COMP0002 C Coursework General Feedback from Previous Years

These comments highlight common issues seen when marking coursework from previous years, adding to the specific information given in the individual feedback on the marksheet you get. Coursework changes each year, so not all items will be directly relevant to this year's coursework. Review these comments to help understand good design and programming practice for the future.

1. Your source code has few or no comments, but your marksheet indicates that your comments are satisfactory, why is that?

   It means that your source code is sufficiently readable and understandable without needing comments (which is good!).

   Should you start adding more comments? No, if comments aren't needed then they aren't needed, don't go adding some just for the sake of it. Where comments are needed do them properly!

2. Your feedback on the marksheet is short, along the lines of 'there are no significant issues…', but others have a lot more feedback, what does this mean? It means you were making the expected progress, and didn't have any particular mistakes or problems to need more feedback about. This is good!

3. Is my code compiled and run? Yes, every coursework submission is compiled and run. The majority will run without error, but some may trigger segmentation faults or other errors at runtime. Where possible (and it is not too time consuming) the errors are fixed to allow the code to be run, but in a few cases this might not be possible. Details of such problems will be in the individual feedback on the marksheet.

4. The marksheet includes a list of warning messages produced by the compiler (see later), which are good indications of where there might be problems. Warnings should always be investigated and removed. Sometimes a warning provides the answer as to why your program crashes at runtime or corrupts data.

5. The code runs on your machine but the feedback says there were runtime errors? Yes, this can happen as different compiler versions and operating systems have differences in memory management. What might happen to work on your machine, doesn't work on the machine used to mark the coursework. All the errors reported in the feedback are traced to real errors, though, nearly always to do with memory management. For example, calling free on a pointer not allocated with calloc/malloc, or using an uninitialized variable, which will have a random value depending on what happens to be in memory when the program is run.

6. Use blank lines between sections of code consistently. A typical style is to leave one blank line between each function definition, and at least one blank line after the

#include directives, a block of global (external/file scope) variables, function declarations, structs or #defines.

The aim is to improve readability and avoid confusion when reading code due to inconsistent source code formatting. The use of blank lines has no effect on the runtime behaviour or performance of the code. Don't forget that Visual Studio Code will reformat code for you, so there is no excuse for messy looking code!

7. The same approach also applies to the use of indentation and spaces. Most programmers use 2 or 4 spaces for indentation, and also aim to line up blocks of code having the same pattern (e.g., lining up '=' operators).

8. Keep functions cohesive and short. Cohesive means that the function performs one focussed piece of behaviour. Short typically means less than 10-15 lines of source code, but there is not absolute number. It should always be possible to easily see all of a function in your editor window without needing to scroll.

9. The purpose of one distinct piece of behaviour depends on the kind of function. A basic function carries out one specific thing, such as calculating a value, fetching a value from a data structure or getting some input. A control function will carry out a slice of behaviour primarily by calling more basic functions, but does not attempt to implement all the behaviour itself. A control function may call other control functions (delegating behaviour) and can use loops or if statements.

10. If a function gets too long it will benefit from sections of code being removed into shorter functions and replaced by a function call. In particular, where comments have been used to explain a section of code in a long function, it would be much better to extract the code into a well-named short function, with the name of the function in the function call replacing the need to have the comment. This is called the Extract Function refactoring. As you will discover in the second year, long functions make code testing much harder or impossible, and also mess up the design of programs in general.

11. Beware of unintended recursion, in particular indirect recursion where function A calls function B and then function B calls function A. This creates a sort of loop where the functions keep calling each other without returning so that memory will eventually run out if the program runs for a long enough time. This can be harder to spot if you have a chain of functions, for example function B might call C which calls D and so on, with the function at the end of the chain then making a call back to A.

12. If a variable or function has been defined but never used, then remove it from the code.

13. Don't leave commented-out source code. If code is not used then remove it before submitting the work.

14. Variable and function names are important. Chose names that make the intent or purpose of the variable or function clear. The same applies to source code file names, avoid names like src.c!

15. By convention a name in all capitals like ‘INDEX’ is assumed to be the name of a constant, either a const variable or a #define macro. This is not defined by the C language but most programmers use the convention and not using it will cause confusion.

16. If the compiler displays warning messages when compiling your code, don’t ignore them! A warning will not prevent the code being compiled and run, but often means that your code has one or more problems and can be unreliable when run.

    Depending on which version of gcc or clang you are using, you may not see warning messages by default but can enable all warnings using the -Wall flag when compiling code. For example:
    ```
    gcc -Wall program.c
    ```
    You can also turn warnings into errors by adding the -Werror flag:
    ```
    gcc -Wall -Werror program.c
    ```
    This is also a good way of finding unused variables.

17. An array variable will have an array type, for example given char s[10] then s has the type ‘array of char of size 10’. If the array name is used in an expression where a pointer is expected then the type is decayed to a pointer type. For example, in the statement printf(“%s\n”, s), the type of s is converted to a pointer (char*), and printf will print the string held in the array that the pointer is pointing at.

    If you use the expression &c (address of s) you will still get the pointer to the array but also a type mismatch. You may see a warning by default, or need to compile using the -Wall flag, telling you that “the printf format specifies type 'char *' but the argument has type 'char (*)[10]’ ”. The char (*)[10] is the type pointer to an array of char of size 10, which is the full type of the expression &c. In this case you can get away with ignoring the warning as the code still happens to work because you get the correct pointer, but it is much better to take notice of the warning and fix the code to remove the warning.

    This means that where a pointer to the array is needed usually you can just give the array name (i.e., s) and the & operator is not needed.

18. If you have a statement like:
    ```
    if (x == 10) {
      return 1; // True
    }
    else {
      return 0; // False
    }
    ```
    It can be reduced to just:

```
return (x == 10);
```
The if statement is redundant and can be removed!
Actually, you don’t need the parentheses either in this example, so end up with:
```
return x == 10;
```

19. You can use 1 and 0 directly to represent true and false, but you end up with ‘magic numbers’ in your source code that can cause confusion. It is better to use the names true and false (or True and False). This can be done using #define:
    ```
    #define true 1
    #define false 0
    ```
    Or you can use #include<stdbool.h>, which defines true and false for you.

20. If you have a block of code that has this pattern:
    ```
    If (x == 1) // For each value of x call a different function.
    {
      doSomething();
    }
    else
    {
       if (x == 2)
       {
         doSomething2();
       }
       else
       {
          If (x == 3)
          {
            doSomething3();
          }
          else
          {
            // And so on for more values of x

             else
             {
                // do the default behaviour if the value of x is not matched.
                doDefault();
              }
          }
       }
    }
    ```
    then a switch statement should be used instead:
    ```
    switch(x)
    {
         case 1: doSomething(); break;
         case 2: doSomething2(); break;
         case 3: doSomething3(); break;
    ```

```
    default: doDefault();
}
```
This will be a lot easier to create, edit, and maintain. It also makes it easier to manage the default behaviour if no values are matched by a case. Don't forget to include the break statements.

Beware though! When using an object-oriented language like Java switch statements can be evil and a sign of poor design as they increase the coupling between sections of code.

21. A .h file should contain *declarations* only, no definitions. This means that complete function definitions (functions with function bodies) or variable definitions should *not* be in a .h file, just function signatures or extern variable declarations. Don't use .h files as a general purpose mechanism for including sections of code (definitions) into another source file.

22. .h files can also contain type declarations for structs and typedefs, but not variable definitions using a struct or typedef.

23. Functions should always be declared or defined before they are called. This means that a function declaration (i.e., function signature) or a complete definition must appear in the source code before any calls to the function. Often the compiler will report undeclared names by default but compiling with the -Wall flag will always identify where this has not happened.

24. If a function is defined as returning an int value, or any other type except void, then make sure that the function body does actually include a return statement! The function will still compile but will return an undefined value. Compiling the code using the -Wall flag will give you a warning about this. Further, a function with multiple possible exits, that may contain an if statement for example, should return a suitable value at each point the function can exit. The compiler will still give a warning when if it finds a path through the function with no return, even if you know that when the code is actually run it will evaluate a return statement.

25. Don't put a lot of code in the main function, it should just contain what is necessary to get the program started, calling other functions to do the main initialisation and other work.

26. The function itoa, to convert an int to a C string is supported by some C compiler libraries but is not part of the standard C library. In particular, not by gcc. If you use it in your code you have to define an implementation of the function as well.

27. Opening a file returns a FILE* pointer, but your code is *not* responsible for calling free on the pointer when no longer needed. You should call fclose, which will take care of any memory management issues. If you do call fclose and then free with code compiled by some versions of gcc you will get a “pointer being freed was not allocated” error at runtime and your program will terminate.

28. The background layer in a drawing program only needs to be drawn once to display the arena walls, obstacles and tiles. The robot and any other moving or temporary features are drawn on the foreground layer, leaving the background alone. If you find you are redrawing the background multiple times then review how you are trying to display the robot animation. Redrawing the background too frequently will also cause the window to appear to start flickering.

29. Printf statements in a drawing program get displayed in the message panel at the bottom of the drawing window. This is useful for checking something is working or basic debugging, but it is preferable to remove the printf statements before submitting to keep the drawing window tidy.

30. If a drawing program is running but appears to be doing nothing as the drawing window doesn't update and doesn't display the finished message, then either you program is stuck in a non-terminating loop, or it has failed with a segmentation fault. To find out if there has been a segmentation fault just run the a.out or a.exe file without piping the output to the drawing program and see what happens.