Started on	Friday, 19 July 2019, 4:12 PM
State	Finished
Completed on	Thursday, 1 August 2019, 12:00 AM
Time taken	12 days 7 hours
Marks	12.90/13.00
Grade	**9.92** out of 10.00 (**99**%)

Information

Expressions and flow control

Goals

This lab covers the use of C's expressions, selection and iteration constructs and gives you more practice in writing simple C programs. By the end of the lab you should:

- be familiar with C expression syntax, including pre- and post- increment and decrement operators
- be familiar with C's notion of conditional expressions
- be able to use *if* statements, with or without *else* clauses
- be acquainted with *switch* statements
- be able to use *while*, *do* and *for* loops
- be able to use nested selection and iteration constructs
- be able to use all the above in writing a simple C program

This lab covers the material in chapters 4, 5, 6 and 7 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).

You should watch the week 2 videos before attempting this lab, and you might also wish to revise the video on Memory, Data and Assignment.

Information

Integer data

Note: the video on Memory, Data and Assignment introduces this material.

Integer values in C are usually just declared as being of type *int*, e.g.

```
int i = 0;
```

The C standard leaves it to the implementer to decide how many bytes of memory should be used for an *int*, but typically 16-bit *ints* are used on small 8- or 16-bit architectures (nowadays used only in micro-controllers, such as the one you'll be meeting in term 4), while 32-bit *ints* are used with 32-bit and 64-bit architectures.

In lab 1 you saw that the code

```
int n = 60000;
printf("Big number = %d\n", n * 60000);
```

didn't print the answer 3,600,000,000 as you might have expected. [What did it print?]

Hopefully you realised that the problem was that the expected answer of 3,600,000,000 was too big to fit within a signed 32-bit *int* (which is limited to the range -2,147,483,648 to 2,147,483,647; this will be explained in tutorials and in the computer architecture segment of the course).

As a C programmer, you have lots of options available to you when declaring integer datatypes. For example, on our lab machines:

Type	Size
`char`	8-bit
`short`	16-bit
`int`	32-bit
`long`	64-bit

You can also prefix the type by either `signed` or `unsigned` according to whether you need to represent negative numbers or not, respectively. Thus for example if you declare

```
unsigned int n;
```

n can be in the range 0 to 4,294,967,295, i.e., it's always non-negative.

Another important type is returned by both the *sizeof* operator (see below) and the *strlen* (string-length) function. This type is *size_t*. Since you can't have negative-sized data, it is an unsigned integer and since the data could in theory occupy all possible memory, the type has to be wide enough to accommodate the maximum memory size. On a 64-bit architecture, *size_t* is thus a 64-bit integer.

To print unsigned integers in decimal, use %u instead of %d:

```
unsigned int n = 60000;
printf("Big number = %u\n", n * 60000);
```

Question: what does the above code snippet print if you use %d instead?

To find out how many bytes any variable occupies you can use the *sizeof* operator. For example, the C statement

```
printf("n occupies %zu bytes of memory\n", sizeof n);
```

will report how many bytes the variable *n* requires. The format *%zu* is used for printing *size_t* values in a platform-independent manner (it denotes "unsigned size-type").

Question **1**

Correct

Mark 1.00 out of 1.00

Match up the integer data types with the range of values they can represent. Assume a 64-bit architecture, like our lab machines. [You're not expected to check or even know how to work out the actual numbers - you may assume that all supplied answers match the given types.]

`unsigned short`	0 to 65,535 ✔
`signed char`	-128 to 127 ✔
`unsigned int`	0 to 4,294,967,295 ✔
`long long`	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 ✔
`signed int`	-2,147,483,648 to 2,147,483,647 ✔
`unsigned char`	0 to 255 ✔
`signed short`	-32,768 to 32,767 ✔
`size_t`	0 to 18,446,744,073,709,551,615 ✔

Your answer is correct.

Correct

Marks for this submission: 1.00/1.00.

Information

Expressions

Expressions in C are roughly the same as in Python with the following important differences. (See the video on expressions here).

1. The divide operator, `'/'` with integer operands behaves like the Python 2 version, not the Python 3 version, i.e., it yields an integer. For example 3 / 7 evaluates to 0.
2. In logical expressions, `&&`, `||` and `!` are used in lieu of `and`, `or` and `not`.
3. There is no exponentiation operator (i.e., no equivalent to Python's `**` operator).
4. The increment and decrement operators, `'++'` and `'--'`, allow pre- and post- incrementing and decrementing of variables during expression evaluation. For example:

```
int i = 1;
int j = 0;
j = ++i + 10;  /* Sets j to 12, i to 2 */
i = 1;         /* Reset i to 1 */
j = i++ + 10;  /* Sets j to 11, i to 2 */
```

5. In Python and Java an assignment operator can appear only in an assignment statement. In C there is no such thing as an assignment statement. Instead, any expression can be used as a statement and the assignment operator, `'='`, is a general-purpose operator that can be used (multiple times, if required) within any expression. For example, in C you can write:

```
int c = 0;
while ((c = getchar()) != EOF) {
     // Some code that uses the variable c
}
```

This is a loop that repeatedly calls the `getchar` function to get a character from standard input, assigns the result to the variable `c` and then executes the body of the loop until the character just read is equal to the sentinel end-of-file value EOF (which is *#define*-d in *stdio.h* to be the value -1). It is an extension of the more obvious but useless loop

```
while (getchar() != EOF) {
    // Can't do anything useful here because we didn't
    // save the character that getchar() returned!
}
```

Create a file *expressions.c* containing the code below, study it, run it and make sure you understand the output.

```
/* A program to illustrate increment, decrement and assignment operators.
 * While you're expected to understand how the operators work,
 * DON'T WRITE CODE LIKE THIS IN ENCE260!
 * Richard Lobb, June 2012.
 */

#include <stdio.h>

int main(void)
{
    int i = 1;
    int j = 10;
    int k = 100;

    printf("Initially, i, j, k = %d, %d, %d\n", i, j, k);
    j = i++;
    printf("After j = i++, i, j, k = %d, %d, %d\n", i, j, k);
    k = ++i;
    printf("After k = ++i, i, j, k = %d, %d, %d\n", i, j, k);
    i = --k;
    printf("After i = --k, i, j, k = %d, %d, %d\n", i, j, k);
    j = i--;
    printf("After j = i--, i, j, k = %d, %d, %d\n", i, j, k);
    i = (j = 20) + (k = 30);
    printf("i = (j = 20) + (k = 30), i, j, k = %d, %d, %d\n", i, j, k);

    return 0;
}
```

Although you're expected to understand how the `=`, `++` and `--` operators work, you shouldn't write dreadful code like the above. Injudicious use of these "side-effect" operators is liable to confuse the reader or, worse, lead to undefined results. [See the text for examples of situations where the C standard doesn't even define how an expression will be evaluated!] I personally recommend you don't even use the operators `++` and `--`, but use `+=` and `-=` instead. However, if you do use them — and certainly they're standard throughout the C community — then the ENCE260 style guideline is to use them only in isolation, i.e. in expressions that involve no other operators. Similarly, the assignment operator should generally be used only in statements of the form

```
   lvalue = [lvalue = ]... expression
```

where *lvalue*, meaning "left-hand-side value", is an expression that can be the target of an assignment, such as a simple variable, an indexed variable, or a pointer expression. [Note: the above syntax definition uses the standard Linux "man page" conventions, where square brackets denotes an optional element and an ellipsis ('...') denotes an arbitrary number of repetitions.]

For example,

```
result = 23 * 5;     // acceptable
i = j = k = 10 * n;  // acceptable
i++;                 // acceptable
--j;                 // acceptable
data[i = 5] = 2;     // unacceptable
k = --n;             // unacceptable
data[i++] = n;       // breaks the above rule but acceptable
while ((c = getchar()) != EOF) { ... } // also acceptable
```

The last two lines are exceptions to the usual rule but are both so convenient and in such universal use in the C community that they are considered acceptable in ENCE260.

Question **2**

Correct

Mark 1.00 out of 1.00

Match up each of the following statements with the resulting values of i, j and k, given that before each statement is executed i, j and k have the values 1, 10 and 100 respectively. For example, if a statement had no effect you would choose the answer 1, 10, 100.

`i = j = k = 1;`	1, 1, 1 ✔
`i = j++ + k++;`	110, 11, 101 ✔
`i = ++j + ++k;`	112, 11, 101 ✔
`i = j = k;`	100, 100, 100 ✔
`j = k = i;`	1, 1, 1 ✔
`i = ++j + k--;`	111, 11, 99 ✔
`i = ++j + --k;`	110, 11, 99 ✔

Correct

Marks for this submission: 1.00/1.00.

Question **3**

Correct

Mark 0.90 out of 1.00

According to the style guidelines you've just been given, which of the following statements are acceptable for use in ENCE260?

Select one or more:

- [ ] a. `i = 1 + (j = 5);`
- [x] b. `++k;` ✔
- [ ] c. `i = j++;`
- [x] d. `i = j = k = 1;` ✔
- [x] e. `j--;` ✔
- [x] f. `i = j = k;` ✔
- [ ] g. `i = ++j;`
- [x] h. `--k;` ✔

Correct

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

Information

if statements

Note: *if* statements are introduced in the *Conditionals and astyle* video.

Type the following program into a file `selection.c`, referring to the notes below as you do so. Run the program a few times, typing in different input, until you're sure you understand it.

```
/* A program to illustrate if and switch statements.
 * Richard Lobb, July 2015.
 */

#include <stdio.h>

int main(void)
{
    int value = 0;
    printf("What's the pip-value of your playing card? ");
    scanf("%d", &value);
    if (value < 1 || value > 13) {   ①②③
        puts("No such card.");    ④
    } else if (value == 1) {   ⑤
        puts("It's an ace.");
    } else if (value < 11) {
        puts("It's a boring number card.");
    } else {
        printf("It's a face card. Specifically a ");
        switch (value) {   ⑥
        case 11:
            puts("Jack.");
            break;   ⑦
        case 12:
            puts("Queen.");
            break;
        case 13:
            puts("King.");
        }
    }
}
```

1. The condition expression in an *if* statement must be enclosed in parentheses.
2. The condition expression is evaluated to yield a number; 0 is false, anything else is true.
3. ENCE260 style rules require you to enclose the statements to be conditionally executed in braces, even when there's only one statement.
4. Do not fall into the trap of putting a semicolon straight after the condition, i.e. do not write *if (...); { ...* . [What goes wrong if you do that? ***Richard's TODO*** Have a question on this next year.]
5. You *must* indent your code properly (i.e., as in Python) in ENCE 260. Indent 4 spaces per level, please. Incorrectly indented code will be rejected by CodeRunner. See the info panel on *astyle* later in this quiz.
6. There's no explicit *elseif* syntax in C, the same effect is created by using a nested if statement, but without the braces and additional indentation following *else*.
7. There is a *switch* statement in C, but it's pretty horrible and rarely preferable to an *if* statement.
8. If you do use a *switch* statement, don't forget to put a *break* statement at the end of the statements for each case except the last. Without the break, execution just "falls through" into the following code block, then the one after that, etc. until the end of the switch statement or a break is encountered.

Question **4**

Correct

Mark 1.00 out of 1.00

What's the output from the following syntactically valid code fragment? The 'putchar' function prints a single character to standard output.

```
int value = 11;
switch (value) {
case 11:
    putchar('J');
case 12:
    putchar('Q');
case 13:
    putchar('K');
}
```

Answer: JQK ✔

That's right. Horrible, eh?!

Correct

Marks for this submission: 1.00/1.00.

Information

Unlike Python, the layout of your C code has no semantic significance. You can add and remove white space, including newline characters, just about anywhere in the code except within strings without altering the behaviour of your program. However, good layout is an important component of good style and is vital for readability, both by you and by other programmers who may need to inspect or maintain your code.

There are many different layout conventions for C code. The Wikipedia Indent_style page lists the 8 main variants, although there are many subvariants. When you are writing code for a particular company or project you will generally be told which style to use, to ensure consistency of coding style across the organisation or project. In ENCE260 the specified coding convention is a variant of the original Kernighan and Ritchie (K&R) convention called *1TBS* or "One True Brace Style"; see here.

The Linux lab machines have an installed program called *astyle* (http://astyle.sourceforge.net/) to make it easy for you to layout your code. *astyle* supports many different conventions; see http://astyle.sourceforge.net/astyle.html for details.

The astyle program on our lab machines *should* be properly set up for our standard. If not, or if you're setting up a home machine, see the instructions below.

When astyle is correctly set up, you can fix your program layout at any time by selecting all the code (CTRL/A) and then choosing the menu option

```
Edit > Format > Send selection to > astyle
```

Alternatively, just type CTRL/A CTRL/1.

Almost all CodeRunner questions will check that your code is laid out in the astyle-approved way and will refuse to run it if it is not. The *Precheck* button will allow you to check astyle compatibility (and various other local style rules) before you click *Check*.

Setting up *astyle* in geany on a home machine

Firstly install the *astyle* program with the command

```
sudo apt install astyle
```

Then you can run *geany*, open a C file, and set up the astyle command as follows.

Using the program menus, select

```
Edit > Format > Send selection to > Set Custom Commands
```

Delete the existing *astyle* command and *Add* a new command (giving a label *astyle*):

```
bash -c "astyle --style=1tbs --indent-labels"
```

Quit from geany and restart.

Question **5**

Correct

Mark 1.00 out of 1.00

Write a program that uses *scanf* to read a single integer from standard input and prints 'Odd' if the integer is odd, 'Even' if it's a non-zero even number, or 'Zero' if it's zero. You may assume that the program will be tested only with valid numeric input.

For example:

Input	Result
1	Odd
-122	Even
0	Zero

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

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

int main(void)
{
    int value = 0;
    scanf("%d", & value);
    if (value == 0) {
        puts("Zero");
    } else if ((value % 2) == 0) {
        puts("Even");
    } else if ((value % 2) == 1) {
        puts("Odd");
    }
}

```

	Input	Expected	Got	
✔	1	Odd	Odd	✔
✔	-122	Even	Even	✔
✔	0	Zero	Zero	✔
✔	11	Odd	Odd	✔
✔	1234567	Odd	Odd	✔
✔	-123456	Even	Even	✔

Passed all tests! ✔

Correct

Marks for this submission: 1.00/1.00.

Question **6**

Correct

Mark 1.00 out of 1.00

Write a C program that reads three white-space separated floating point values *a, b* and *c* from standard input and prints the solution to the quadratic equation $ax^2 + bx + c = 0$ as follows:

1. If the equation is not a quadratic (because *a* is zero), the output is

```
Not a quadratic
```

2. If the roots are imaginary the output is

```
Imaginary roots
```

3. Otherwise, the output is of the form

```
Roots are -1.0000 and 1.0000
```

where both roots are printed with 4 decimal digits after the decimal point. **The smaller root should be printed first.**

Notes:

1. The solution to $ax^2 + bx + c = 0$ is

$$x = \frac{-b \pm \sqrt{(b^2 - 4ac)}}{2a}$$

2. If the two roots are equal, just print them both - don't treat it as a special case.
3. In order to use the *sqrt* function, you have to tell the compiler what type it is, i.e. what type parameters it takes and what type it returns. This is done by adding the line `#include <math.h>` right after the `#include <stdio.h>` line. That's all you have to do when submitting to the quiz server. However, when you're building the program on a lab machine you also need to tell the linker what libraries it must search to find the actual code for `sqrt`. To do this, add the option `-lm` (that's a lower-case L not the number one) *at the end of* the compile-and-build command line.
4. If you use *doubles* for the quadratic coefficients a, b, c, remember that you must use *%lf* rather than *%f* in the scanf format specifier. [Again, that's an 'el' not the number one. Conversely, if you're using *floats* DON'T use *%lf*!]
5. Remember the motto DRY ("Don't Repeat Yourself"). Writing multiple copies of the expression $\sqrt{(b^2 - 4ac)}$ is ugly and error prone.
6. The variable names *a*, *b* and *c* are allowed in this question.

For example:

Input	Result
`1.0 -4.0 3.0`	`Roots are 1.0000 and 3.0000`
`1.0 2.0 3.0`	`Imaginary roots`
`0.0 2.0 3.0`	`Not a quadratic`
`1.0 0.0 -1.0`	`Roots are -1.0000 and 1.0000`

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

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
int main(void)
{
    float Var1 = 0.0;
    float Var2 = 0.0;
    float Var3 = 0.0;
    float Root1;
    float Root2;
    float Determinant;
    scanf("%f", & Var1);
    scanf("%f", & Var2);
    scanf("%f", & Var3);
    Determinant = ((Var2 * Var2) - (4 * Var1 * Var3));
    if (Var1 == 0) {
        puts("Not a quadratic");
    } else if (Determinant < 0.0 ) {
        puts("Imaginary roots");
    } else {


        Root1 = (-Var2 - sqrt((Var2 * Var2)- (4 * Var1 * Var3)))/(2 * Var1);
        Root2 = (-Var2 + sqrt((Var2 * Var2)- (4 * Var1 * Var3)))/(2 * Var1);
        if (Root1 < Root2) {
            printf("Roots are  %.4f and %.4f", Root1, Root2);
        } else {
            printf("Roots are  %.4f and %.4f", Root2, Root1);
        }
    }
}
```

```
31 }
32
33
```

	Input	Expected	Got	
✔	1.0 -4.0 3.0	Roots are 1.0000 and 3.0000	Roots are 1.0000 and 3.0000	✔
✔	1.0 2.0 3.0	Imaginary roots	Imaginary roots	✔
✔	0.0 2.0 3.0	Not a quadratic	Not a quadratic	✔
✔	1.0 0.0 -1.0	Roots are -1.0000 and 1.0000	Roots are -1.0000 and 1.0000	✔
✔	-1.0 0.0 1.0	Roots are -1.0000 and 1.0000	Roots are -1.0000 and 1.0000	✔
✔	4.0 1.0 -3.0	Roots are -1.0000 and 0.7500	Roots are -1.0000 and 0.7500	✔

Passed all tests! ✔

Correct

Marks for this submission: 1.00/1.00.

Information

Conditional expressions

Although we didn't teach it in COSC121, Python has a *conditional expression* syntax that allows you to have the equivalent of *if* statements embedded in expressions. For example, in Python you can write:

```
freezingPoint = 32 if isFahrenheit else 0
```

as a more compact alternative to

```
if isFahrenheit:
    freezingPoint = 32
else:
    freezingPoint = 0
```

C has conditional expressions too, and they're more widely used than Python's, although the syntax is more cryptic. In C the above statement would be written:

```
freezingPoint = isFahrenheit ? 32 : 0;
```

The '?' ... ':' construct is sometimes referred to as *the ternary operator* as it takes three operands: the expression, the value to use when true and the value to use when false.

Question **7**

Correct

Mark 1.00 out of 1.00

Although we haven't actually introduced C functions yet, you can probably appreciate that the code shown below defines a C function that returns a value of 42 if *boink* is equal to *flunk* or the value *flunk* - 11 otherwise.

Rewrite the function with just a single statement by using a conditional expression i.e., by using the "? ... :" ternary operator. The answer box has been preloaded with the correct form for the function definition - all you have to do is replace the comment with an appropriate one-line statement. **Hint:** the statement will be a *return* statement.

```
int gringe(int boink, int flunk) {
    int floodle = 0;
    if (boink == flunk) {
        floodle = 42;
    } else {
        floodle = flunk - 11;
    }
    return floodle;
}
```

For example:

Test	Result
`printf("%d\n", gringe(23, 23));`	42
`printf("%d\n", gringe(23, 24));`	13

Answer: (penalty regime: 0, 10, 20, ... %)

Reset answer

```
1 int gringe(int boink, int flunk)
2 {
3     return ((boink == flunk) ? 42: (flunk - 11));
4 }
```

	Test	Expected	Got	
✔	`printf("%d\n", gringe(23, 23));`	42	42	✔
✔	`printf("%d\n", gringe(23, 24));`	13	13	✔
✔	`printf("%d\n", gringe(24, 23));`	12	12	✔
✔	`printf("%d\n", gringe(52, 53));`	42	42	✔
✔	`printf("%d\n", gringe(53, 53));`	42	42	✔
✔	`printf("%d\n", gringe(54, 54));`	42	42	✔

Passed all tests! ✔

Correct

Marks for this submission: 1.00/1.00.

Information

while loops

Note: *while*, *do* and *for* loops are introduced in the Loops video.

C `while` loops are very similar to Python `while` loops, except that the loop condition must be enclosed in parentheses. The body of the while loop is a *block*, i.e., a sequence of declarations, followed by a sequence of statements, all enclosed in braces. For example:

```
/* Print a geometric progression, with the start value, multiplier,
 * and number of elements read from standard input.
 * Written by Richard Lobb, June 2012/July 2019, for ENCE 260.
 */
#include <stdio.h>
int main(void)
{
    int numTerms = 0;
    int value = 1;
    int multiplier = 1;
    int i = 0; // Number of terms printed so far
    printf("Enter value, multiplier and number of terms to print: ");
    scanf("%d %d %d", &value, &multiplier, &numTerms);
    while (i < numTerms) {
        printf("%d\n", value);
        value *= multiplier;
        i++;
    }
    return 0;
}
```

Question **8**

Correct

Mark 1.00 out of 1.00

Write a program that uses a *while* loop to read integers from standard input, printing each one on a line by itself, until the number 42 has been printed. At that point the program should quit. You may assume that only integers separated by white space will appear in the input and that the number 42 will appear somewhere. [Hint: read the numbers with *scanf* and a format specifier of just "%d". Each call to scanf will get you the next number from standard input; you don't need to worry about things like line breaks, extra white space, etc.]

Important: you are *required* to use a *while* loop and you must not use *for* or *break*.

For example:

Input	Result
10 -1 3 42 7	10 -1 3 42
42 43 44 45	42

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

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

int main(void)
{
    int values = 0;
    bool found;
    found = false;
    while (found == false) {
        scanf("%d", &values);
        printf("%d\n", values);
        if (values == 42) {
            found = true;
        }

    }

}

```

	Input	Expected	Got	
✔	10 -1 3 42 7	10 -1 3 42	10 -1 3 42	✔
✔	42 43 44 45	42	42	✔
✔	-42 +42 42 3	-42 42	-42 42	✔
✔	3 4 5 41 42 43	3 4 5 41 42	3 4 5 41 42	✔

Passed all tests! ✔

Correct

Marks for this submission: 1.00/1.00.

Question **9**

Correct

Mark 1.00 out of 1.00

Write a program that uses a *do...while* loop (**not** an ordinary *while* loop) to read integers from standard input, printing each one on a line by itself, until the number 42 has been printed. At that point the program should quit. You may assume that only integers separated by white space will appear in the input and that the number 42 will appear somewhere. [Hint: read the numbers with *scanf* and a format specifier of just "%d".]

You are not permitted to use a *for* loop or a basic *while* loop in this program. Nor may you used *break* anywhere.

For example:

Input	Result

`10 -1`
`3 42`
`7`
	`10`
	`-1`
	`3`
	`42`

`42 43 44 45 46 47`
`48 49 50 51 52 53`
`54 55 56 57 58 59`
`60`
`42`
	`42`

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

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

int main(void)
{
    int value;
    do {
        scanf("%d", &value);
        printf("%d\n", value);
    } while (value != 42);
}

```

	Input	Expected	Got	

✔	`10 -1`
	`3 42`
	`7`
		`10`
		`-1`
		`3`
		`42`
			`10`
			`-1`
			`3`
			`42`
				✔

✔	`42 43 44 45 46 47`
	`48 49 50 51 52 53`
	`54 55 56 57 58 59`
	`60`
	`42`
		`42`	`42`	✔

✔	`-42`
	`+42`
	`42`
	`3`
		`-42`
		`42`
			`-42`
			`42`
				✔

✔	`3 4 5`
	`41 42 43`
		`3`
		`4`
		`5`
		`41`
		`42`
			`3`
			`4`
			`5`
			`41`
			`42`
				✔

Passed all tests! ✔

Correct

Marks for this submission: 1.00/1.00.

Information

for loops

The most commonly used loop in C is the `for` loop. However, this is different from Python's `for` loop, which iterates over the elements of a collection. C's `for` loop is instead a generalisation of a `while` loop, that allows you to specify the loop initialisation, the loop condition, the loop body and the loop increment code as separate syntactic elements.

Syntax:

```
for ( Initialisation ; Condition ; Update ) LoopBody
```

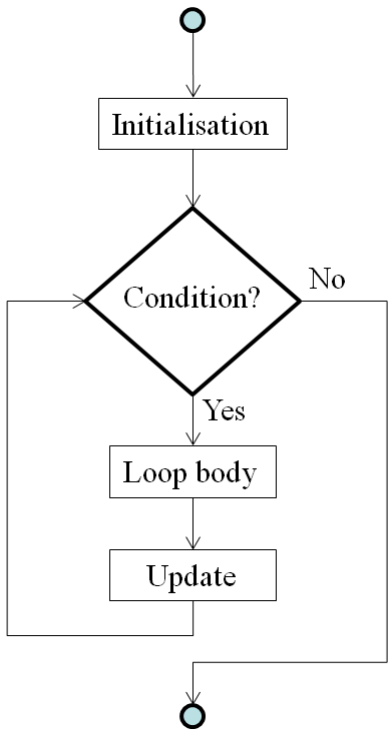
For example:

```
/* Print all the integers between 1 and a value n (read from standard
 * input) that are factors of n.
 * Uses a for loop this time.
 */
#include <stdio.h>
int main(void)
{
    int n = 0;

    scanf("%d", &n);
    for (int i = 1; i <= n; i++) {
        if (n % i == 0) {
            printf("%d\n", i);
        }
    }
}
```

The *for* loop in the above example makes use of the fact that in C99 (but not in C89) the for loop initialisation can declare and initialise a new temporary loop control variable (*i* in this case).

The flow chart for a *for* loop is:



ENCE260 style rule: use *for* loops only as "counting" loops, i.e., where there is a single loop control variable that is counting up or down, e.g.

```
for (int i = 0; i < n; i++) { ... do something with i ...}
```

Don't use them in more-general "while loop" contexts, as they rapidly become cryptic and unreadable. Also, the fact that the loop update expression is executed after the loop body is liable to confuse the reader in more complex loops.

Question **10**

Correct

Mark 1.00 out of 1.00

What number is printed by the following program? [You can check your answer by computer, of course, but please try to work out first what the answer will be.]

```
#include <stdio.h>
#include <stdbool.h>  // Define the type bool, literals true and false
int main(void)
{
    int i = 0;
    bool done = false;
    for (i = 0; !done; i++) {
        if (i == 10) {
            done = true;
        }
    }
    printf("%d\n", i);
}
```

Note: this example is intended to show you how **not** to use a *for* loop. Please do not copy this code; instead you should only ever use the simple counted loop version i.e., a loop of the form

```
for (int i = 0; i < n; i++) { ...
```

Answer: 11 ✔

Correct

Marks for this submission: 1.00/1.00.

Question **11**

Correct

Mark 1.00 out of 1.00

Write a program that reads a single integer *n* from standard input (without a prompt) and then prints to standard output the squares of the integers from 1 up to *n* (inclusive), one per line. Use a *for* loop. You may not use *while*, *do* or *break*.

For example:

Input	Result
4	1 4 9 16
1	1

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

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


int main(void)
{
    int i = 0;
    int j = 1;
    int store;
    scanf("%d",  &i);
    for(j = 1; j <= i; j++) {
        store = (j * j);
        printf("%d\n", store);
    }
}

```

	Input	Expected	Got	
✔	4	1 4 9 16	1 4 9 16	✔
✔	1	1	1	✔
✔	0			✔

Passed all tests! ✔

Correct

Marks for this submission: 1.00/1.00.

Question **12**

Correct

Mark 1.00 out of 1.00

If you've done COSC121 you may remember Herbert the Heffalump, who climbs scree slopes in a series of rushes. [See the definition of "scree" below.] Each rush gains him a certain amount of height, *rushHeight*, but then, unless he is already at the top, he slips back down again by an amount *slideBack*. Write a program to help Herbert determine how many rushes it will take him to get to the top of a scree slope of a given height.

The program should read 3 float values *screeHeight*, *rushHeight* and *slideBack* from standard input, in that order (without issuing any prompts), and print to standard output the number of rushes it takes Herbert to get to the top. You may assume that *screeHeight* and *slideBack* are both non-negative and that *rushHeight* is greater than *slideBack*.

Consider whether any of the built in loop constructs (*while*, *do-while* or *for*) are really a good fit to this problem.

Once you've got the program working with one of the loop constructs, try implementing it using the other two. [Once you have full marks you can experiment without penalty to your heart's content.]

Note: "scree" is loose gravel that tends to slide backwards underfoot as you go up a hill.

For example:

Input	Result
`100 50 10`	`3`
`100 100 90`	`1`
`0 100 30`	`0`

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

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


int main(void)
{
    float screeHeight;
    float rushHeight;
    float slideBack;
    int counter = 0;
    float height = 0;
    bool reached;
    reached = false;
    scanf("%f", &screeHeight);
    scanf("%f", &rushHeight);
    scanf("%f", &slideBack);
    while ((reached == false)) {

        height += rushHeight;
        counter += 1;

        if (height >= screeHeight) {
            reached = true;
        }
        if (screeHeight < rushHeight) {
            counter -= 1;

        }

        else {
            height -= slideBack;
        }

    }
    printf("%d", counter);
}

```

	Input	Expected	Got	
✔	`100 50 10`	`3`	`3`	✔
✔	`100 100 90`	`1`	`1`	✔
✔	`1.0 0.5 0.1`	`3`	`3`	✔

	Input	Expected	Got	
✔	`100 40 10`	`3`	`3`	✔
✔	`1000 20 19.7`	`3268`	`3268`	✔
✔	`0 100 30`	`0`	`0`	✔

Passed all tests! ✔

Correct

Marks for this submission: 1.00/1.00.

Question **13**

Correct

Mark 1.00 out of 1.00

Write a C program that uses *scanf* to read two integers *n1* and *n2*, where *n1 > 1* and *n2 >= n1*. The program should then print all the prime numbers between *n1* and *n2* inclusive.

A simplistic algorithm with nested loops is all that's expected here. The outer loop iterates through all integers in the range *n1* to *n2* inclusive. The inner loop iterates through all the integers from 2 to *n* - 1 inclusive, checking if each one is a factor of the current number *n* whose 'primeness' is being assessed. More efficient algorithms exist, however, and you're welcome to try to find one if you wish.

For example:

Input	Result
2 40	2 3 5 7 11 13 17 19 23 29 31 37

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

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

int main(void)
{
    int n1 = 0;
    int n2 = 0;
    scanf("%d", &n1);
    scanf("%d", &n2);
    for (int i = n1 ; (i <= n2); i++) {
        bool prime = true;
        for (int j = 2; (j < (i - 1)); j++) {
            if ((i % j) == 0) {
                prime = false;
            }
        }
        if (prime == true) {
            printf("%d\n", i);

        }
    }
}

```

	Input	Expected	Got	
✔	2 40	2 3 5 7 11 13 17 19 23 29 31 37	2 3 5 7 11 13 17 19 23 29 31 37	✔
✔	60 65	61	61	✔

	Input	Expected	Got	
✔	`100 100`			✔
✔	`100 101`	`101`	`101`	✔
✔	`79 79`	`79`	`79`	✔

Passed all tests! ✔

Correct

Marks for this submission: 1.00/1.00.

◄ Quiz 1: Linux and C Basics

Jump to...

Quiz3: Arrays and Functions ►