cases, including crashing.

Example Sessions

Some example interactions with `uqexpr` are shown below. Lines typed by the user (i.e. the command and text entered on standard input) are shown in **bold green** for clarity. Lines printed to standard error are shown in **bold red**. Other lines are printed to standard output. Note that the $ character is the shell prompt – it is not entered by the user nor output by `uqexpr`.

Example 1: Example `uqexpr` session with only `--sigfigures` specified on the command line and demonstrating the use of the `@print` command. Note that the user presses Ctrl-D (to terminate input – detected by the program as EOF) just before the line "`Thanks for using uqexpr!`" is printed.

```
$ ./uqexpr --sigfigures 3
Welcome to uqexpr!
This program was written by s4828041.
There are no variables.
No loop variables were found.
Submit your expressions and assignment operations to be evaluated.
hello
Error in command, expression or assignment operation detected
z = 1
z = 1
# comments like this are ignored and whitespace is ignored in the expression below
      23    +          10
Result = 33
e
Result = 2.72
# Euler's Number approximation
eApprox = 1 + 1/1 + 1/(1*2) + 1/(1*2*3) + 1/(1*2*3*4) + 1/(1*2*3*4*5) + 1/(1*2*3*4*5*6)
Result = 2.72
pi
Result = 3.14
VarOne = sin(pi)
```

```
VarOne = 1.22e-16
VarTwo = sin(pi/2)
VarTwo = 1
@print
Variables:
z = 1
eApprox = 2.72
VarOne = 1.22e-16
VarTwo = 1
No loop variables were identified.
Thanks for using uqexpr!
$ echo $?
0
$
```

Example 2: Example `uqexpr` session with `--sigfigures`, initialised variables and an input file. Notice that the variables a, b, c, V and apparentTemperature are set dynamically within the input file. The variables ambientTemperature, humidity and windSpeed are set on the command line, so a different apparentTemperature is calculated when their specified values change.

```
$ cat inputFile
# comment to be ignored
this is an invalid expression
# set these variables
a = 1
b = 2
c = 3
# volume of an ellipsoid
V = (4/3) * pi * a * b * c
# calculate apparent temperature
apparentTemperature = ambientTemperature + 0.0201465 * humidity * exp(17.27 *
    ambientTemperature / (237.7 + ambientTemperature)) - (0.7 * windSpeed) - 4
$ ./uqexpr --sigfigures 5 --def ambientTemperature=29.0 \
           --def humidity=49.5 --def windSpeed=0.1 inputFile
Welcome to uqexpr!
This program was written by s4828041.
Variables:
ambientTemperature = 29
humidity = 49.5
windSpeed = 0.1
No loop variables were found.
Error in command, expression or assignment operation detected
a = 1
b = 2
c = 3
V = 25.133
apparentTemperature = 31.452
Thanks for using uqexpr!
$ ./uqexpr --sigfigures 5 --def ambientTemperature=-1.6 \
           --def humidity=66 --def windSpeed=5 inputFile
Welcome to uqexpr!
This program was written by s4828041.
Variables:
ambientTemperature = -1.6
humidity = 66
windSpeed = 5
No loop variables were found.
Error in command, expression or assignment operation detected
a = 1
```

```
b = 2
c = 3
V = 25.133
apparentTemperature = -7.9172
Thanks for using uqexpr!
```

Example 3: Example `uqexpr` session with `--forloop`, `--def` and `--sigfigures` arguments specified on the command line and the use of `@range`, `@print` and `@loop` commands. A loop variable is used to perform integration. The factorial of n can be calculated using $\int_0^1 (-\ln x)^n \, dx$. This integral will be evaluated to find the factorial of 5, which we expect to be 120. Note that `x` is set to 1 initially and the `@range` command changes its value to 0.000001 (1e-06). After the loop, the value of `x` is 0.999999 instead of 1 due to a floating point rounding issue. During the loop, each iteration uses and updates the current value of `sum`.

```
$ ./uqexpr --forloop x,1,1,3 --def sum=0 --def n=5 --sigfigures 7
Welcome to uqexpr!
This program was written by s4828041.
Variables:
sum = 0
n = 5
Loop variables:
x = 1 (1, 1, 3)
Submit your expressions and assignment operations to be evaluated.
@range y,0.25,0.25,1
y = 0.25 (0.25, 0.25, 1)
@range x,0.000001,0.000001,1
x = 1e-06 (1e-06, 1e-06, 1)
@loop x sum = sum + ((-ln(x))^n) * 0.000001
sum = 0.503309 when x = 1e-06
sum = 0.8924082 when x = 2e-06
sum = 1.224996 when x = 3e-06
sum = 1.521628 when x = 4e-06
sum = 1.792572 when x = 5e-06
sum = 2.043877 when x = 6e-06
sum = 2.279479 when x = 7e-06
sum = 2.502125 when x = 8e-06
sum = 2.713819 when x = 9e-06
sum = 2.916088 when x = 1e-05
[Output is hidden for brevity]
sum = 119.5148 when x = 0.99999
sum = 119.5148 when x = 0.999991
sum = 119.5148 when x = 0.999992
sum = 119.5148 when x = 0.999993
sum = 119.5148 when x = 0.999994
sum = 119.5148 when x = 0.999995
sum = 119.5148 when x = 0.999996
sum = 119.5148 when x = 0.999997
sum = 119.5148 when x = 0.999998
sum = 119.5148 when x = 0.999999
@print
Variables:
sum = 119.5148
n = 5
Loop variables:
x = 0.999999 (1e-06, 1e-06, 1)
y = 0.25 (0.25, 0.25, 1)
```

Provided Libraries

libtinyexpr

The `tinyexpr` library has been pre-compiled into a shared library for you, providing all of the functions and data types required.

These functions are declared in `/local/courses/csse2310/include/tinyexpr.h` on moss. To use them, you will need to add `#include <tinyexpr.h>` to your code and use the compiler flag `-I/local/courses/csse2310/include` when compiling your code so that the compiler can find the include file. You will also need to link with the library containing these functions. To do this, use the compiler arguments `-L/local/courses/csse2310/lib -ltinyexpr`

Note that the `tinyexpr` library uses the C math library, so you will also need to link your programs with the `-lm` option.

Important note: **You must use the version of `tinyexpr.h` and the pre-compiled library provided for you on `moss`** – do not download and use the `tinyexpr.c` and `tinyexpr.h` from GitHub. The original version will generate style warnings that will cost you marks, and we will instrument `libtinyexpr` to support the marking process. **Failure to use the provided version will cause your marking tests to fail**.

Using the `tinyexpr` library

`tinyexpr` is a small C library for parsing and evaluating mathematical expressions. For example, you can use `tinyexpr` to evaluate the function "`sqrt(sin(x+y+z)+1)`" using C calls. This will give your programs the ability to work with arbitrary mathematical functions. There a few steps to using `tinyexpr`, as illustrated below.

Example 4: `tinyexpr` usage example using `argv` to dynamically define arbitrary variables and evaluate an arbitrary expression.

```c
#include <tinyexpr.h>
#include <stdio.h>

int main(int argc, char* argv[])
{
    argc--;
    argv++;
    if (argc < 1) {
        fprintf(stderr, "Usage: ./tinyexprtest expression [variables]\n");
        return 1;
    }
    // Create arrays of type te_variable and double
    te_variable teVars[argc - 1];
    double values[argc - 1];
    // Loop through each variable and initialise it
    for (int i = 1; i < argc; i++) {
        values[i - 1] = i;
        te_variable var = {.name = argv[i], .address = &(values[i - 1]), .type = TE_VARIABLE,
                .context = NULL};
        teVars[i - 1] = var;
        printf("Created new var \"%s\" with value %.3g\n", argv[i], values[i - 1]);
    }
    int errPos;
    // Parse and compile the expression, passing the vars, var count and ptr to an error return
    te_expr* expr = te_compile(argv[0], teVars, argc - 1, &errPos);
    // expr is not NULL if compilation succeeded
    if (expr) {
        // Evaluate the expression
        printf("%s = %.3g\n", argv[0], te_eval(expr));
        // Free the memory associated with the expression
        te_free(expr);
    } else {
        printf("Error parsing expression \"%s\" at character %d\n", argv[0], errPos);
        return 2;
    }
    return 0;
}
```

Let's compile the program and then run:

```
./tinyexprtest "sqrt(sin(x+y+z)+1)" x y z
```

This will produce the following output:

```
Created new var "x" with value 1
Created new var "y" with value 2
Created new var "z" with value 3
sqrt(sin(x+y+z)+1) = 0.849
```

A few things to note from the above example:

- `te_variable` variables must be bound to a memory location that can hold a double precision floating point number.
- After variable binding, `te_compile` is used to compile the expression string for later evaluation.
- If compilation fails, `te_compile` will return NULL and the `errPos` variable will be updated to the character number in the string where the parse error occurred.
- After parsing and compilation, updates to the original variables (`x`, `y` and `z` in this example) will be reflected on each call to `te_eval()`.
- After you are finished with an expression, release the memory by calling `te_free()`.

This little introduction should be enough for the use of `tinyexpr` required in this assignment, however, you may also refer to the source code and more complex examples at the project GitHub repository if you wish: `https://github.com/codeplea/tinyexpr`

Style

Your program must follow version 3 of the CSSE2310/CSSE7231 C programming style guide available on the course Blackboard site. Your submission must also comply with the *Documentation required for the use of AI tools* (if applicable).

Hints

1. Run the demo program to see how your program should behave. If your program exactly mimics the demo program then you should receive full marks for functionality.
2. Code from the Ed Lessons C exercises may be useful to you.
3. The string representation of a single digit positive integer has a string length of 1.
4. You **may** wish to consider the use of the standard library functions `isspace()` and/or `isdigit()`. Note that these functions operate on individual characters, represented as integers (ASCII values).
5. Some other functions which **may** be useful include: `strtod()`, `strcmp()`, `strncmp()`, `strlen()`, `strdup()`, `strstr()`, `strtok()`, `exit()`, `fopen()`, `fprintf()`, and `fgetc()`. You should consult the man pages of these functions.
6. To assist with satisfying the requirement of freeing all allocated memory before exiting, you can use the program `valgrind` with `--leak-check=full` and `--show-leak-kinds=all` arguments, i.e. run `valgrind --leak-check=full --show-leak-kinds=all ./uqexpr`. Ensure that you compile your program with the `gcc -g` flag to allow `valgrind` to tell you which specific lines of your program are leading to memory issues. You can be confident that your program is freeing all memory for a particular execution scenario when you see the following message in `valgrind`: `All heap blocks were freed -- no leaks are possible`. Use of `valgrind` will be covered in more detail in the week 4 contact.
7. You **may** wish to consider using a function from the `getopt()` family to parse command line arguments. This is **not** a requirement and if you do so your code is unlikely to be any simpler or shorter than if you don't use such a function. You may wish to consider it because it gives you an introduction to how programs can process more complicated combinations of command line arguments. See the `getopt(3)` man page for details. To allow for the use of a `getopt()` family function, we will not test your program's behaviour with the argument "`--`" (which is interpreted by `getopt()` as a special argument that indicates the end of option arguments). We also won't test abbreviated arguments.

Possible Approach

It is suggested that you write your program using the following steps. Test your program at each stage and commit to your SVN repository frequently. Note that the specification text above and the behaviour of the demo program are the definitive descriptions of the expected program behaviour. The list below does not cover all required functionality.

1. Write code to iterate over the command line arguments and check for `--def`, `--forloop` and `--sigfigures` and save their associated strings. Check that the `--sigfigures` argument (if specified) is valid. Detect whether an input filename was provided. Print a usage error message and exit with status 12 if there are any errors in the command line.
2. Check that the input file can be opened for reading. Exit appropriately if not.
3. Check that all variables (if specified) are valid. Exit appropriately if not.
4. Write code that prints the startup messages and repeatedly reads a line from `stdin` or the input file (if provided) until EOF is received. Make sure that you ignore or strip off the newline of each line read.
5. Add checks for the expression or assignment operation that is read as described in the Expression and Assignment Operation Validation and Evaluation section on page 5. Print the appropriate message if an invalid expression or assignment operation is detected.
6. Add code that evaluates expressions and assignment operations and prints appropriate messages.
7. Add code to handle the `@print` command.
8. Add code to handle the `@range` command.
9. Add code to handle looping through loop variables.
10. Implement remaining functionality as required ...

Forbidden Functions

You must not use any of the following C statements/directives/etc. If you do so, you will get zero (0) marks for the assignment.

- `goto`
- `#pragma`
- `gcc` attributes (other than the possible use of `__attribute__((unused))` as described in the style guide)

You must not use any of the following C functions. If you do so, you will get zero (0) marks for any test case that calls the function.

- `longjmp()` and equivalent functions
- `system()`
- `popen()`
- `mkfifo()` or `mkfifoat()`
- `fork()`
- `pipe()`
- `rewind()`
- `fseek()`
- `execl()`, `execvp()` or any other members of the exec family of functions
- Functions described in the man page as non-standard, e.g. `strcasestr()`. Standard functions will conform to a POSIX standard – often listed in the "CONFORMING TO" section of a man page.

The use of comments to control the behaviour of `clang-format` and/or `clang-tidy` (e.g., to disable style checking) will result in zero marks for automatically marked style.

Testing

You are responsible for ensuring that your program operates according to the specification. You are encouraged to test your program on a variety of scenarios. You should create your own input file and test your `uqexpr` program with a variety of command lines and input from `stdin`. A variety of programs will be provided to help you in testing:

- A demonstration program (called `demo-uqexpr`) that implements the correct behaviour is available on `moss`. You can test your program with a set of command line arguments and input values and also run the same test with `demo-uqexpr` to check that you get the same output. You can also use `demo-uqexpr` to check the expected behaviour of the program if some part of this specification is unclear.
- A test script will be provided on `moss` that will test your program against a subset of the functionality requirements – approximately 50% of the available functionality marks. The script will be made available about 7 to 10 days before the assignment deadline and can be used to give you some confidence that you're on the right track. The "public tests" in this test script will not test all functionality and you should be sure to conduct your own tests based on this specification. The "public tests" will be used in marking, along with a set of "private tests" that you will not see.
- The Gradescope submission site will also be made available about 7 to 10 days before the assignment deadline. Gradescope will run the test suite immediately after you submit. When this is complete[4] you will be able to see the results of the "public tests". You should check these test results to make sure your program is working as expected. Behaviour differences between `moss` and Gradescope may be due to memory initialisation assumptions in your code, so you should allow enough time to check (and possibly fix) any issues after submission.

Submission

Your submission must include all source and any other required files (in particular you must submit a `Makefile`). Do not submit compiled files (e.g. `.o`, compiled programs) or test input files.

Your program (named `uqexpr`) must build on `moss.labs.eait.uq.edu.au` and in the Gradescope environment with:

```
make
```

Your program must be compiled with `gcc` with at least the following options:

```
-Wall -Wextra -pedantic -std=gnu99
```

You are not permitted to disable warnings or use pragmas to hide them. You may not use source files other than `.c` and `.h` files as part of the build process – such files will be removed before building your program.

If any errors result from the `make` command (i.e. the `uqexpr` executable can not be created) then you will receive 0 marks for functionality (see below). Any code without academic merit will be removed from your program before compilation is attempted (and if compilation fails, you will receive 0 marks for functionality).

Your program must not invoke other programs or use non-standard headers/libraries.

Your assignment submission must be committed to your Subversion repository under

```
svn+ssh://source.eait.uq.edu.au/csse2310-2025-sem1/csse2310-sXXXXXXX/trunk/a1
```

where `sXXXXXXX` is your `moss`/UQ login ID. Only files at this top level will be marked so **do not put source files in subdirectories**. You may create subdirectories for other purposes (e.g. your own test files) but these will not be considered in marking – they will not be checked out of your repository.

You must ensure that all files needed to compile and use your assignment (including a Makefile) are committed and within the `trunk/a1` directory in your repository (and not within a subdirectory) and not just sitting in your working directory. Do not commit compiled files or binaries. You are strongly encouraged to check out a clean copy for testing purposes.

To submit your assignment, you must run the command

```
2310createzip a1
```

[4] Gradescope marking may take only a few minutes or more than 30 minutes depending on the functionality and efficiency of your code.

on `moss` and then submit the resulting zip file on Blackboard (a GradeScope submission link will be made available in the Assessment area on the CSSE2310/7231 Blackboard site)[5]. The zip file will be named

`sXXXXXXX_csse2310_a1_`*`timestamp`*`.zip`

where `sXXXXXXX` is replaced by your moss/UQ login ID and *`timestamp`* is replaced by a timestamp indicating the time that the zip file was created.

The `2310createzip` tool will check out the latest version of your assignment from the Subversion repository, ensure it builds with the command '`make`', and if so, will create a zip file that contains those files and your Subversion commit history and a checksum of the zip file contents. You may be asked for your password as part of this process to check out your submission from your repository. You will be asked to confirm references in your code and also to confirm your use (or not) of AI tools to help you.

You must not create the zip file using some other mechanism and you must not modify the zip file before submission. If you do so, you will receive zero marks. Your submission time will be the time that the file is submitted via GradeScope on Blackboard, and **not** the time of your last repository commit nor the time of the creation of your submission zip file.

Multiple submissions to Gradescope are permitted. We will mark whichever submission you choose to "activate" – which by default will be your last submission, even if that is after the deadline and you made submissions before the deadline. Any submissions after the deadline[6] will incur a late penalty – see the CSSE2310/7231 course profile for details.

Marks

Marks will be awarded for functionality and style and documentation. Marks may be reduced if you attend an interview about your assignment and you are unable to adequately respond to questions – see the CSSE7231 Student Interviews section below.

Functionality (60 marks)

Provided your code compiles (see above) and does not use any prohibited statements/functions (see above), and your zip file has been generated correctly and has not been modified before submission, then you will earn functionality marks based on the number of features your program correctly implements, as outlined below. Not all features are of equal difficulty.

Partial marks will be awarded for partially meeting the functionality requirements. Several tests will be run for each marking category listed below, testing a variety of scenarios. Your mark in each category will be proportional (or approximately proportional) to the number of tests passed in that category.

If your program does not allow a feature to be tested then you will receive 0 marks for that feature, even if you claim to have implemented it. For example, if your program can never validate and store loop variables then we can not determine if your program can correctly loop through loop variables. Your tests must run in a reasonable time frame, which could be as short as a few seconds for usage checking to many tens of seconds when `valgrind` is used to test for memory leaks. If your program takes too long to respond, then it will be terminated and you will earn no marks for the functionality associated with that test.

Exact text matching of output messages is used for functionality marking. Strict adherence to this specification and matching behaviour of the demo program are critical to earn functionality marks.

The markers will make no alterations to your code (other than to remove code without academic merit).

Marks will be assigned in the following categories.

1. Program correctly handles invalid command lines (usage errors) (8 marks)
2. Program correctly handles file errors: inability to read from a file (2 marks)
3. Program correctly detects invalid loop and/or non-loop variables provided on the command line, including in amongst valid loop and non-loop variable definitions (8 marks)
4. Program correctly prints all messages upon startup, which includes correctly reporting all variables defined on the command line (4 marks)

[5] You may need to use scp or a graphical equivalent such as WinSCP, Filezilla or Cyberduck to download the zip file to your local computer and then upload it to the submission site.

[6] or your extended deadline if you are granted an extension.

5. Program operates correctly with no command line arguments, dynamically assigned variables or loops (i.e. expression operations but no assignment operations) (3 marks)
6. Program operates correctly with an input file specified on the command line (and no other command line arguments, dynamically assigned variables or loops) (2 marks)
7. Program operates correctly when non-loop variables are dynamically defined and when given the `--def` argument and no others, and none or one input file (6 marks)
8. Program operates correctly when given the `--sigfigures` argument and none or one input file (and no other command line arguments, dynamically assigned variables or loops) (4 marks)
9. Program operates correctly when given the `@print` command (3 marks)
10. Program operates correctly when given multiple command line arguments and commands, dynamically assigned variables and none or one input file (and no loops) (6 marks)
11. Program operates correctly when loop variables are dynamically defined (or redefined) using the `@range` command, loops are run using the `@loop` command, and when given the `--forloop` argument and no others, and none or one input file (8 marks)
12. Program operates correctly and frees all memory upon exit (for a variety of scenarios tested above) (6 marks)

Some functionality may be assessed in multiple categories. For example, if your program can't correctly print the startup messages then it will fail most tests.

Style Marking

Style marking is based on the number of style guide violations, i.e. the number of violations of version 3 of the CSSE2310/CSSE7231 C Programming Style Guide (found on Blackboard). Style marks will be made up of two components – automated style marks and human style marks. These are detailed below. Your style marks can never be more than your functionality mark – this prevents the submission of well styled programs which don't meet at least a minimum level of required functionality.

You should pay particular attention to commenting so that others can understand your code. The marker's decision with respect to commenting violations is final – it is the marker who has to understand your code.

You are encouraged to use the `2310reformat.sh` and `2310stylecheck.sh` tools installed on `moss` to correct and/or check your code style before submission. The `2310stylecheck.sh` tool does not check all style requirements, but it will determine your automated style mark (see below). Other elements of the style guide are checked by humans.

All `.c` and `.h` files in your submission will be subject to style marking. This applies whether they are compiled/linked into your executable or not[7].

Automated Style Marking (5 marks)

Automated style marks will be calculated over all of your `.c` and `.h` files as follows. If any of your submitted `.c` and/or `.h` files are unable to be compiled by themselves then your automated style mark will be zero (0). If your code uses comments to control the behaviour of `clang-format` and/or `clang-tidy` then your automated style mark will be zero. If any of your source files contain C functions longer than 100 lines of code[8] then your automated and human style marks will both be zero. If you use any global variables then your automated and human style marks will both be zero.

If your code does compile and does not contain any C functions longer than 100 lines and does not use any global variables and does not interfere with the expected behaviour of `clang-format` and/or `clang-tidy` then your automated style mark will be determined as follows: Let

- W be the total number of distinct compilation warnings recorded when your `.c` files are individually built (using the correct compiler arguments)

[7] Make sure you remove any unneeded files from your repository, or they will be subject to style marking.

[8] Note that the style guide requires functions to be 50 lines of code or fewer. Code that contains functions whose length is 51 to 100 lines will be penalised somewhat – one style violation (i.e. one mark) per function. Code that contains functions longer than 100 lines will be penalised very heavily – no marks will be awarded for human style or automatically marked style.

- A be the total number of style violations detected by `2310stylecheck.sh` when it is run over each of your `.c` and `.h` files individually[9].

Your automated style mark S will be

$$S = 5 - (W + A)$$

If $W + A \geq 5$ then S will be zero (0) – no negative marks will be awarded. If you believe that `2310stylecheck.sh` is behaving incorrectly or inconsistently then please bring this to the attention of the course coordinator prior to submission, e.g., it is possible the style checker may report different issues on moss than it does in the Gradescope environment. Your automated style mark can be updated if this is deemed to be appropriate. You can check the result of Gradescope style marking soon after your Gradescope submission – when the test suite completes running.

Human Style Marking (5 marks)

The human style mark (out of 5 marks) will be based on the criteria/standards below for "comments", "naming" and "modularity". Note that if your code contains any functions longer than 100 lines or uses a global variable then your human style mark is zero and the criteria/standards below are not relevant.

The meanings of words like *appropriate* and *required* are determined by the requirements in the style guide.

Comments (3 marks)

Mark	Description
0	25% or more of the comments that are present are inappropriate AND/OR at least 50% of the required comments are missing
1	At least 50% of the required comments are present AND the vast majority (75%+) of comments present are appropriate AND the requirements for a higher mark are not met
2	All or almost all required comments are present AND all or almost all comments present are appropriate AND the requirements for a mark of 3 are not met
3	All required comments are present AND all comments present are appropriate AND additional comments are present as appropriate to ensure clarity

Naming (1 mark)

Mark	Description
0	At least a few names used are inappropriate
0.5	Almost all names used are appropriate
1	All names used are appropriate

Modularity (1 mark)

Mark	Description
0	There are two or more instances of poor modularity (e.g. repeated code blocks)
0.5	There is one instance of poor modularity (e.g. a block of code repeated once)
1	There are no instances of poor modularity

SVN Commit History Marking (5 marks)

Markers will review your SVN commit history for your assignment up to your zip file creation time. This element will be graded according to the following principles:

- Appropriate use and frequency of commits (e.g. a single monolithic commit of your entire assignment will yield a score of zero for this section). Progressive development is expected, i.e., no large commits with multiple features in them.

[9] Every `.h` file in your submission must make sense without reference to any other files, e.g., it must `#include` any `.h` files that contain declarations or definitions used in that `.h` file. You can check that a header file compiles by itself by running `gcc -c` *`filename.h`* – with any other `gcc` arguments as required.

- Appropriate use of log messages to capture the changes represented by each commit. (Meaningful messages explain briefly what has changed in the commit (e.g. in terms of functionality, not in terms of specific numbered test cases in the test suite) and/or why the change has been made and will be usually be more detailed for significant changes.).

The standards expected are outlined in the following rubric. The mark awarded will be the highest for which the relevant standard is met.

Mark (out of 5)	**Description**
0	Minimal commit history – only one or two commits OR all commit messages are meaningless.
1	Some progressive development evident (three or more commits) AND at least one commit message is meaningful.
2	Progressive development is evident (multiple commits) AND at least half the commit messages are meaningful
3	Multiple commits that show progressive development of almost all or all functionality AND at least two-thirds of the commit messages are meaningful.
4	Multiple commits that show progressive development of ALL functionality AND meaningful messages for all but one or two of the commits.
5	Multiple commits that show progressive development of ALL functionality AND meaningful messages for ALL commits.

We understand that you are just getting to know Subversion, and you won't be penalised for a few "test commit" type messages. However, the markers must get a sense from your commit logs that you are practising and developing sound software engineering practices by documenting your changes as you go. In general, tiny changes deserve small comments – larger changes deserve more detailed commentary.

Total Mark

Let

- F be the functionality mark for your assignment (out of 60).
- S be the automated style mark for your assignment (out of 5).
- $A = F + \min\{F, S\}$ (the automatically determined mark for your assignment).
- H be the human style mark for your assignment (out of 5).
- C be the SVN commit history mark (out of 5).
- V be the scaling factor (0 to 1) determined after interview (see the CSSE7231 Student Interviews section below) – or 0 if you fail to attend a scheduled interview without having evidence of exceptional circumstances impacting your ability to attend.

Your total mark for the assignment will be:

$$M = (A + \min\{A, H\} + \min\{A, C\}) \times V$$

out of 75.

In other words, you can't get more marks for automated style than you do for functionality. Similarly, you can't get more marks for human style or SVN commit history than you do for functionality and automated style combined. Pretty code that doesn't work will not be rewarded!

Late Penalties

Late penalties will apply as outlined in the course profile.

CSSE7231 Student Interviews

The teaching staff will conduct interviews with all CSSE7231 students about their submissions, for the purposes of establishing genuine authorship. If you write your own code, you have nothing to fear from this process. If you legitimately use code from other sources (following the usage/referencing requirements outlined in this assignment, the style guide, and the AI tool use documentation requirements) then you are expected to understand that code. If you are not able to adequately explain the design of your solution and/or adequately explain your submitted code (and/or earlier versions in your repository) and/or be able to make simple modifications to it as requested at the interview, then your assignment mark will be scaled down based on the level of understanding you are able to demonstrate and/or your submission may be subject to a misconduct investigation where your interview responses form part of the evidence. Failure to attend a scheduled interview will result in zero marks for the assignment unless there are documented exceptional circumstances that prevent you from attending. Interviews will take place in your first prac session after the assignment deadline (or your extended deadline). You must attend the session that you have been allocated to.

Specification Updates

Any errors discovered in the assignment specification will be corrected and new versions will be released with adequate time for students to respond before the due date. Potential specification errors or inconsistencies can be discussed on the discussion forum. If your program's expected behaviour is not described in this specification then you can use the demo program (`demo-uqexpr`) to determine the expected behaviour. We will not update this specification when the expected behaviour is missing from this specification, only when this specification is inconsistent with itself or with the behaviour of the demo program or where it is not reasonably possible to determine the expected behaviour from the demo program.