Started on	Wednesday, 31 July 2019, 12:19 PM
State	Finished
Completed on	Thursday, 8 August 2019, 12:00 AM
Time taken	7 days 11 hours
Marks	17.90/18.00
Grade	**9.94** out of 10.00 (**99**%)

Information

Arrays and Functions

Goals of lab

This lab introduces C arrays (without the complexity of pointers) and C functions. By the end of the lab you should:

- be able to write simple programs that use 1-D and 2-D arrays (without the use of pointers),
- understand how C programs can be broken into functions,
- appreciate the difference between local variables (within a function's stack frame) and external (or "global") variables,
- understand how parameters get passed by value to a function, except for arrays, and
- be able to write your own functions.

This lab covers the material in chapters 8, 9 and 10 of the text by King. You should have the text with you in the lab and be reading it concurrently with this lab (or, better, have read it beforehand).

A playlist for the videos relating to this lab is here.

WARNING: this is a long lab. Get started on it early and don't expect to complete it in a single 2-hour lab session.

Information

The char data type, getchar() and putchar()

Create a file *chario.c* containing the following code, which is explained in the video *chario.c*.

Hide `chario.c` code

```
/* Program to introduce the 'char' data type, plus the use of getchar and
 * putchar to read and write chars from/to standard input.
 * Reads characters from stdin, printing each one out in both decimal and octal,
 * until '\n' or end-of-file is reached.
 * Written for ENCE260 by RJL.
 * June 2016
 */

#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    char someChar = 0;          // A character
    int c = 0;                  // An int with a char in its low byte

    someChar = '*';
    printf("someChar = %c\n", someChar); // Print a single char

    printf("Enter a line of text, terminated by 'Enter'\n");

    // Read and print characters until newline or End-Of-File
    c = getchar();              // Get char (cast to int) or EOF
    while (c != '\n' && c != EOF) {
        printf("Character '%c', decimal %d, octal %o, hexadecimal %x\n", c, c, c, c);
        c = getchar();
    }

    return EXIT_SUCCESS;
}
```

Build it and run it, entering a line of input text like "ENCE260 is great!". You should see all the characters you typed printed out as characters and as decimal, octal and hexadecimal integers.

Note the following:

- The `char` data type can be used to hold single characters. Character literals (i.e. constants) are written with single quotes, distinguishing them from strings (covered later), which are written with double quotes.
- The use of the one-character identifier *c* to hold a generic character is acceptable throughout this course. Please don't use it for any other purpose, however.
- Characters in C are essentially just 8-bit integers. These integer values, or at least those in the range 0 to 127, are interpreted as characters via the ASCII table (see King Appendix E, or type `man ascii` to the shell, or visit `http://www.asciitable.com/`). Thus, if `c` is some integer type like `char` or `int`, we can set it to represent the character 'A' by any of the following statements:

```
c = 'A';
c = 65;      /* Look in the ASCII table if you don't understand! */
c = 0101;    /* an octal constant */
c = '\101';  /* octal is assumed in this form of char constant */
c = 0x41;    /* a hexadecimal constant */
c = '\x41';  /* the char whose hexadecimal value is 41 */
```

- Whether the variable `c` represents a character or a number depends only on how you use it. If you use `printf` to print `c` with a format "%c", you'll get the character 'A' printed, but if you print it with "%d" you'll get the number 65. If you do arithmetic on it, it's treated as an integer.
- Functions `getchar` and `putchar` respectively read a character from standard input and write a character to standard output. The odd (and important) thing to note about *getchar* is that it's defined to return an *int* value, not a *char*. The reason is a bit subtle. If *getchar* returned a char (which is an 8-bit integer, having $2^8 = 256$ possible states), it could represent any of the characters 0 through 255. But how then could *getchar* indicate that it had hit the end of the input file (if reading from a file rather than the keyboard) and so was unable to return a character? In C, functions can't throw an exception (there's no such thing in C!), so error situations have to be specially encoded into the return value. In this case, the return value from *getchar* is either an integer value in the range 0 - 255, indicating that a genuine character was read, or the special value EOF (#defined as -1 in *stdio.h*, and standing for End Of File) to indicate that no character was available.

A more idiomatic version of the line-reading code

The above version of the program uses the COSC121-style "one-and-a-half loop" to read a line of characters. However, most C programmers would write that loop more like the following:

```
// Read characters until newline or EOF
while((c = getchar()) != '\n' && c != EOF) {
    ...
}
```

This uses the assignment operator within the while loop condition to fetch the character before testing it, thereby eliminating the need for the clumsy one-and-a-half loop. This idiom is in such common use in the C community that is deemed totally acceptable in this course. It will be used in most future examples.

Question **1**

Correct

Mark 1.00 out of 1.00

If *c* is a variable of type *char*, which of the following statements can be used to set *c* to be a "space" character? You must select all the correct answers to get full marks; the retry penalty is 33%.

Select one or more:

- [ ] `c = '\32';`
- [x] `c = 32;` ✔
- [x] `c = ' '; /* The RHS is a quote char, a space and another quote */` ✔
- [x] `c = '\40';` ✔
- [ ] `c = ''; /* The RHS is just two quote chars */`
- [ ] `c = 0x40;`
- [ ] `c = 40;`
- [x] `c = 0x20;` ✔
- [x] `c = 040;` ✔

Correct

Marks for this submission: 1.00/1.00.

Question **2**

Correct

Mark 1.00 out of 1.00

What is the actual decimal numeric value of the symbol *EOF*?

Your answer must be written as a number, using only a subset of the characters '+, '-' and 0 through 9.

Answer: -1 ✔

The appropriate command to find the answer here was 'grep EOF /usr/include/stdio.h'.

Correct

Marks for this submission: 1.00/1.00.

Question **3**

Correct

Mark 1.00 out of 1.00

Why is the return type of getchar *int* rather than *char*?

Select one:

- a. It isn't -- the return type of *getchar* is actually *char*!
- b. Historical reasons only -- either would have been possible.
- c. A *char* variable only has 8 bits, and hence 256 states, which is not sufficient to accommodate the large set of all Unicode characters.
- d. In order to make it possible to distinguish the EOF case from the 256 possible *char* values. ✔

Correct

Marks for this submission: 1.00/1.00.

Question **4**

Correct

Mark 1.00 out of 1.00

Write a program that loops, reading characters from standard input and writing their decimal values to standard output one per line, until EOF occurs.

For example:

Input	Result
12345	49 50 51 52 53 10
Hi! Line 2	72 105 33 10 76 105 110 101 32 50 10

Answer: (penalty regime: 0, 10, ... %)

```
#include <stdio.h>
#include <stdlib.h>
// 0x = Octal \ = hexdeciaml int e.g. 32 = decimal
// %c and %d diff for printing EOF

int main(void)
{
    int c = 0;                  // An int with a char in its low byte "

    // Read and print characters until newline or End-Of-File
    c = getchar();              // Get char (cast to int) or EOF
    while ( c != EOF) {
        printf("%d\n", c);
        c = getchar();
    }

    return EXIT_SUCCESS;
}

```

	Input	Expected	Got	
✔	12345	49 50 51 52 53 10	49 50 51 52 53 10	✔
✔	Hi! Line 2	72 105 33 10 76 105 110 101 32 50 10	72 105 33 10 76 105 110 101 32 50 10	✔

	Input	Expected	Got	
✔	A B C	65 10 66 10 67 10	65 10 66 10 67 10	✔

Passed all tests! ✔

Correct

Marks for this submission: 1.00/1.00.

Question **5**

Correct

Mark 1.00 out of 1.00

[This program builds on the last one]

Write a program that loops, reading characters from standard input until EOF occurs. For each character read, a single line of output should be printed as follows:

1. If the character is a new-line character (which has the value '\012', i.e. 10 decimal), the printed line should be '\n' (including the quotes). Otherwise ...
2. The character itself should be printed, enclosed in single quotes followed by a colon and a space, followed by either:
   - `Digit <n>` if the character represents one of the digits 0 through 9, where <n> is the represented digit, or
   - `Letter <n>` if the character is one of the letters 'a' through 'z' or 'A' through 'Z', in which case <n> is the ordinal value of the letter, i.e. 1 for 'a' or 'A', 2 for 'b' or 'B' etc, or
   - `Non-alphanumeric` if the character is neither a digit nor a letter

Notes:

1. You may assume that no non-printing control characters (characters less than 32 decimal) other than newline are present in the input. This is to save you worrying about the messed-up output that you'd get from characters like Tab ('\t') and Return ('\r').
2. You will probably want to use various functions from ctype.h, such as *isdigit* and *isalpha*.
3. Remember that you can do arithmetic on characters so that 'E' - 'D' evaluates to 1.
4. Since backslash in *printf* format strings usually has a special meaning (as in '\n' or '\012'), if you wish to print a backslash you should 'escape' it with another backslash! For example, 'printf("\\Hi!\n") prints the string "\Hi!" followed by a newline.

For example:

Input	Result
Go ENCE260! End	'G': Letter 7 'o': Letter 15 ' ': Non-alphanumeric 'E': Letter 5 'N': Letter 14 'C': Letter 3 'E': Letter 5 '2': Digit 2 '6': Digit 6 '0': Digit 0 '!': Non-alphanumeric '\n' 'E': Letter 5 'n': Letter 14 'd': Letter 4 '\n'

Answer: (penalty regime: 0, 10, ... %)

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
// 0x = Octal \ = hexdeciaml int e.g. 32 = decimal
// %c and %d diff for printing EOF

int main(void)
{
    int c = 0;                  // An int with a char in its low byte "
    // Read and print characters until newline or End-Of-File           // Get char (cast to int) or
    c = getchar();
    while (c != EOF) {
        if (c ==  '\012') {
            printf("'\\n'\n");
        } else if (isalpha(c)) {
            if (c >= 91) {
                printf("'%c': Letter %d\n", c, (c - '`'));
            } else {
                printf("'%c': Letter %d\n", c, (c - '@'));
            }

        } else if (isdigit(c)) {
            printf("'%c': Digit %c\n", c, c);
        } else {
            printf("'%c': Non-alphanumeric\n", c);
        }
        c = getchar();
    }
    return EXIT_SUCCESS;
}
```

	Input	Expected	Got	
✔	Go ENCE260! End	'G': Letter 7 'o': Letter 15 ' ': Non-alphanumeric 'E': Letter 5 'N': Letter 14 'C': Letter 3 'E': Letter 5 '2': Digit 2 '6': Digit 6 '0': Digit 0 '!': Non-alphanumeric '\n' 'E': Letter 5 'n': Letter 14 'd': Letter 4 '\n'	'G': Letter 7 'o': Letter 15 ' ': Non-alphanumeric 'E': Letter 5 'N': Letter 14 'C': Letter 3 'E': Letter 5 '2': Digit 2 '6': Digit 6 '0': Digit 0 '!': Non-alphanumeric '\n' 'E': Letter 5 'n': Letter 14 'd': Letter 4 '\n'	✔

Passed all tests! ✔

Correct

Marks for this submission: 1.00/1.00.

Information

Arrays

Create a file *backwards.c* containing the following code, which is explained in the video *backwards.c*.

Hide `backwards.c` code

```
/* Simple array demo program.
 * Also demonstrates sizeof.
 * Reads a line of input characters into a 1D array
 * of 'char', then writes the line out backwards.
 * Written for ENCE260 by RJL.
 * February 2011, June 2013, July 2015, June 2016
 */

#include <stdio.h>
#include <stdlib.h>

#define N_MAX 100

int main(void)
{
    char line[N_MAX] = { 0 };     // An array of char, init'ed to zero
    int c = 0;                    // An int with a char in its low byte
    int n = 0;

    printf("Variable n requires %zu bytes of memory\n", sizeof n);
    printf("Array line occupies %zu bytes of memory\n", sizeof line);

    printf("Enter a line of text, terminated by 'Enter'\n");

    // Read characters until EOF, newline or buffer full

    c = getchar();              /* Get char (cast to int) or EOF */
    while (c != EOF && c != '\n' && n < N_MAX) {
        line[n] = c;
        n += 1;
        c = getchar();
    }

    // Now print out all those characters backwards

    printf("Your input line, written backwards, is:\n");
    for (int i = n - 1; i >= 0; i--) {
        putchar(line[i]);
    }
    putchar('\n');

    return EXIT_SUCCESS;
}
```

Build it and run it, typing a line of input text to it. You should see the line printed out backwards.

Note the following:

- If you're a Python programmer, arrays are superficially similar to fixed length lists, without the ability to add or remove elements. Also, all elements of the array must be of the same type specified in the array declaration.
- This program declares an array of items of type `char`, with size N_MAX (which is defined as 100 in this case).
- If you're a Java programmer, you'll be familiar with Java arrays, which are essentially objects that reside on the heap and are allocated with *new.* C arrays are much cruder; they are just a chunk of memory that (at least in the above case) resides in the current function's stack frame along with all the other local variables. This means the array space is "de-allocated" when the function returns, and also means that if you inadvertently run off either end of the array (subscripts aren't checked at runtime in C), you'll likely clobber some of your other local variables.
- As when declaring scalar variables, you are strongly advised to include an initialiser to set the initial value of all array elements. Array initialisers have the special syntax `{value0, value1, value2, ... valueN-1}`. This syntax is usable *only* in a declaration. Java uses a similar syntax in its *new* expressions, but unlike Java, C permits an initialiser to have fewer elements than the array. In that case, the rest of the array is set to zero. In the code above, the first value is set to the specified value (which is 0) and the rest of the array is initialised to zero. In C99 it is not permitted to have an empty initialiser list. However, if you don't include an initialiser at all, the array is (or may be) left uninitialised.
- The `sizeof` operator can be used to report how many bytes of memory a variable occupies. The value returned by sizeof is a 64-bit unsigned int on 64-bit machines or a 32-bit unsigned int on 32-bit machines. To print such numbers in a platform independent manner we use the '%zu' format specifier ('z' denotes 'si**Z**e type', 'u' denotes '**u**nsigned').
- C arrays can be indexed just like Java arrays and Python lists, with indices ranging from 0 up to the array size minus 1. However, subscripts aren't checked at runtime so you have to be careful to stay within that range or Bad Things will happen.
- The function `puts` ("put string") prints a single string, adding a newline at the end.

Although C arrays superficially look like Java arrays, they are much cruder. To get a first inkling of what I mean by that, set N_MAX to be just 1 (allocating a 1-byte array) and remove the condition `&& n < N_MAX` from the while loop. Run the program and type in the input string "Just testing!". What happens?

A more idiomatic version of the line-reading code

As in the *chario.c* program, C programmers would generally write the *while* loop more idiomatically as:

```
// Read characters until EOF, newline or buffer full
while((c = getchar()) != EOF && c != '\n' && n < N_MAX) {
    line[n++] = c;
}
```

Again, this uses the assignment operator within the while loop condition to fetch the character before testing it, thereby eliminating the need for the clumsy one-and-a-half loop. This example also uses the autoincrement operator to put the character into the output array and increment it in one statement. Both of these idioms are in such common use in the C community that they are deemed entirely acceptable in this course.

Question **6**

Correct

Mark 1.00 out of 1.00

Use *sizeof* to find how many bytes of memory a *double* variable occupies on the lab computers. What's the answer? [Write your answer as a number, i.e., using just the digits 0 - 9.]

Answer: 8 ✔

Correct

Marks for this submission: 1.00/1.00.

Question **7**

Correct

Mark 1.00 out of 1.00

From your experiment setting N_MAX to 1 and removing the check for array overflow, what do you conclude about C arrays?

Select one:

- C subscripts are not checked at runtime so programs can read from and write to memory that's outside the array, thereby possibly corrupting other unrelated variables. ✔ Yes that's right. All sorts of things can go wrong if you overflow an array: you may get rubbish output, your program may crash, or (worst of all) it may appear to work OK on simple test data but fail dramatically in the field.
- If you attempt to write outside the array bounds, nothing happens. If you attempt to read beyond the array bounds you always get zero.
- The instant you attempt to read or write beyond the array bounds you get a C exception thrown. If you wish to catch the exception, you have to use a try ... catch construct.
- C automatically reallocates memory space for an array if you attempt to overflow it.

Correct

Marks for this submission: 1.00/1.00.

Question **8**

Correct

Mark 1.00 out of 1.00

How many bytes would a C array *data* occupy on the lab computers if declared as

int data[100];

?

Select one:

- 100
- 200
- 400 ✔ As the program shows, *int*s require 4 bytes each, so the total requirement is 400 bytes.
- 800

Correct

Marks for this submission: 1.00/1.00.

Question **9**

Correct

Mark 0.90 out of 1.00

Use the combo boxes below to ensure that the two code fragments below are equivalent.

Code fragment, version 1 ...

```
char buff[100];
int c;
int i = 0;
while ((c = getchar()) != '\n' && i < 100) {
    buff[i++] = c;
}
```

Code fragment, version 2 ...

```
char buff[100];
int c;
int i = 0;
 c = getchar()          ✔ ;
while (  c != '\n' && i < 100    ✔ ) {
     buff[i] = c                          ✔ ;
     i = i + 1              ✔ ;
    c = getchar();
}
```

Correct

Marks for this submission: 1.00/1.00. Accounting for previous tries, this gives **0.90/1.00**.

Question **10**

Correct

Mark 1.00 out of 1.00

Write a C program that reads its standard input until EOF occurs and prints out a table of how many times each of the 26 letters in the alphabet occurred. Ignore all non-alphabetic characters. The program should be case-insensitive, i.e., each letter count includes both the lowercase and the uppercase letters.

Your program can use at most one *while* loop and one *for* loop.

Hints:

- The declaration `int counts[26] = {0}` could be useful!
- The function `isalpha(c)` returns a non-zero value if `c` is alphabetic or zero otherwise. It is declared in `ctype.h`.
- The function `toupper(c)` converts a character stored in an `int` value `c` to its uppercase equivalent if it has one. Otherwise it leaves the character unchanged. `toupper` is declared in `ctype.h`, too.
- You can do arithmetic on characters, which in C are treated as 8-bit integers. Hence, if `c` is a number representing an alphabetic uppercase character, `c - 'A'` is a number in the range 0 through 25 for `'A'` through `'Z'` respectively.
- Similarly, the expression `i + 'A'` evaluates to the characters `'A'` through `'Z'` for `i` in the range 0 to 25.
- When testing your program you can use the shell's input redirection operator `'<'` to have the program take its standard input from a file rather than from the keyboard. In `geany`, for example, you can redirect standard input to the file `testCounts.txt` by changing the *Execute* command from `"./%e"` to `"./%e" < testCounts.txt`.

Your program does not need to issue a prompt for the input - it just reads it (from *stdin*, e.g. using *getchar* or *getc* or *fgets* or *scanf* or ....).

Note that this program requires fewer than 20 lines of C code (not including comments and blank lines) - if you find yourself writing much more than this, ask a tutor for advice.

If you are testing with standard input coming from the keyboard, type CTRL/D at the start of a line to force an EOF condition (or two CTRL/Ds elsewhere in a line).

For example:

Input	Result

`The quick brown fox`
`Jumps over`
`The lazy dog.`
	`A: 1`
	`B: 1`
	`C: 1`
	`D: 1`
	`E: 3`
	`F: 1`
	`G: 1`
	`H: 2`
	`I: 1`
	`J: 1`
	`K: 1`
	`L: 1`
	`M: 1`
	`N: 1`
	`O: 4`
	`P: 1`
	`Q: 1`
	`R: 2`
	`S: 1`
	`T: 2`
	`U: 2`
	`V: 1`
	`W: 1`
	`X: 1`
	`Y: 1`
	`Z: 1`

Answer: (penalty regime: 0, 10, ... %)

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
// 0x = Octal \ = hexdeciaml int e.g. 32 = decimal
// %c and %d diff for printing EOF

int main(void)
{
    int counts[26] = {0};
    int c = 0;                  // An int with a char in its low byte "
    // Read and print characters until newline or End-Of-File           // Get char (cast to int) or
    c = getchar();
    while (c != EOF) {
        if (isalpha(c)) {
            if (c >= 91) {
                counts[c - 'a'] += 1;
            } else {
                counts[c - 'A' ] += 1;
            }
```

```
20
21          }
22          c = getchar();
23      }
24      for (int i = 0 ; i < 26; i++) {
25          printf("%c: %d\n", (i + 'A'), counts[i]);
26      }
27
28
29      return EXIT_SUCCESS;
30  }
31
```

	Input	Expected	Got	
✔	The quick brown fox Jumps over The lazy dog.	A: 1 B: 1 C: 1 D: 1 E: 3 F: 1 G: 1 H: 2 I: 1 J: 1 K: 1 L: 1 M: 1 N: 1 O: 4 P: 1 Q: 1 R: 2 S: 1 T: 2 U: 2 V: 1 W: 1 X: 1 Y: 1 Z: 1	A: 1 B: 1 C: 1 D: 1 E: 3 F: 1 G: 1 H: 2 I: 1 J: 1 K: 1 L: 1 M: 1 N: 1 O: 4 P: 1 Q: 1 R: 2 S: 1 T: 2 U: 2 V: 1 W: 1 X: 1 Y: 1 Z: 1	✔
✔	aBbcCcDDDDee eeefFfFfF	A: 1 B: 2 C: 3 D: 4 E: 5 F: 6 G: 0 H: 0 I: 0 J: 0 K: 0 L: 0 M: 0 N: 0 O: 0 P: 0 Q: 0 R: 0 S: 0 T: 0 U: 0 V: 0 W: 0 X: 0 Y: 0 Z: 0	A: 1 B: 2 C: 3 D: 4 E: 5 F: 6 G: 0 H: 0 I: 0 J: 0 K: 0 L: 0 M: 0 N: 0 O: 0 P: 0 Q: 0 R: 0 S: 0 T: 0 U: 0 V: 0 W: 0 X: 0 Y: 0 Z: 0	✔

Passed all tests! ✔

Correct

Marks for this submission: 1.00/1.00.

Information

Functions

Having a program comprised of just a single `main` function is workable only for trivial programs. Bigger programs need to be broken into a set of support functions plus the `main` function.

Program `average`

The following program, `average.c`, is taken from King section 9.2. Run it with some test data. The program is discussed in the video *Introducing Functions*.

```
/* Rather meaningless program demonstrating the use of functions.
 * Taken from the text by King, section 9.2, with slight modifications.
 */

#include <stdio.h>
#include <stdlib.h>

double average(double a, double b)
{
    return (a + b) / 2;
}

int main(void)
{
    double x = 0.0, y = 0.0, z = 0.0;
    printf("Enter three numbers: ");
    scanf("%lf%lf%lf", &x, &y, &z);
    printf("Average of %g and %g: %g\n", x, y, average(x,y));
    printf("Average of %g and %g: %g\n", y, z, average(y,z));
    printf("Average of %g and %g: %g\n", x, z, average(x,z));
    return EXIT_SUCCESS;
}
```

Note the following:

- The syntax for declaring a function is the same as the syntax for declaring a method in Java, but very different from Python's syntax. Since C uses static typing, like Java, the return type of the function must be explicitly specified, as must the types of all parameters.
- Java programmers: a C program is like a Java program with just a single (implicit) class: the functions we write in C correspond to a set of static support methods.
- Python programmers: a C program is like a Python program that consists only of a set of functions, and possibly some global variables (shudder), with no classes or separate modules. However, C functions are "static" - they are compiled by the compiler before execution begins. This is different from Python, where a function is an object, dynamically defined at run time.
- The declaration of the function must precede any calls to it (unlike Java). It would not be legal simply to move the declaration of the `average` function to the end of the program, although we'll see how to achieve that effect in the next subsection.
- The '%g' format specification is a 'general' format specification for floats - it is equivalent to '%f' except for very large or very small numbers which are displayed in scientific (exponent) form. For example, 0.000001 in '%g' format is displayed as 1.e-6, meaning 1.0×10^{6}.
- This program introduces the use of the *double* type for storing 64-bit floating point values. In the early days of C, computers only had a few kilobytes of memory and 64-bit data was considered an extravagant waste of memory unless you really needed high precision data. So the *float* data type, which used only 32-bits and represented numbers with about 7-digit precision, was considered adequate for most work. Nowadays we rarely bother using *float* unless we have very large data arrays (i.e. billions of elements).
- When inputting doubles with *scanf* you need to use format specifiers like `%lf` (denoting "long float") rather than `%f`. Remember that %d denotes decimal integer, not double.

Program `average2`

Consider the following variant of `average.c` called `average2.c`

```
/* Variant of average.c in which the function is separated into
 * declaration and definition. The declaration must precede any calls
 * to the function.
 * Taken from the text by King, section 9.2, with slight modifications.
 */

#include <stdio.h>
#include <stdlib.h>

// DECLARATION of the function average must precede its use
double average(double a, double b);

int main(void)
{
    double x = 0.0, y = 0.0, z = 0.0;
    printf("Enter three numbers: ");
    scanf("%lf%lf%lf", &x, &y, &z);
    printf("Average of %g and %g: %g\n", x, y, average(x,y));
    printf("Average of %g and %g: %g\n", y, z, average(y,z));
    printf("Average of %g and %g: %g\n", x, z, average(x,z));
    return EXIT_SUCCESS;
}

// DEFINITION of the function average now follows
double average(double a, double b)
{
    return (a + b) / 2;
}
```

You'll see that the function `average` has been moved to the end of the program, but a declaration of its *type signature* has been left at the top. This is called a `forward declaration`. Now when the compiler encounters the calls to `average` from within `main`, it already "knows" the types of the parameters and the return value and can therefore check the syntax appropriately. Note the terminology here: the *declaration* of a function specifies its signature, while the *definition* of a function defines its body i.e. the code that gets executed when it's called.

Edit your `average.c` program so it's the same as `average2.c`. Run it to convince yourself it works. Then comment out the forward declaration at the top. Try to make and run the program again. Study the output carefully. What is happening here?

Question **11**

Correct

Mark 1.00 out of 1.00

If you remove the forward declaration of the function *average* within the program *average2.c*, which of the following statements are true? It is assumed you are using *geany* with the compile options specified in lab 1.

[Select *all* true statements, not just one.]

Select one or more:

- [x] The conflict between the compiler's implicitly-assumed return type of the function *average* and its actual declared return type causes a compiler warning. Since we have the -Werror option set in the compile and build command(s), the warning is treated as an error and the program cannot be run. ✔
- [ ] The program compiles with warning messages but when run it outputs garbage answers.
- [x] When the compiler encounters a call to an undeclared function, it implicitly assumes the function returns an *int*. ✔
- [ ] The program still compiles and runs correctly, though with warning messages.

Correct

Marks for this submission: 1.00/1.00.

Question **12**

Correct

Mark 1.00 out of 1.00

Write a function with signature *double discriminant(double a, double b, double c)* that returns the discriminant of the quadratic with coefficients *a*, *b* and *c*, i.e., the value $b^2 - 4ac$. Paste *only* the function itself into the answer box; do not include any *#include* lines or test code.

For example:

Test	Result
`printf("%.2lf\n", discriminant(1, 2, 3))`	`-8.00`
`printf("%.2lf\n", discriminant(1.5, 1.5, 1.5))`	`-6.75`

Answer: (penalty regime: 0, 10, ... %)

```
1 double discriminant(double a, double b, double c)
2 {
3     return ((b*b) - (4 * a * c));
4 }
```

	Test	Expected	Got	
✔	`printf("%.2lf\n", discriminant(1, 2, 3))`	`-8.00`	`-8.00`	✔
✔	`printf("%.2lf\n", discriminant(1.5, 1.5, 1.5))`	`-6.75`	`-6.75`	✔

Passed all tests! ✔

Correct

Marks for this submission: 1.00/1.00.

Information

Local and external variables

Now we will start to consider how we might write functions that manipulate textual data. C doesn't have strings of the sort found in Python and Java. It only has arrays of char. So how do we pass such strings around, between functions? As a simple example, how might we write a function *readName* that reads someone's name (a string) from standard input. In languages like Python and Java we'd have the function or method return a string object so we could write, say

```
name = readName()
```

but we can't (easily) write functions that return strings in C. Instead we must preallocate memory for the required name and 'ask' readName to use that preallocated memory block.

There are two ways to preallocate the block: as a global variable (not nice) and as a local variable in the caller's stack frame. We'll look at these two options in turn.

Firstly consider the following program. `twiddle.c` which is explained in the video entitled *twiddle.c*.

Hide `twiddle.c` code

```
/* A program to demonstrate the (mis)use of external
 * variables. Reads a name (well, any string of chars,
 * really), converts it to upper case, then prints it
 * out.
 * Written for ENCE260, June 2011/2015/2018
 * Author: Richard Lobb
 */

#include <stdio.h>
#include <ctype.h>  // Various character-handling functions defined here

#define MAX_NAME_LENGTH  80
char name[MAX_NAME_LENGTH];  // declares a global (aka "external") variable

// Read a name (or any string) into the "name" array.
// Terminate it with null.
void readName(void)
{
    int c = 0;
    int i = 0;
    printf("Enter your name: ");
    while ((c = getchar()) != '\n' && c != EOF && i < MAX_NAME_LENGTH - 1) {
        name[i++] = c;
    }
    name[i] = 0;  // Terminator
}

// Convert the global "name" string to upper case
void convertNameToUpper(void)
{
    int i = 0;
    while (name[i] != '\0') {
        name[i] = toupper(name[i]);
        i++;
    }
}

// Main program reads name, converts it to upper case and prints it
int main(void)
{
    readName();
    convertNameToUpper();
    printf("Your name in upper case: %s\n", name);
}
```

The `name` array is an example of an *external* or *global* variable, i.e., one that is outside all the program functions and is visible to them all. Python programmers will be familiar with global variables (and perhaps with my attitude towards them!). Java programmers: if you regard a C program as corresponding to a single implicit Java main class, then the external variables are all the (static) fields of that class and the functions are all the (static) methods of that class. However, it is generally bad style in C to use external variables, and in ENCE260 you will be penalised for using them unless there is a good reason.

Some other points to note about program `twiddle`:

- The external `name` array has not been given an explicit initialiser. However, because it is external ("statically allocated") it will be initialised to zero, unlike the dynamically-allocated local variables of a function.
- The function `readName` reads characters into the external `name` array until either the array is full or a newline character is read. It then adds a null byte (`'\0'`) to the end of the characters it has read. This acts as a *sentinel* value, signalling the end of the string. C doesn't have a *String* datatype, but instead uses explicit arrays of chars, terminated by null bytes.

- Python programmers: note that the `while` loop syntax is similar to Python's except that the parentheses around the loop control condition are essential, and of course the loop body is enclosed in braces rather than being an indented block.
- The `convertToUpper` function does not explicitly check for an array overrun - it assumes that the null terminator byte is present before the end of the array is reached.
- The `printf` call uses a new format specifier: '%s', which denotes a string of characters is to be printed. The corresponding argument should be an array of `char`, terminated by a null byte.

Question **13**

Correct

Mark 1.00 out of 1.00

What would happen with program *twiddle* if *readName* did not add the null-byte terminator to the *name* string? [Warning: in C, questions of this sort cannot generally be reliably answered by experimentation. Suppose you were to make the change and the program worked fine. Would that prove it would always work? Certainly not in C! You need to think such questions through, instead.]

Select one:

- This particular program would still always work correctly. ✔ Indeed the program works *in this particular case*, because the array is located in the uninitialised globals area, which (despite its name) is initialised to zero before the program starts. However, this would be a dreadfully wrong answer if the array were local to a function. It would also be a dreadfully wrong answer if the program had multiple calls to *readName*, because data from previous calls would still be lying around in the global array.
  So ... you get 100%. But, did you deserve it, because you understood all the details? If so ... bravo! Or did you just run the program, discover it worked, and mindlessly tick this box? If the latter ... may your unterminated strings crash and burn in the next superquiz!!
- The program would either crash or print extra junk at the end of the upper case name. Sometimes, though, it might appear to work perfectly!
- The program wouldn't compile.
- The program would throw an exception at runtime.

Correct

Marks for this submission: 1.00/1.00.

Question **14**

Correct

Mark 1.00 out of 1.00

The program *twiddle* explicitly checks for an EOF on input. If the program is run under Linux with input coming from the keyboard, this isn't usually necessary, as the only way of forcing EOF is to type CTRL+D as the first character in a line (or by typing two CTRL+Ds anywhere else). Suppose, however, that standard input were taken from a file containing just the two characers "Hi" with no newline. What do you think would then happen if the program did *not* check for EOF, i.e., if the while loop were written as:

```
    while ((c = getchar()) != '\n' && i < MAX_NAME_LENGTH - 1) { ...
```

Again, be warned that answering this by experimentation rather than by thinking it through will probably get you the wrong answer. Why? Because the editor will probably add a new line character at the end of the file whether you type one or not, so your experiment won't actually work. However if you really really must generate a file containing just the two letters 'Hi' with no newline, you can do so with the command

```
echo -n "Hi" >testfile.dat
```

Select one:

- It would crash at run time within the `main` function.
- Its behaviour would be unpredictable, because it depends on the state of a memory area that has not been explicitly initialised.
- It would output the two characters "Hi" followed by some characters with an octal value of 0377. ✔ Yes, this is the right answer. When the end of file is encountered, *getchar* returns the value -1, and continues to return -1 on all subsequent calls. Because the code doesn't check for this, it will store the low byte of the returned int value into the array repeatedly, until it hits the end of the array. The low byte of -1 is 0377 (octal). When a character with octal value of 0377 is written to the terminal, its appearance will depend on what character encoding has been set for the terminal. If it's in extended ASCII mode the character 0377 will just be a "delete" character (invisible), if it's in Windows-1252 mode it will be a y-diaeresis (a 'y' with two dots over it) and if it's in UTF-8 mode the character is illegal and may be displayed as '#'.
- It would crash at run time within the function *readName*.
- It would output the two-character string "Hi" and nothing else.

Correct

Marks for this submission: 1.00/1.00.

Information

An exercise with functions

Program `twiddle2.c`

Below is an improved version of `twiddle` that doesn't use external variables. However, it's missing one of the functions required to make it work. Your job is to get it working by writing the missing function. The skeleton of the program, without the code that converts a name to upper case, is explained in the video twiddle2.c

Hide `twiddle2.c` code

```
/* An improved (but incomplete) version of twiddle.c that
 * doesn't use external (global) variables.
 * Written for ENCE260 June 2011/2013/2018
 * Author: Richard Lobb
 */

#include <stdio.h>
#include <ctype.h>

#define MAX_NAME_LENGTH 80

// Read a name (or any string) into the parameter array.
// Terminate it with null.
void readName(int maxLen, char name[])
{
    int c = 0;
    int i = 0;
    printf("Enter your name: ");
    while ((c = getchar()) != '\n' && c != EOF && i < maxLen - 1) {
        name[i++] = c;
    }
    name[i] = '\0';  /* terminator */
}


int main(void)
{
    char name[MAX_NAME_LENGTH];
    readName(MAX_NAME_LENGTH, name);
    convertStringToUpper(name);
    printf("Your name in upper case: %s\n", name);
}
```

Before starting, though, note the following important points about parameters:

- Parameters in C are always passed by value (i.e., parameters behave like local variables whose initial value is given by the passed-in argument) *except* that arrays are passed by *reference*. Hence, if a function alters the data in an array that is passed as a parameter, the data back in the calling function gets changed too, because it's the same array. [Note for enthusiasts only: actually, we'll see in the next chapter of the course that array parameters are actually just *pointers*, which are passed by value. Passing a pointer to the base of an array by value is equivalent to passing the array by reference.]
- When declaring a one-dimensional array parameter, we generally omit the size of the array and use a syntax like `char name[]`. If you include a size within the brackets it will simply be ignored by the compiler, since the called function doesn't need to allocate space for the array in its local stack frame. [Remember: the array was passed by reference.]
- A function (such as `readName`) has no way of determining the size of a passed-in array, i.e., there is no equivalent to Python's `len` function or Java's `length` field of an array. Hence, when passing arrays to functions we must always either explicitly pass in the length too (as in the case of the `readName` function or include a sentinel value within the passed array (as in the case of the `convertStringToUpper` function that you have to write, which takes a null-terminated array).
- While the function `readName` could have been written to use the `MAX_NAME_LENGTH` macro rather than taking an explicit `maxLen` parameter, that would still be poor style. `MAX_NAME_LENGTH` is like an external variable in the sense that the function couldn't simply be copied into some other program, which would in general not have such a macro defined.

Now - go to it! Get that missing function written and tested, ready for the following quiz questions.

Question **15**

Correct

Mark 1.00 out of 1.00

Which of the following statements about *twiddle2.c* are correct? You must select all correct answers to get full marks.

Select one or more:

- [ ] The return type of the missing function is *char []*.
- [ ] The missing function needs to make use of the MAX_NAME_LENGTH macro.
- [ ] The missing function takes two parameters: an array of char and the length of that array.
- [x] The missing function is *convertStringToUpper*. ✔
- [x] The return type of the missing function is *void*. ✔

Correct

Marks for this submission: 1.00/1.00.

Question **16**

Correct

Mark 1.00 out of 1.00

Paste your code for the *convertStringToUpper* function into the answer box. Paste *only* the function itself, no other code. The *toupper* function will be available - you don't need any #includes.

For example:

Test	Result

`char testString[] = "aB0X?";`
`convertStringToUpper(testString);`
`printf("%s\n", testString);`
	AB0X?

Answer: (penalty regime: 0, 10, ... %)

```
void convertStringToUpper(char name[])
{
    for (int doo = 0; (name[doo] != '\0'); doo++) {
        name[doo] = toupper(name[doo]);
    }
}
```

	Test	Expected	Got	

✔	`char testString[] = "aB0X?";`
	`convertStringToUpper(testString);`
	`printf("%s\n", testString);`
		AB0X?	AB0X?	✔

✔	`char testString[] = "Ab0x?";`
	`convertStringToUpper(testString);`
	`printf("%s\n", testString);`
		AB0X?	AB0X?	✔

Passed all tests! ✔

Correct

Marks for this submission: 1.00/1.00.

Question **17**

Correct

Mark 1.00 out of 1.00

Write a function with signature `int countMatches(int n, int data[], int searchValue)` that returns a count of the number of times the given `searchValue` is found in the *n*-element array `data`. [This was a question from the 2015 test.]

For example:

Test	Result
`int nums[] = {1, 2, 3, 4, 1, 1, 5};` `printf("%d\n", countMatches(7, nums, 1));`	3
`int nums[] = {10, 20, 30};` `printf("%d\n", countMatches(3, nums, 1));`	0

Answer: (penalty regime: 0, 10, ... %)

```
int countMatches(int n, int data[], int searchValue)
{
    int counter = 0;
    for (int position = 0; (position <= n); position++) {
        //printf("%d %d %d\n", data[position], searchValue, counter);
        if (data[position] == searchValue) {
            counter += 1;
        }
    }
    return counter;
}

```

	Test	Expected	Got	
✔	`int nums[] = {1, 2, 3, 4, 1, 1, 5};` `printf("%d\n", countMatches(7, nums, 1));`	3	3	✔
✔	`int nums[] = {10, 20, 30};` `printf("%d\n", countMatches(3, nums, 1));`	0	0	✔

Passed all tests! ✔

Correct

Marks for this submission: 1.00/1.00.

Information

Finally: 2D arrays

C also has multidimensional arrays, i.e., arrays with 2 or more indices. For example, a noughts and crosses game ("tic-tac-toe" to Americans), won by 'X', might be represented as

```
char game[3][3] = {{'X', '.', '.'},
                   {'O', 'X', '.'},
                   {'O', 'O', 'X'}};
```

and such a game could be printed out with a pair of nested loops as

```
for (int row = 0; row < 3; row++) {
    for (int col = 0; col < 3; col++) {
        printf("%3c", game[row][col]);
    }
    printf("\n");
}
```

However, don't be fooled into thinking this is a "real" 2D array, like an array of arrays in Java or a list of lists in Python. A 2D array in C is just a linear chunk of memory – essentially a 1D array – consisting of row 0, followed by row 1, followed by row 2, etc. The compiler does the simple indexing calculation to convert the double subscript to a single subscript. A consequence of this is that if you run off the end of one row you just land in the next one. For example, the following bit of code does actually correctly display the noughts and crosses game:

```
for (int col = 0; col < 9; col++) {
    printf("%3c", game[0][col]);
    if (col % 3 == 2) {
        printf("\n");  // Add a newline after every 3 numbers
    }
}
```

But please don't do horrible things like that in ENCE260.

Question **18**

Correct

Mark 1.00 out of 1.00

The code to check if a noughts and crosses game has been won by the the player who just moved might be (inelegantly) written as:

```
isWonGame =   isWonRow(player, game, 0)   /* Row 0 */
          ||  isWonRow(player, game, 1)   /* Row 1 */
          ||  isWonRow(player, game, 2)   /* Row 2 */
          ||  isWonCol(player, game, 0)   /* Col 0 */
          ||  isWonCol(player, game, 1)   /* Col 1 */
          ||  isWonCol(player, game, 2)   /* Col 2 */
          ||  isWonDiag(player, game, 0)  /* Diag 0 */
          ||  isWonDiag(player, game, 1); /* Diag 1 */
```

This uses support functions `isWonRow`, `isWonCol` and `isWonDiag`, which check whether the given player in the given game occupies all three squares in the given row, column or diagonal respectively.

Write the function `int isWonRow(char player, char game[3][3], int rowNum)`, which returns 1 or 0 according to whether the row is won by that player or not.

For example:

Test	Result
`char game[3][3] = {{'X', 'O', ' '},{'X', 'X', 'X'}, {' ', ' ', ' '}};` `printf("%d\n", isWonRow('X', game, 1));`	1
`char game[3][3] = {{'X', 'O', ' '},{' ', ' ', ' '}, {'X', 'X', 'O'}};` `printf("%d\n", isWonRow('X', game, 2));`	0

Answer: (penalty regime: 0, 10, ... %)

```
1  int isWonRow(char player, char game[3][3], int rowNum)
2  {
3      int i = 0;
4      int isWon = 0;
5      for(i = 0; i < 3; i++) {
6          if (game[rowNum][i] == player) {
7              isWon = 1;
8          } else {
9              isWon = 0;
10         }
11     }
12     return isWon;
13 }
```

	Test	Expected	Got	
✔	`char game[3][3] = {{'X', 'O', ' '},{'X', 'X', 'X'}, {' ', ' ', ' '}};` `printf("%d\n", isWonRow('X', game, 1));`	1	1	✔
✔	`char game[3][3] = {{'X', 'O', ' '},{' ', ' ', ' '}, {'X', 'X', 'O'}};` `printf("%d\n", isWonRow('X', game, 2));`	0	0	✔
✔	`char game[3][3] = {{'X', 'O', ' '},{' ', ' ', ' '}, {'O', 'O', 'O'}};` `printf("%d\n", isWonRow('O', game, 2));`	1	1	✔
✔	`char game[3][3] = {{'X', 'O', ' '},{' ', ' ', ' '}, {'O', 'O', 'O'}};` `printf("%d\n", isWonRow('O', game, 0));`	0	0	✔
✔	`char game[3][3] = {{'X', 'X', 'X'},{' ', ' ', ' '}, {'O', 'O', 'O'}};` `printf("%d\n", isWonRow('O', game, 0));`	0	0	✔

Passed all tests! ✔

Correct

Marks for this submission: 1.00/1.00.

Jump to...

◄ Quiz 2: Expressions and Flow Control

Quiz 4: Pointers and Strings ►