Course coordinator: Lawrence Ong (lawrence.ong@newcastle.edu.au)
School of Engineering
College of Engineering, Science and Environment
The University of Newcastle

Assignment 1

Instructions

- All exercises except the first one contain a multiple-choice question. Enter your answers to these questions on Canvas → Assignment 1. Complete all questions on Canvas.
- Some exercises come with multiple variations of the question. You will see a randomly chosen one on Canvas. You only need to answer the question selected for you on Canvas.
- You can download the code for all exercises as a ZIP file.
- Work individually.
- Assignment due 11:55pm, Sunday 24/3/2024 (week 4).
- Late submissions will have 10% deducted per day.
- Total mark possible 15.
- This assignment counts towards 10% of your final grade.

Assessment

This assignment will be assessed using the following criteria:

Component	Unsatisfactory	Improving	Satisfactory	Excellent
Multiple Choice Questions (15)	Incorrect answer	–	–	Correct Answer

Objectives

The main objective is to review some basics of C programming while at the same time learn some common techniques used in embedded system programming. There will be opportunities to reuse code from this assignment and what you have learnt in future assignments.

Software Development Tools

- C compiler. GNU Compiler Collection preferred (GCC).
  - Linux and MacOS: GCC has been preinstalled. You may want to use an IDE.

- Windows: Dev-C++ has been preinstalled on all PCs in EE104. If you are using your own Windows laptop, you can download Dev-C++:
  `https://www.embarcadero.com/free-tools/dev-cpp`

Notes

The laboratory exercises are built on topics covered during lectures. The exercises cover C specific features, C preprocessor functionality and standard C library functions. The preprocessor is a macro processor which is not strictly part of the C programming language, although is used ubiquitously with the C compiler. Library functions make programming easier by providing a suite of commonly used functions. However, for embedded systems, the use of library functions depends on the available hardware resources.

Each of the lab exercises has an associated program. The program can also be found in a ZIP file. Experiments to modify the code in the programs are suggested to demonstrate functionality, however you are encouraged to explore and make your own modifications.

These exercises are designed to give you an opportunity to experiment with the code to get an understanding of how it work. So be creative and have fun!

0 Hello World

The first exercise is a simple program that prints a message to the console.

Source Code

```c
// Ass-01-HelloWorld.c

#include <stdio.h>
int main (void)
{
    printf("Hello␣World.\n"); /* Print a string */

    int a = 1;

    printf("The␣number␣is␣%d.\n", a); /* Print formatted output */

    return 0;
}
```

Experiments

- Add another `printf()` statement.
- See what happens when `\n` is removing from the string.
- Try substituting the first `printf()` with `puts()`. Do you get exactly the same output with this substitution? Why?
- Repeat the above step for the second `printf()`. Can you explain what happens? Hints: See `https://cplusplus.com/reference/cstdio/puts/`.

1 Outputting Formatted Strings

The **`printf()`** and related library functions can be used for generating formatted strings to display variables. It has already been used in the previous examples, however some further functionality will be explored in this exercise. This includes outputting strings, characters, integers and floating point numbers. This is only a sample of what can be done, although it does cover the most common ones used. Refer to the *The GNU C Library Reference Manual* for further formatting options.

Source Code

```
// Ass-01-FormattedOutput.c

#include <stdio.h>

int main ( void )
{
  // Declare some variables and initialise them
  char s[] =
    { "This␣is␣a␣string" };
  char c = 'A';
  int i = 1234;
  float f = 12.3456;

  // Output formatted strings
  printf ("a)␣Output␣a␣string:␣%s\n", s);
  printf ("b)␣Output␣a␣character:␣%c\n", c);
  printf ("␣␣␣->␣As␣integer:␣%d\n", c);
  printf ("c)␣Output␣an␣integer:␣%d\n", i);
  printf ("␣␣␣->␣With␣seven␣characters:␣%7d\n", i);
  printf ("␣␣␣->␣With␣seven␣characters␣and␣leading␣zeros:␣%07d\n", i);
  printf ("d)␣Output␣a␣float:␣%f\n", f);
  printf ("␣␣␣->␣With␣six␣characters␣and␣two␣decimal␣places:␣%6.2f\n", f);

  return 0;
}
```

Experiments

- Change the formatting strings **`%7d`**, **`%07d`** and **`%6.2f`**.
- Change the numbers and strings being formatted.

Question

Given a few numbers (each between -999.0 and 999.0) of the `float` type, which of the following format specifier will always vertically align the decimal points (that is, the dot ".", for example in 12.45) of all the printed numbers? Select all correct answers.

- ☐ a) `%d`
- ☐ b) `%6.2f`
- ☐ c) `%-6.2f`
- ☐ d) `%7.2f`

Enter your answer on Canvas.

2 Data Types

This exercise introduces the most common data types that are used. In particular, the focus is on the amount of memory each data type uses as this will determine the range of possible numbers that can be encoded. The size of Standard C data types can change microprocessor architecture, however there are ways to declare variables where the size of the data type is the same across all architectures.

A later exercise will explore operators that can be used with different data types. The size of pointers to variables is introduced in this exercise, however pointers will be explored in a later exercise.

Source Code

```c
// Ass-01-DataTypes.c

#include <stdio.h>
#include <stdint.h>

int main ( void )
{
    printf ("a)␣Standard␣C␣data␣type␣sizes:\n");
    printf ("␣␣␣->␣sizeof(char)␣␣␣␣␣␣␣␣␣␣=␣%2lu\n", sizeof(char));
    printf ("␣␣␣->␣sizeof(short)␣␣␣␣␣␣␣␣␣=␣%2lu\n", sizeof(short));
    printf ("␣␣␣->␣sizeof(int)␣␣␣␣␣␣␣␣␣␣␣=␣%2lu\n", sizeof(int));
    printf ("␣␣␣->␣sizeof(unsigned␣int)␣␣=␣%2lu\n", sizeof(unsigned int));
    printf ("␣␣␣->␣sizeof(long)␣␣␣␣␣␣␣␣␣␣=␣%2lu\n", sizeof(long));
    printf ("␣␣␣->␣sizeof(unsigned␣long)␣=␣%2lu\n", sizeof(unsigned long));
    printf ("␣␣␣->␣sizeof(long␣long)␣␣␣␣␣=␣%2lu\n", sizeof(long long));
    printf ("␣␣␣->␣sizeof(float)␣␣␣␣␣␣␣␣␣=␣%2lu\n", sizeof(float));
    printf ("␣␣␣->␣sizeof(double)␣␣␣␣␣␣␣␣=␣%2lu\n", sizeof(double));
    printf ("␣␣␣->␣sizeof(long␣double)␣␣␣=␣%2lu\n", sizeof(long double));

    printf ("b)␣Standard␣integer␣data␣type␣sizes:\n");
    printf ("␣␣␣->␣sizeof(int8_t)␣␣␣␣␣␣␣␣=␣%2lu\n", sizeof(int8_t));
    printf ("␣␣␣->␣sizeof(int16_t)␣␣␣␣␣␣␣=␣%2lu\n", sizeof(int16_t));
    printf ("␣␣␣->␣sizeof(int32_t)␣␣␣␣␣␣␣=␣%2lu\n", sizeof(int32_t));
    printf ("␣␣␣->␣sizeof(int64_t)␣␣␣␣␣␣␣=␣%2lu\n", sizeof(int64_t));

    printf ("c)␣Pointer␣data␣sizes␣(note␣they␣are␣all␣the␣same):\n");
    printf ("␣␣␣->␣sizeof(char␣*)␣␣␣␣␣␣␣␣=␣%2lu\n", sizeof(char *));
    printf ("␣␣␣->␣sizeof(short␣*)␣␣␣␣␣␣␣=␣%2lu\n", sizeof(short *));
    printf ("␣␣␣->␣sizeof(int␣*)␣␣␣␣␣␣␣␣␣=␣%2lu\n", sizeof(int *));
    printf ("␣␣␣->␣sizeof(long␣*)␣␣␣␣␣␣␣␣=␣%2lu\n", sizeof(long *));
    printf ("␣␣␣->␣sizeof(long␣long␣*)␣␣␣=␣%2lu\n", sizeof(long long *));

    return 0;
}
```

Experiments

- Why do we use `%2lu` instead of `%2d` as the `printf()` specifier?
- Change `%2lu` to `%2d`. What is the output?
- Note that pointers to different data types all have the same size? Why?

Question option 1

What is the size (in number of **bytes**) of a `uint8_t` type variable?

- ☐ a) 2.
- ☐ b) 8.
- ☐ c) 16.
- ☐ d) None of the above.

Enter your answer on Canvas.

Question option 2

What is the size (in number of **bytes**) of a `uint32_t` type variable?

- ☐ a) 4.
- ☐ b) 8.
- ☐ c) 32.
- ☐ d) None of the above.

Enter your answer on Canvas.

Question option 3

What is the size (in number of **bits**) of a `uint8_t` type variable?

- ☐ a) 2.
- ☐ b) 8.
- ☐ c) 16.
- ☐ d) None of the above.

Enter your answer on Canvas.

Question option 4

What is the size (in number of **bits**) of a `uint32_t` type variable?

- ☐ a) 4.
- ☐ b) 8.
- ☐ c) 32.
- ☐ d) None of the above.

Enter your answer on Canvas.

3 Preprocessor Directives

The preprocessor interprets directives before the C compiler compiles the source code. There are a number of directives and this exercise explores the most common ones used. Some of the preprocessor directives have already been used in previous exercises, such as the `#include` directive. To explore this directive, this exercise also uses an include file that contains other preprocessor directives.

The main directives explored are

- `#include`
- `#define`
- `#ifdef ... #else ... #endif` or
  `#ifndef ... #else ... #endif`

If is also possible to use `#define` to define a macro that accepts arguments, which is very useful.

Source Code

The include file is:

```c
// Ass-01-Preprocessor.h

#ifndef ASS_01_PREPROCESSOR_H_
#define ASS_01_PREPROCESSOR_H_

// Test if DEF1 gets defined twice
#ifdef DEF1
#define DEF1_DEFINED_TWICE "Yes"
#else
#define DEF1_DEFINED_TWICE "No"
#endif /* DEF1 */

// Make some definitions
#define DEF1        // Defined only
#define DEF2 1234   // Defined to be a value

#endif /* ASS_01_PREPROCESSOR_H_ */
```

The exercise source is:

```c
// Ass-01-Preprocessor.c

#include <stdio.h>

int main ( void )
{

    // Conditional statement
    printf ("a)␣Conditional␣statement\n");
    printf ("␣␣␣Check␣if␣DEF1␣has␣been␣defined:\n");
```

```
#ifdef DEF1
    printf ("␣␣␣->␣DEF1␣has␣been␣defined\n");
#else
    printf ("␣␣␣->␣DEF1␣has␣not␣been␣defined\n");
#endif

    // Include statement
    // Note that DEF1 is defined in Ass-01-Preprocessor
    printf ("b)␣Include␣statement\n");
    printf ("␣␣␣Include␣Ass-01-Preprocessor.h␣(which␣defines␣DEF1)\n");
#include "Ass-01-Preprocessor.h"

    // Conditional statement for value
    printf ("c)␣Conditional␣statement␣for␣value\n");
    printf ("␣␣␣Check␣value␣of␣DEF2:\n");
#if DEF2 == 1234
    printf ("␣␣␣->␣DEF2␣=␣1234\n");
#else
    printf ("␣␣␣->␣DEF2␣does␣not␣equal␣1234\n");
#endif

    // Include statement
    // Note that Ass-01-Preprocessor.h is included again
    printf ("d)␣Include␣statement␣to␣check␣for␣reinclusion\n");
    printf ("␣␣␣Include␣Ass-01-Preprocessor.h␣again␣(which␣defines␣DEF1)\n");
#include "Ass-01-Preprocessor.h"
    printf ("␣␣␣Has␣DEF1␣been␣defined␣more␣than␣once:\n");
    printf ("␣␣␣->␣%s\n", DEF1_DEFINED_TWICE);

    // Use defines where numbers are used in more that one place
    printf ("e)␣Use␣defines␣rather␣than␣literals␣in␣more␣that␣once␣place\n");
    printf ("␣␣␣DEF2␣used␣in␣two␣places:\n");
    printf ("␣␣␣->␣DEF2␣=␣%d\n", DEF2);
    printf ("␣␣␣->␣DEF2*2␣=␣%d\n", DEF2 * 2);

    // Be careful with complex definitions
    // Use parenthesis since don't know where definition will be used
#define DEF3_BAD 1+2
#define DEF3_GOOD (1+2)
    printf ("f)␣Complex␣definitions\n");
    printf ("␣␣␣Be␣careful␣with␣complex␣definitions:\n");
    printf ("␣␣␣->␣DEF3_BAD␣=␣%d\n", DEF3_BAD);
    printf ("␣␣␣->␣DEF3_GOOD␣=␣%d\n", DEF3_GOOD);
    printf ("␣␣␣->␣DEF3_BAD*3␣=␣%d\n", DEF3_BAD * 3);
    printf ("␣␣␣->␣DEF3_GOOD*3␣=␣%d\n", DEF3_GOOD * 3);

    // Defining a macro that passes arguments
    {
        printf ("g)␣Macro␣that␣passes␣parameters\n");
#define MIN(x,y) ((x)<(y)?(x):(y))
        int a = 7;
        int b = 10;
        printf ("␣␣␣a␣=␣%d,␣b␣=␣%d\n", a, b);
        printf ("␣␣␣->␣MIN(a,b)␣=␣%d\n", MIN(a, b));
        printf ("␣␣␣->␣MIN(3+5,b)␣=␣%d\n", MIN(3+5,b));
    }

    return 0;
}
```

Experiments

- Understand the code so that you can explain the output of the program.
- Why are there so many parentheses in a macro definition?
- Write more examples using the `MIN()` macro, using complex parameters.
- Add this line just before `return 0;` and after the closed bracket `}`
  `printf ("␣->␣MIN(0,4)␣=␣%d\n", MIN(0, 4));`
  What do you get? Why?
- Add this line just before `return 0;` and after the closed bracket `}`
  `printf ("␣->␣MIN(0,a)␣=␣%d\n", MIN(0, a));`
  What do you get? Why?
- Define complex macros and make sure that the result is as you expect.
- Define your own macros with and without parameters.

Question option 1

What is the effect of deleting lines 3, 4, and 17 in `Ass-01-Preprocessor.h`? Select all correct answers.

☐ a) The code does not compile.

☐ b) `DEF1` is defined twice.

☐ c) `DEF2` is defined twice.

☐ d) There is no effect on the program.

Enter your answer on Canvas.

4 Data Structures

It is possible to organise different data types into a single data structure. This allows for convenient data manipulation when all of the elements within the data structure are related to each other.

This exercise shows how structures are defined and the elements in the structure are accessed. The method used to initialise the contents of structures is given, which can also be used for other data types. Structures can be used in a similar was to other data types, include as will be seen later, creating arrays and using pointers.

Source Code

```
// Ass-01-DataStructures.c

#include <stdio.h>
#include <stdint.h>

int main ( void )
{
    struct myFirstStruct
    {
        char c;
        int i;
        float f;
    } a =
        { 'a', 1, 2.34 };

    printf ("a)␣Elements␣in␣a␣(initialised␣at␣declaration):\n");
    printf ("␣␣␣a.c␣=␣'%c'\n", a.c);
    printf ("␣␣␣a.i␣=␣'%d'\n", a.i);
    printf ("␣␣␣a.f␣=␣'%f'\n", a.f);

    printf ("b)␣Elements␣in␣a␣(modified␣values):\n");
    a.c = 'b';
    a.i = 2;
    a.f = 5.67;
    printf ("␣␣␣a.c␣=␣'%c'\n", a.c);
    printf ("␣␣␣a.i␣=␣'%d'\n", a.i);
    printf ("␣␣␣a.f␣=␣'%f'\n", a.f);

    struct myFirstStruct *structPtr = &a;

    printf ("c)␣Accessing␣elements␣using␣a␣pointer:\n");
    printf ("␣␣␣structPtr->c␣=␣'%c'\n", structPtr->c);
    printf ("␣␣␣structPtr->i␣=␣'%d'\n", structPtr->i);
    printf ("␣␣␣structPtr->f␣=␣'%f'\n", structPtr->f);

    return 0;
}
```

Experiments

- Change the values in the data structure.
- Create your own data structures.

Question

How do we access the element `i` of `a` using `*structPtr`?

☐ a) `*structPtr.i`

☐ b) `(*structPtr).i`

☐ c) `*(structPtr.i)`

☐ d) `*structPtr->i`

☐ e) `*(structPtr)->i`

☐ f) `*(structPtr->i)`

Enter your answer on Canvas.

5 ASCII Characters

We have seen that strings are made up of characters between quotes. Each character has an associated character number. The relationship between the character and its associated character number is defined by the *American Standard Code for Information Interchange* (ASCII). The character number is also called ASCII code or character code. This exercise explores the mapping between the character number and the character/symbol. You will see that not all characters numbers produce a displayed symbol, but some provide control functions (for example, line feed).

This exercise will also look at reading characters from the keyboard to see what the character number is for various keys that are pressed. An ASCII character can be described using (i) the symbol within single quotes or (ii) the character number / character code. The former is only possible if a symbol for the character is available. Characters numbers that do not have a symbol can be generated using an escape sequence to produce a particular outcome (for example, `'\007'` or `'\a'` for bell).

Refer to https://en.wikipedia.org/wiki/Escape_sequences_in_C#Table_of_escape_sequences and https://en.wikipedia.org/wiki/ASCII#Control_code_chart for how to use an escape sequence.

Source Code

```c
//  Ass-01-ASCII.c

#include <stdio.h>

// Uncomment to enable reading characters from the console
//
// #define READ_INPUT

int main ( void )
{
#ifdef READ_INPUT
    int c;
#endif

    printf ("a)␣ASCII␣values␣for␣some␣characters:\n");
    printf ("␣␣␣%3d␣=>␣'%c'\n", 'A', 'A');
    printf ("␣␣␣%3d␣=>␣'%c'\n", 'a', 'a');
    printf ("␣␣␣%3d␣=>␣'%c'\n", '1', '1');
    printf ("␣␣␣%3d␣=>␣'%c'\n", '2', '2');
    printf ("b)␣Some␣escape␣sequences:\n");
    printf ("␣␣␣%3d␣=>␣'%c'\n", '\n', '\n');
    printf ("␣␣␣%3d␣=>␣'%c'\n", '\r', '\r');
    printf ("␣␣␣%3d␣=>␣'%c'\n", '\t', '\t');
    printf ("␣␣␣%3d␣=>␣'%c'\n", '\007', '\007');
    printf ("c)␣Press␣a␣key␣and␣then␣ENTER:\n");

#ifdef READ_INPUT
    fflush(stdout);
    c=getchar();
    printf ("␣␣␣Read␣%3d␣=>␣'%c'\n", c, c);
#else
```

```
    printf ("␣␣␣->␣Not␣enabled.\n");
#endif

    return 0;
}
```

Experiments

- Run the program noting what characters and character numbers are printed.
- Did you hear a sound? Which of the escape sequences produces the sound?
- Enable the inputting of a number of characters from the keyboard (disabled by default).
- Try pressing different keys and see the ASCII code.

Question option 1

What character is printed out for the ASCII character number 42 (in decimal)?

☐ a) `'*'`.

☐ b) `'3'`.

☐ c) `'C'`.

☐ d) None of the above.

Enter your answer on Canvas.

Question option 2

What character is printed out for the ASCII character number 35 (in decimal)?

☐ a) `'#'`.

☐ b) `'&'`.

☐ c) `'a'`.

☐ d) None of the above.

Enter your answer on Canvas.

Question option 3

What is the ASCII character number (in decimal) of the character `'0'` (the number zero)?

- ☐ a) 0.
- ☐ b) 60.
- ☐ c) 48.
- ☐ d) 30.

Enter your answer on Canvas.

Question option 4

What is the ASCII character number (in decimal) of the character `'1'` (the number one)?

- ☐ a) 1.
- ☐ b) 61
- ☐ c) 49.
- ☐ d) 31.

Enter your answer on Canvas.

6 Input and Output

A C program does not read input directly from the keyboard; rather, it reads from the input stream buffer (stdin). Operating in the line-buffering mode, the keyboard buffer is sent to stdin after the *enter* key in pressed, allowing any correction of key press (using *backspace*).

Also, a C program does not display characters directly to the console. It sends the output to the output stream buffer (stdout), and the buffer is then read by the console. The way that a console reads characters from the program's stdout and displays them varies among different consoles. PowerShell and Dev-C++ have been configured to read each character at a time. The console in STM32CubeIDE on a Windows PC will wait for the *enter* key to be pressed first before reading all the characters. Because of this, a **fflush()** statement is required to flush stdout to the console.

Source Code

```c
//  Ass-01-InputOutput.c

#include <stdio.h>

int main ( void )
{
    int i;
    int c;

    puts("Enter␣five␣characters␣and␣press␣ENTER:");

    for (i=1;i<=5;i++)
    {
        fflush(stdout);
        c = getchar();
        printf ("␣Input␣%d␣:␣'%c'\n", i, c);
    }

    return 0;
}
```

Experiments

- Pay special attention to pressing the `enter` key. Try pressing the `enter` key at different character input positions. Try the following sequence and observe the output:
  - a b c d e `enter`
  - a b c d `enter`
  - a b `enter` c `enter`
  - a b `backspace` c `enter` d e `enter`
- Note use of **fflush()** to flush out everything in the stdout buffer to the console.
- Compile and run the program under STM32CubeIDE on a Windows PC as a *Local C/C++ Application*. Comment out the **fflush()** statement and see what happens.

- Replace the last two lines in the `for` loop with the following code:
  `scanf("%d", &c);`
  `printf (" Input %d : '%d'\n", i, c);`
  Change the text `character` to `integer` in the argument of `puts()`.

- Experiment with different ways to input five integers:
  - 10 `enter` 200 `enter` 3 `enter` 40 `enter` 5 `enter`
  - 10 `space` 200 `space` 3 `space` 40 `space` 5 `enter`
  - 10 `space space` 200 `tab` 3 `space` 40 `enter enter` 5 `enter`
  - What happens when you enter a non-integer character, for example, `'a'`? Why?

Question

Which of the following are true? Select all correct answers.

☐ a) Both `scanf()` and `getchar()` read from stdin.

☐ b) `scanf()` ignores spaces, tabs, and enter (newline) when getting user's input, while `getchar()` does not.

☐ c) `getchar()` can read the *value* of an integer typed by the user, but the integer is restricted to one digit.

☐ d) None of the above.

Enter your answer on Canvas.

7 Operators

Operators can be used to perform calculations and comparisons of variables and constants. This includes arithmetic operators, comparison operators and logical operators.

This exercise explores these operations and also the effect of using different data types. Variables can be *explicitly type cast* to be a certain type when an assignment or calculation takes place. Alternatively, a variable are *implicitly type cast* based on rules defined by the compiler. It is always best to explicitly type cast variables if there are operations with mixed types.

Source Code

```
// Ass-01-Operators.c

#include <stdio.h>
#include <stdint.h>

int main ( void )
{
  uint8_t a1 = 20;
  uint8_t a2 = 15;
  uint8_t a3;
  uint16_t b1 = 20;
  uint16_t b2 = 15;
  uint16_t b3;
  float f1 = 20.0;
  float f2 = 15.0;
  float f3;

  printf ("a)␣Arithmetic␣operations:\n");
  a3 = a1 + a2;
  b3 = b1 + b2;
  f3 = f1 + f2;
  printf ("␣␣␣a1+a2␣=␣%d,␣b1+b2␣=␣%d,␣f1+f2␣=␣%f\n", a3, b3, f3);
  a3 = a1 - a2;
  b3 = b1 - b2;
  f3 = f1 - f2;
  printf ("␣␣␣a1-a2␣=␣%d,␣b1-b2␣=␣%d,␣f1-f2␣=␣%f\n", a3, b3, f3);
  a3 = a1 * a2;
  b3 = b1 * b2;
  f3 = f1 * f2;
  printf ("␣␣␣a1*a2␣=␣%d,␣b1*b2␣=␣%d,␣f1*f2␣=␣%f\n", a3, b3, f3);
  a3 = a1 / a2;
  b3 = b1 / b2;
  f3 = f1 / f2;
  printf ("␣␣␣a1/a2␣=␣%d,␣b1/b2␣=␣%d,␣f1/f2␣=␣%f\n", a3, b3, f3);
  a3 = a1 % a2;
  b3 = b1 % b2;
  printf ("␣␣␣a1%%a2␣=␣%d,␣b1%%b2␣=␣%d\n", a3, b3);

  printf ("b)␣Effect␣of␣implicit␣and␣explicit␣type␣casting:\n");
  f3 = a1 / a2;
  printf ("␣␣␣a1/a2␣=␣%f␣(implicit)\n", f3);
  f3 = (float)a1 / (float)a2;
  printf ("␣␣␣a1/a2␣=␣%f␣(explicit)\n", f3);

  printf ("c)␣Comparison␣operations:\n");
  printf ("␣␣␣a1==a2␣=␣%d,␣a1!=a2␣=␣%d\n", a1 == a2, a1 != a2);
  printf ("␣␣␣a1<a2␣␣=␣%d,␣a1>a2␣␣=␣%d\n", a1 < a2, a1 > a2);
  printf ("␣␣␣a1<=a2␣=␣%d,␣a1>=a2␣=␣%d\n", a1 <= a2, a1 >= a2);

  return 0;
}
```

Experiments

- Note the result of operations. Experiment with the values used in the operations and be able to explain the results.
- Try using explicit type casting and see what the result is over implicit type casting.
- Note what happens when the data type is not big enough to hold the result. See the result of **`a3 = a1 * a2;`**

Question

What is the result of **`a1 == a1 - 1`**?

☐ a) -1.

☐ b) 0.

☐ c) 1.

☐ d) It depends on what the value of **`a1`** is.

Enter your answer on Canvas.

8 Pointers and Addressing

One of the powerful features of C is the ability to reference variables by their address. However, this generally leads to the most common problems, referencing to invalid locations in memory. With some discipline and only using pointers where necessary, they can be used successfully. One method of avoiding confusion is to indication in a pointer variable name that the variable is a pointer. Pointers are used implicitly when using arrays (will see in a future exercise). Pointers are also useful for creating linked lists of information.

This exercise explores the use of *pointers*, which are effectively the address of variables in memory space. This includes *declaring a pointer*, *assigning a pointer value* and *dereferencing a pointer value*. Pointers can be the address of a variable of any type, although a `int16_t` will be used for this exercise. Note that the number of bytes used to store a pointer value depends on the architecture of the microprocessor. To print a pointer value, the formatting string `"%p"` can be used.

Source Code

```c
// Ass-01-Pointers.c

#include <stdio.h>
#include <stdint.h>

int main ( void )
{
  int16_t a;     // Declare a int16_t
  int16_t *aPtr; // Declare a pointer to a int16_t (use *)

  printf ("a)␣Assigning␣a␣pointer:\n");
  a = 1234;
  aPtr = &a; // Assign to be the address of a (use &)
  printf ("␣␣␣a␣=␣%d,␣*aPtr␣=␣%d,␣aPtr␣=␣%p,␣&a␣=␣%p\n", a, *aPtr, aPtr, &a);
  printf ("b)␣Dereferencing␣a␣pointer:\n");
  *aPtr = 5678; // Set value of a by dereferencing pointer to a (use *)
  printf ("␣␣␣a␣=␣%d,␣*aPtr␣=␣%d,␣aPtr␣=␣%p,␣&a␣=␣%p\n", a, *aPtr, aPtr, &a);

  return 0;
}
```

Experiments

- Note whether the code is compiled as a 32-bit or 64-bit application. How can you tell?
- See what happens if `a` is not assigned an initial value.
- See what happens if the pointer `aPtr` is set to zero.

Question

What is the result of `*(int8_t *)aPtr` when `*aPtr = 0x0103`?

- ☐ a) 3.
- ☐ b) 259.
- ☐ c) 0x0103.
- ☐ d) None of the above.

Enter your answer on Canvas.

9 Arrays

An array is a collection of data that can be referenced by an index. The index of an array starts at zero and finishes at one less than the size of the array. The array variable is effectively a pointer to the location where the array is being stored. The array elements can be of any type and can even by structures. However, each of the array elements have to be the same type. Multidimensional arrays can also be declare, however they are not explored in this exercise.

This exercise explores the *declaration* of arrays, *indexing* elements in the array and the link to *pointers*. Pointer arithmetic will also be introduced. While pointer arithmetic can be useful, it is generally better to avoid using it for arrays if possible.

Source Code

```c
// Ass-01-Arrays.c

#include <stdio.h>
#include <stdint.h>

int main ( void )
{
    int16_t a[3];    // Declare an array of three int16_t

    printf ("a)␣Indexing␣array␣elements:\n");
    a[0] = 10;
    a[1] = 20;
    a[2] = 30;
    printf ("␣␣␣a[0]␣=␣%d,␣a[1]␣=␣%d,␣a[2]␣=␣%d\n",
            a[0], a[1], a[2]);

    printf ("b)␣Array␣name␣as␣pointer:\n");
    printf ("␣␣␣a␣=␣%p\n", a);
    printf ("␣␣␣&(a[0])␣=␣%p,␣&(a[1])␣=␣%p,␣&(a[2])␣=␣%p\n",
            &(a[0]), &(a[1]), &(a[2]));

    printf ("c)␣Pointer␣arithmetic:\n");
    printf ("␣␣␣(a+0)␣=␣%p,␣(a+1)␣=␣%p,␣(a+2)␣=␣%p\n",
            (a + 0), (a + 1), (a + 2));
    printf ("␣␣␣*(a+0)␣=␣%d,␣*(a+1)␣=␣%d,␣*(a+2)␣=␣%d\n",
            *(a + 0), *(a + 1), *(a + 2));

    printf ("d)␣Note␣the␣need␣for␣parenthesis:\n");
    printf ("␣␣␣*a+0␣=␣%d,␣*a+1␣=␣%d,␣*a+2␣=␣%d\n",
            *a+0, *a+1, *a+2);

    return 0;
}
```

Experiments

- Try changing the type of `a` and observe the memory locations of each element.

- Try using pointer arithmetic to index all values in the array.
- Can you explain why the output of `*(a+1)` is different from that of `*a+1`?
- See what happens if the index value is too large. The program may crash.

Question

What does the result of `sizeof(a)` show?

☐ a) The number of bits used to store the entire array `a`.

☐ b) The number of bytes used to store the entire array `a`.

☐ c) The number of bits used to store one element of the array `a`.

☐ d) The number of bytes used to store one element of the array `a`.

Enter your answer on Canvas.

10 Array of Pointers

The previous two exercises covered pointers and arrays. The elements in an array can be of any type, including pointers. However, since the array variable itself is a pointer, it can confusing declaring arrays of pointers. Care also needs to be taken to ensure that memory for variables is allocated when required. Generally the pointer values will point to variables that have already been declared separately.

This exercise explores the use of array of pointers. The first part declares an array of pointers to the same data type, making dereferencing of pointers straight forward as the type of the dereferenced variable is implicit in the array declaration. The second part introduces a *void pointer* which is a pointer that does not indicate what data type it is pointing to. When dereferencing the pointer, the type needs to be explicitly stated. In both cases, the array index is evaluated first.

Source Code

```c
// Ass-01-ArrayPointers.c

#include <stdio.h>
#include <stdint.h>

int main ( void )
{
  int16_t i, j, k; // Declare three int16_t variables
  int16_t *aPtr[3]; // Declare an array of three pointers to int16_t

  int8_t x;
  int16_t y;
  float z;
  void *bPtr[3];     // Declare an array of three void pointers;

  printf ("a)␣Indexing␣array␣elements:\n");
  i = 12;
  j = 34;
  k = 56;
  aPtr[0] = &i;
  aPtr[1] = &j;
  aPtr[2] = &k;
  printf ("␣␣␣aPtr[0]␣=␣%p,␣aPtr[1]␣=␣%p,␣aPtr[2]␣=␣%p\n",
          aPtr[0], aPtr[1], aPtr[2]);
  printf ("␣␣␣*aPtr[0]␣=␣%d,␣*aPtr[1]␣=␣%d,␣*aPtr[2]␣=␣%d\n",
          *aPtr[0], *aPtr[1], *aPtr[2]);

  printf ("b)␣Pointers␣to␣different␣data␣types:\n");
  x = 11;
  y = 22;
  z = 3.3;
  bPtr[0] = &x;
  bPtr[1] = &y;
  bPtr[2] = &z;
  printf ("␣␣␣bPtr[0]␣=␣%p,\n␣␣␣bPtr[1]␣=␣%p,\n␣␣␣bPtr[2]␣=␣%p\n",
          bPtr[0], bPtr[1], bPtr[2]);
```

```
37   printf ("␣␣␣*(int8_t␣*)bPtr[0]␣=␣%d,\n"
38       "␣␣␣*(int16_t␣*)bPtr[1]␣=␣%d,\n␣␣␣*(float␣*)bPtr[2]␣=␣%f\n",
39       *(int8_t *) bPtr[0], *(int16_t *) bPtr[1], *(float *) bPtr[2]);
40   printf ("␣␣␣&(*(int8_t␣*)bPtr[0])␣=␣%p,\n"
41       "␣␣␣&(*(int16_t␣*)bPtr[1])␣=␣%p,\n␣␣␣&(*(float␣*)bPtr[2])␣=␣%p\n",
42       &(*(int8_t *) bPtr[0]),
43       &(*(int16_t *) bPtr[1]), &(*(float *) bPtr[2]));
44
45   return 0;
46 }
```

Experiments

- Try changing the data type for `i`, `j`, `k` and `aPtr`. Note any changes in the output.
- When dereferencing the void pointers, try type casting to an incorrect type to see what happens.

Question

What is the result of changing `*(int8_t *)bPtr[0]` to `*bPtr[0]` in line 39?

☐ a) There is not difference in output.

☐ b) Does not output the expected value.

☐ c) Does not compile.

☐ d) None of the above.

Enter your answer on Canvas.

11 Flow Control

There are a number of constructs within C that allow for decision making that alters flow of instructions that are executed. These can be decisions that are made to take one of a number of forward paths (`if` and `switch`) and decisions that are made to create a loop (`for` and `while`).

This exercise explores the flow control constructs. The use of each flow control construct depends on the application. An `if` statement can be used for simple decisions and a `switch` statement used when there are many options. Similarly for the two looping options, generally the one that creates a more logical structure for the program is used. It is also possible to do the test for the loop at the end using a `do` statement.

Curly brackets are used to group a number of statements. Even if there is only one statement, it is better to add curly brackets to make the code more readable, as done for this exercise.

Source Code

```
// Ass-01-FlowControl.c

#include <stdio.h>
#include <stdint.h>

int main ( void )
{
    int16_t a = 1;
    int16_t i;

    printf ("a)␣Use␣of␣the␣if␣statement:\n");
    printf ("␣␣␣a␣=␣%d\n", a);
    printf ("␣␣␣");
    if (a == 1)
    {
        printf ("a␣is␣1\n");
    }
    else if (a == 2)
    {
        printf ("a␣is␣2\n");
    }
    else
    {
        printf ("a␣not␣1␣or␣2\n");
    }

    printf ("b)␣Use␣of␣the␣switch␣statement:\n");
    printf ("␣␣␣a␣=␣%d\n", a);
    printf ("␣␣␣");
    switch (a)
    {
    case 1:
        printf ("a␣is␣1\n");
        break;
    case 2:
        printf ("a␣is␣2\n");
        break;
    default:
        printf ("a␣not␣1␣or␣2\n");
        break;
    }

    printf ("c)␣Use␣of␣for␣loop:\n");
    for (i = 0; i < 4; i++)
    {
        printf ("␣␣␣i␣=␣%d\n", i);
    }
    printf ("␣␣␣i␣=␣%d␣(after␣loop)\n", i);

    printf ("d)␣Use␣of␣while␣loop:\n");
    i = 0;
    while (i < 4)
    {
        printf ("␣␣␣i␣=␣%d\n", i);
        i++;
    }
    printf ("␣␣␣i␣=␣%d␣(after␣loop)\n", i);

    printf ("e)␣Use␣of␣do␣while␣loop:\n");
    i = 0;
    do
    {
        printf ("␣␣␣i␣=␣%d\n", i);
        i++;
    }
    while (i < 4);
    printf ("␣␣␣i␣=␣%d␣(after␣loop)\n", i);

    return 0;
}
```

Experiments

- Change the value of `a` to see the effect.
- See what happens if a `break` statement is removed.
- Change the value of `i` at the start of the `while` loop to see the effect.

Question

What happens if `i = 0` (in line 60) is replaced with `i = 4` before the start of the

```
do
{
   ...
} while()
```

loop?

- ☐ a) There is no difference in output.
- ☐ b) The `do...while` loop code is executed once only.
- ☐ c) The `do...while` loop code does not execute at all.
- ☐ d) None of the above.

Enter your answer on Canvas.

12 Multiple Source Files

In larger software projects, it is common practice to organise code across multiple source files to enhance readability and maintainability. This approach allows for the creation of reusable functions, which can be stored in individual files for future use in other projects.

In C programming, when a variable or function is defined in one file but referenced in another, the compiler must be made aware of its existence. This is typically accomplished by declaring the variable using the `extern` keyword before its usage. Similarly, for functions, a function prototype is provided prior to calling the function.

To streamline this process, it is often more convenient to declare variables and functions in a shared header file. By including this header file in all relevant source files, developers can ensure that the necessary declarations are readily available throughout the project.

This exercise delves into the use of header files for declaring variables and functions, providing insight into effective code organisation and inter-file communication.

Source Code

```c
// Ass-01-MultiFiles.h

#ifndef ASS_01_MULTIFILES_H_
#define ASS_01_MULTIFILES_H_

extern int a;
void increaseA(void);

#endif /* ASS_01_MULTIFILES_H_ */
```

```c
// Ass-01-MultiFiles1.c

#include <stdio.h>
#include "Ass-01-MultiFiles.h"

int main () {
    printf("a␣=␣%d\n", a);
    increaseA();
    printf("a␣=␣%d\n", a);
    return 0;
}
```

```c
// Ass-01-MultiFiles2.c

int a = 0;

void increaseA (void) {
    a++;
}
```

Experiments

- Try moving the declaration into `Ass-01-MultiFiles1.c`
- Try moving the definition of `a` to another source file, say `Ass-01-MultiFiles3.c`
- Modify the binary code to output text data. That is, make `dd_out` a string.
- Set the properties of the output file to read only using Windows Explorer and run the program.

Question

Which of the following actions will result in either a compilation or a linking error? Select all correct answers.

☐ a) Removing the declaration of the variable `a` from `Ass-01-MultiFiles.h`

☐ b) Removing the declaration of the function `increaseA()` from `Ass-01-MultiFiles.h`

☐ c) Removing `#include "Ass-01-MultiFiles.h"` from `Ass-01-MultiFiles1.c`

☐ d) Removing the keyword `extern` in line 6 of `Ass-01-MultiFiles.h`

Enter your answer on Canvas.

13 Dynamic Memory Allocation

There a situations where the amount of storage required is not known at compile time. Memory can be allocation for storage from the heap at run time using the library function **`malloc()`**. A pointer is returned to the area of memory which can be used. When this memory is not longer required, it can be returned to the heap using **`free()`**. Care must be taken not to use previously allocated memory once it is freed since it may be allocated for another purpose elsewhere.

This exercise explores allocating and freeing memory. Note that the allocated amount is in bytes, so the size of the data needs to be taken into account when requesting an amount of memory.

Source Code

```c
// Ass-01-MemoryAllocation.c

#include <stdio.h>
#include <stdint.h>
#include <stdlib.h> // for malloc, free

int main ( void )
{
#define NUM_VALUES 100
    int16_t *aPtr; // Pointer to memory to be allocated
    int16_t *bPtr; // Pointer to memory to be allocated
    int16_t *cPtr; // Pointer to memory to be allocated

    printf ("a)␣Allocate␣memory␣to␣aPtr␣and␣assign␣value:\n");
    aPtr = (int16_t *) malloc ((size_t) (NUM_VALUES * sizeof(int16_t)));
    if (aPtr == NULL)
    {
        printf ("␣␣␣ERROR:␣Could␣not␣allocate␣memory");
        return 1;
    }
    *aPtr = 1;
    printf ("␣␣␣aPtr␣=␣%p\n", aPtr);
    printf( "␣␣*aPtr␣=␣%u\n", *aPtr);

    printf ("b)␣Allocate␣memory␣to␣bPtr␣and␣assign␣value:\n");
    bPtr = (int16_t *) malloc ((size_t) (NUM_VALUES * sizeof(int16_t)));
    if (bPtr == NULL)
    {
        printf ("␣␣␣ERROR:␣Could␣not␣allocate␣memory");
        return 1;
    }
    *bPtr = 2;
    printf ("␣␣␣aPtr␣=␣%p\n", aPtr);
    printf( "␣␣*aPtr␣=␣%u\n", *aPtr);
    printf ("␣␣␣bPtr␣=␣%p\n", bPtr);
    printf( "␣␣*bPtr␣=␣%u\n", *bPtr);

    printf ("c)␣Free␣memory␣pointed␣to␣by␣aPtr:\n");
    free (aPtr);
    printf ("␣␣␣aPtr␣=␣%p\n", aPtr);
    printf ("d)␣Allocate␣memory␣to␣cPtr␣and␣assign␣value:\n");
    cPtr = (int16_t *) malloc ((size_t) (NUM_VALUES * sizeof(int16_t)));
    if (cPtr == NULL)
    {
        printf ("␣␣␣ERROR:␣Could␣not␣allocate␣memory");
        return 1;
    }
    *cPtr = 3;
    printf ("␣␣␣aPtr␣=␣%p\n", aPtr);
    printf( "␣␣*aPtr␣=␣%u\n", *aPtr);
    printf ("␣␣␣bPtr␣=␣%p\n", bPtr);
    printf ("␣␣*bPtr␣=␣%u\n", *bPtr);
    printf ("␣␣␣cPtr␣=␣%p\n", cPtr);
    printf ("␣␣*cPtr␣=␣%u\n", *cPtr);

    free(bPtr);
    free(cPtr);

    return 0;
}
```

Experiments

- Try allocating a large amount of memory by modifying **`NUM_VALUES`**. Then try putting a loop around just the first part that only allocates memory to seen when it fails.
- What happens to the original value pointed by **`aPtr`** in part **`d)`** ?
- Can you explain why you get the value of **`*aPtr`** in part **`d)`** ?

Question

What is the value of the pointer **`aPtr`** after **`free`** is called?

☐ a) The value stays the same.

☐ b) It changes to zero.

☐ c) It changes to -1.

☐ d) None of the above.

Enter your answer on Canvas.

14 Memory Management

One facet to writing C code is the *scope* of variables and functions. The aim is to limit the scope of variables to only those functions that require it. This can be done by declaring variables locally within a function. These variables are created on the stack when the function is called. The use of the qualifier **`static`** will place local variables outside of the stack so that they are persistent between function calls, but still local to the function. It can also be used at the file level so that other files cannot see that variable. This also allows for the reuse of variable names which do not cover the same scope. Similarly, when a function is declared **`static`**, access to the function is restricted to the file where it is declared.

This exercise demonstrates the use of scope by noting the address of variables with the same name which are declared in different areas. Note that even though the variable name is the same, the scope is different. The use of **`static`** is also explored. Note that the location in memory of the static variable is close to that declared in the file.

Source Code

```c
// Ass-01-MemoryManagement.c

#include <stdio.h>
#include <stdint.h>

int16_t m; // Local to this file

static void TestStatic ( void ); // Function prototype

int main ( void )
{
    int16_t a; // Local to function

    printf ("a)␣Module␣and␣local␣variables:\n");
    printf ("␣␣␣&m␣=␣%p\n", &m);
    printf ("␣␣␣&a␣=␣%p\n", &a);

    printf ("b)␣Scope:\n");
    {
        int16_t m; // Module level
        int16_t a; // Local to function

        printf ("␣␣␣&m␣=␣%p\n", &m);
        printf ("␣␣␣&a␣=␣%p\n", &a);
    }

    printf ("c)␣Static␣variable:\n");
    TestStatic ();
    TestStatic ();
    TestStatic ();

    return 0;
}
```

```
static void TestStatic ( void ) // This function only visible in this file
{
    static int16_t s = 0;

    if (s == 0)
    {
        printf ("␣␣␣&s␣=␣%p\n", &s);
    }
    printf ("␣␣␣s␣=␣%d\n", s);
    s++;
}
```

Experiments

- Try changing the variable `a` to be static in one or both locations and see what happens.
- Remove the `static` qualifier for `s` and see what happens.

Question

What is the result of removing the `static` qualifier from the declaration of `s` declared in `TestStatic()` and then calling `TestStatic()` three times in a row?

☐ a) `s = 0`, `s = 1`, `s = 2`.

☐ b) `s = 0`, `s = 0`, `s = 0`.

☐ c) `s = 0`, `s = 1`, `s = 1`.

☐ d) None of the above.

Enter your answer on Canvas.

15 Functions and Argument Passing

Functions allow any overall system to be broken down into smaller logical unit. The typical process for developing a complex system is top down design with bottom up refinement. As seen in the previous exercise, there are ways of keeping data local to a particular function. Information can be passed into a function by passing one or more arguments. However, a function can only return a signal value. Whilst it can return only a signal value it is possible to return more if a pointer to returned data is passed.

One standard method for passing information into and out of functions is to use pointers in the arguments and use the return value to indicate if the result of the function was valid. While parameters can be passed to a function by value, if a lot of data has to be passed it can be inefficient as the data has to be copied into variables within the function. If pointers are used, only the pointer value is copied.

This exercise demonstrates how arguments can be passed to a function and how the return value can be used to indicate the status of the function. It is possible to indicate that an argument being passed is an array by using square brackets. This is effectively indicating that the argument being passed is a pointer.

Source Code

```c
// Ass-01-Functions.c

#include <stdio.h>
#include <stdint.h>
#include <math.h>

// Define return status values for mySquareRootArray
#define SRA_OK 0
#define SRA_ERR_ZERO_SIZE 1
#define SRA_ERR_NEG_VALUE 2

int16_t mySquareRootArray (int16_t a_out[], int16_t a_in[], uint16_t a_size);

int main ( void )
{
#define ARRAY_SIZE 3
  int16_t a[ARRAY_SIZE] = { 1, 4, 9 };
  int16_t b[ARRAY_SIZE];
  int16_t s;
  uint16_t i;

  printf ("a)␣Valid␣input␣data:\n");
  if ((s = mySquareRootArray (b, a, ARRAY_SIZE)) == SRA_OK)
  {
    for (i = 0; i < ARRAY_SIZE; i++)
    {
      printf ("␣␣␣sqrt(%d)␣=␣%d\n", a[i], b[i]);
    }
  }
  else
  {
    printf ("␣␣␣ERROR:␣Return␣status␣=␣%d\n", s);
  }

  printf ("b)␣Zero␣size␣data:\n");
  if ((s = mySquareRootArray (b, a, 0)) == SRA_OK)
  {
    for (i = 0; i < ARRAY_SIZE; i++)
    {
      printf ("␣␣␣sqrt(%d)␣=␣%d\n", a[i], b[i]);
    }
  }
  else
  {
    printf ("␣␣␣ERROR:␣Return␣status␣=␣%d\n", s);
  }

  printf ("c)␣Negative␣input␣data:\n");
  a[1] = -a[1]; // Negate one input
  if ((s = mySquareRootArray (b, a, ARRAY_SIZE)) == SRA_OK)
  {
    for (i = 0; i < ARRAY_SIZE; i++)
    {
      printf ("␣␣␣sqrt(%d)␣=␣%d\n", a[i], b[i]);
    }
  }
  else
  {
    printf ("␣␣␣ERROR:␣Return␣status␣=␣%d\n", s);
  }

  return 0;
}

int16_t mySquareRootArray (int16_t a_out[], int16_t a_in[], uint16_t a_size)
{
  uint16_t i;

  if (a_size == 0)
  {
    return SRA_ERR_ZERO_SIZE;
  }

  for (i = 0; i < a_size; i++)
  {
    if (a_in[i] < 0)
    {
      return SRA_ERR_NEG_VALUE;
    }
    a_out[i] = sqrt (a_in[i]);
  }

  return SRA_OK;
}
```

Experiments

- Are there any other error conditions that could be checked?
- Replace the function parameter `a_out[]` with `*a_out` and see if there is any difference.
- Write a new function for `mySquareRootArray()` where the input is passed by value rather than an array pointer. Run this and the other function in a loop to compare speed difference.

Question

What is the result of replacing the function parameter `a_out[]` with `*a_out` in line 65?

☐ a) The code does not compile.

☐ b) The code compiles but gives the wrong result.

☐ c) The result is the same since pointers can be subscripted like arrays.

☐ d) None of the above.

Enter your answer on Canvas.

Further Reading

Books that cover C programming:

- Recommended text
- Mike McGrath *C Programming in Easy Steps*
  https://books.newcastle.edu.au/record=b2076291~S16

Some websites to provide further information on C programming for the beginner:

- Learn C Programming
  https://www.programiz.com/c-programming
- C Tutorial
  https://www.tutorialspoint.com/cprogramming

Reference material for the C preprocessor and library:

- The C Preprocessor
  https://gcc.gnu.org/onlinedocs/gcc-8.2.0/cpp.pdf
- Standard C/C++ Library Reference
  https://www.cplusplus.com/reference/