UESTC 1005 – Introductory Programming

LAB MANUAL

SEMESTER 1

Prelab Assignments

1. Before Lab Sessions

1.1. Configure Programming Environment

Most of the professional programmers rely on integrated development environment (IDE) tools to write, compile, and test their software. However, these tools can be too intimidating for a beginner. Therefore, in this course, we will rely on simple programming environments through which we can easily write, debug and run our C programs. For this, you can use Visual Studio Code, Code::Blocks or CodeLite. Although Code::Blocks and CodeLite are able to perform most of the underlying processes required to run a program such as compiling, linking, and building, it is still important to be aware of these processes. For this we will use command line (CL) tools such as GCC which is an open-source CL compiler, that normally don't come shipped with operating systems such as Windows 10. To do so, please install Cygwin which is a tool that lets us use Linux based apps on Windows. We will primarily use it for GCC compiler.

1.2. Running your First C Program

Please follow the instructions below:

- In Code::Blocks, go to File and create a New Project.
- Click Console Application and press Next. Now, keep the remaining settings to their default. Select C as the programming language.
- Give a name, First One to the project name.
- Make sure you have provided a correct path to where the project will be stored in the hard disk space.
- In the Project Management Windowpane, under the Workspace, you will find your project. Double click it and you will source further lists beneath the project title. Go to source, where you will find `main.c`

file. This is where you will write your C programs. Double click it and you will see the contents of the C program.

- To run the code, in the top Menu Bar, go to the Build menu, and press Build and Run. A new command line screen should pop up in which the output of the C program will be displayed.

1.3. EXERCISE - The Hello World!

By convention, the first thing to learn in a new programming language is to print the text “Hello, world!”. Although text isn’t very exciting by itself, the ability to output text is vital for debugging (fixing programs).

Most programming languages, including C, indicate text with “double quotation marks”: the first “indicates the beginning of text. The computer then treats everything until the next“ as text.

This causes problems if you want to print an actual “symbol in the output. For that reason, we use escape sequences to be symbols which are difficult (or impossible) to indicate with plain text.

In addition to telling a compiler how to create an executable file, source code should also tell future programmers why the code does what it does. If anything in the code is unclear, a programmer should add comments which explain the situation. More information about comments is on the good programming style page.

1.4. EXERCISE – Student Population

Write a short program that displays the number of students in the different cohorts as a list:

EEE	240
CE	220
IE	100

Table 1: Student Population List

- Print a message similar to the above example.

- No line should be longer than 80 characters you will need to use multiple `printf()` statements.
- Use both types of comments (single line and multi-line).
- Use all these escape sequences:
  - `\n`
  - `\`
  - `\\`
  - `\’`
  - `\”`

```
// Name: Hello_world.c
// Purpose: Prints Hello, World! on screen
// Put your name here

// Single Line Comment

/* this type of comment can span
  multiple lines; everything is a
  comment until the computer sees the
  ending. */

#include <stdio.h>

int main() { printf("Hello,
     world!");

     // wait for a keypress
     getchar();

     return 0;
}
```

Lab Session 1

2. Variables

So far, the programs that we have seen consist of simple statements. However, most of them are complex, and therefore, we need a way to store data temporarily while the programs are running. In **C**, we use *variables* to do so, which are basically storage locations in the memory where the data can be stored. Variables can be of different types which will decide what type of data we can actually store in the memory. In this lab session, we will discuss two of them, int and float. We have to be careful in selecting the right type of variable as it can affect the execution of our programs. As an example, a variable of type int can only store natural numbers, i.e., numbers without any decimal points. On the other hand, float type variables can store decimal point numbers that can be much larger than what can be store in int type variables. For example, float variables may look like 234234.2342342342 or 0.00001, whereas int variables can be -263856, or 123.

For your information, there are other types of variables such as char and double, which can be used to store *character strings*, such as your name, and high-precision numbers respectively. But more on that later.

2.1. Variable Data type

As we mentioned earlier, variables in **C** can be primarily categorised in three general categories: numeric, character and user-defined. So far, we have looked at the first two. We'll take a look at the last one later in the course. The characteristic that distinguishes these data types is the size of the memory allocation. Various programs may required high precision along with a wide range of possible values that can be assigned to the variable. Table 2 provides an idea of the size in bytes allowed by each data type. The range of the unsigned is nearly double than the unsigned counterpart.

	Signed	Unsigned
short	`short int a; short b;`	`unsigned short int a; unsigned short b;`
default	`int a;`	`unsigned int a;`

long	long a;	unsigned long a;
char	char a; signed char a;	unsigned char a;

Table 2: Sizes of various data types in C.

To obtain the size and range of the respective variable data types, take a look at the following program.

```
// Program Name:
// Purpose: Prints a Floating-point number // Your name:
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
#include <float.h>
int main(int argc, char** argv) {
        printf("Storage size for int in bytes: %d \n", sizeof(int));
        printf("Storage size for float in bytes: %d \n",
        sizeof(float));
        printf("Storage size for long in bytes: %d \n",
        sizeof(long));
        printf("Storage size for short in bytes: %d \n",
        sizeof(short));
        printf("Storage size for double in bytes: %d \n",
        sizeof(double));
        printf("Storage size for char in bytes: %d \n \n \n",
        sizeof(char));
        printf("Maximum value of float: %g\n", (float) FLT_MAX);
        printf("Minimum value of float: %g\n", (float) FLT_MIN);
        printf("Maximum value of int: %d\n", INT_MAX);
        printf("Minimum value of int: %d\n", INT_MIN);
        printf("Maximum value of long: %ld\n", (long) LONG_MAX);
        printf("Minimum value of long: %ld\n", (long) LONG_MIN);
        printf("Maximum value of short: %d\n", SHRT_MAX);
        printf("Minimum value of short: %d\n", SHRT_MIN);
        printf("Minimum value of double: %g\n", (double) DBL_MAX);
        printf("Minimum value of double: %g\n", (double) DBL_MIN);
        printf("Maximum value of unsigned char: %d\n", UCHAR_MAX);
        printf("Maximum value of unsigned int: %u\n", (unsigned int)
        UINT_MAX);
        printf("Maximum value of unsigned long: %lu\n", (unsigned
        long) ULONG_MAX);
        printf("Maximum value of unsigned short: %d\n", (unsigned
        short) USHRT_MAX);
        return 0;
}
```

In the program above, we use the libraries `float` and limits that contain the definitions of various variable types as constants. Write the above program and show the output to the GTA.

Now as we mentioned in the class, to use variables we have to *declare* them first. By that, we are telling the compiler the type of variable before it can be used. We also need to follow some certain rules regarding the name of the variable. As an example,

```
int marks;

float gpa;
```

After declaring the variable, we provide a value to a variable and this step is called *assignment*. For the variables declared above, we can assign them values using:

```
// Program Name:
// Purpose: Prints a Floating-point number
// Your name:
#include <stdio.h>
int main()
{
        marks = 23;
        int marks;
        printf("My marks in the IP are %d", marks);
        getchar();
        return 0;
}
```

Figure 1: C Program to print a floating point number.

```
marks = 88;
gpa = 3.43;
```

where the numbers on the right are called *constants*.

Keep in mind the variables need to be declared first, and only then they can be assigned a value. As an exercise, try the code below and note down the result:

As a best practice, whenever we assign a value to a float variable, we append the *constant* with the character f in the end, such as `gpa = 3.4f;`.

2.2. Printing a variable

We can use the function `printf()` to display the value of a variable on screen. As an example, the statement:

```
printf("The value of the variable is %d \n", marks);
```

will do the job. `%d` is a placeholder that tells the compiler where the int variable value should be displayed. For other variables, we use other placeholders such as `%f` for float variables. For example, to print a float with a value 0.0001, we can use:

```
// Program Name:
// Purpose: Prints a Floating-point number
// Your name:
#include <stdio.h>
int main()
{
        float some_number;
        some_number = 0.0001;
        printf("The float variable value is %f",
        some_number);
        // wait for keypress
        getchar();
        return 0;
}
```

Try using other placeholders like %3f, %.3f and %6f to see what happens.

2.3. Programming Tasks and Exercises

2.3.1. Computing the Dimensions of a Box

You are required to write a program, which has three variables with values, height, length and width having values 4, 5, and 6 respectively.

Then you are required to calculate the volume of the box, which is the product of all three variables. For multiplication, we use *.

Also, consider the total weight of the box is just the volume divided by 1.5. For division, we use /.

You are required to print the variables in the following fashion:

```
The height of the box:
The length of the box:
The width of the box:

The total volume:
```

```
Total Weight:
```

2.4. Value ranges of variables

Write down what you see as after you try to compile the program of Fig. 1. Indicate the errors and fix them. Write down the corrected lines.

2.4.1. Mathematical Expressions

The exponential function, exp(*x*) is a commonly used function defined by a power series:

$$\exp(x) = 1 + x + \frac{x^2}{2} + \frac{x^3}{6} + \frac{x^4}{24} + \cdots \quad (1)$$

In this exercise, you are required to write a program that computes the exponential function of a variable *x*. *Instructions*

- Remember to represent the power as a product, x^3 = `x * x * x`.
- Remember `x` is an `int`.
- Set the value of *x* equal to 1.
- Create another variable named `exp_x` that stores the series in Eq. 1.

- Print the value of `exp_x`.

You can use the below expression for the power series representation of the exponential function.

```
exp_x = 1+x+(float)x*x/2+(float)x*x*x/6+(float)x*x*x*x/24;
```

2.5. Variable Types

Type casting in **C** is a way through which we can convert a variable from one data type to another. Write a program that declares two int variables with values 6 and 7. Now declare another variable of type float and assign it with a value which is the ratio of the first two variables. In short, you need to write a correct program that does the following:

$$c = \frac{a}{b}. \tag{2}$$

Print the values of all three variables in separate lines.

3. Conditions

The logic of an `if` statement is fairly simple. If the statement is true, the *code block* (indicated by a statement or the contents of a `{...}` block) is run. A closely related command is the `if ... else`, which acts precisely as you would expect.

Here are two cautions to avoid common problems.

1. In C, x = 1 and x == 1 do completely different things:
   - The single = is an *assignment* operator: it changes to the value of x to be 1.
   - The double == is the *comparison* operator: it returns true if x is equal to 1.
2. If you forget to use the `{ ... }` code block, then it will be very easy to make a mistake later on. The computer will happily execute statements which you did not think would be running. This is illustrated at the bottom of the "conditional" example.

3.1. Technical details

```
#include <stdio.h>

int main() {
     int x = 3;

     if (x == 4) {
         // this line will not be printed
         printf("x is equal to 4.\n");
     }
     // one of these two lines will be printed
     if ((x > 10) && (1 != 0)) {
         printf("Expression is true.\n");
         printf("x is greater than 10.\n");
     } else {
         printf("Expression is false.\n");
         printf("x is not greater than 10.\n");
     }

     // something weird happens here!
     if (x > 10)
         printf("Expression is true.\n");
         printf("x is greater than 10.\n");

     getchar();
}
```

The logical operators (not !) (and &&) (or ||) are particularly useful with if. Occasionally the exclusive or ^ is useful. Parentheses () are highly encouraged when using logical operators.

Table 3: Income Tax Bands and Rates in Scotland

Band	**Taxable salary**	**Scottish tax rate**
Personal Allowance	Up to £12,500	0%
Starter rate	£12,501 - £14,585	19%

Band	Taxable salary	Scottish tax rate
Basic rate	£14,586 - £25,158	20%
Intermediate rate	£25,159 - £43,430	21%
Higher rate	£43,431 - £150,000	41%
Top rate	over £150,000	46%

3.2. Exercise

In Scotland, the tax is calculated based on the information given in Table 1. Write a program, that will input the gross salary(taxable salary) from the user, and based on the tax rate, calculates the take-home salary (net salary). For example, if one earns £40,000 a year, they pay:

nothing on the first £12,500

19% (£491.15) on the next £2,585

20% (£2,114.60) on the next £10,573

21% (£3,011.12) on the next £14,342

--

Total Tax: £5,617.57.

As another example, if one earns £45,000 a year, they pay:

nothing on the first £12,500

19% (£491.15) on the next £2,585

20% (£2,114.60) on the next £10,573

21% (£3,837.12) on the next £18,272

41% (£643.70) on the next £1,570.

--

Total Tax: £7,086.57

An example input/output dialog is shown below:

```
Enter Gross Income in £: 40000
Your Take-home salary is: £34382.43
The total tax paid is: £5617.57
```

4. Functions

You are familiar with functions in mathematics -- you've handled functions like g(x) = x^2 + 2x - 3. In this case, the function *returns* a value. The returned value is computed based on the input. You have also seen functions with multiple inputs: h(x,y) = 5x - 6x + x*y.

We use functions like this in programming. In C, we need to specify the type of variable being returned -- `int` or `float`. We also need to specify the type of input variables.

We also sometimes use functions which do not return any values -- these are useful if we want to repeat a few commands, or simply for program organization. These functions are called **`void`** functions.

Functions must be declared before ("higher in the .c file") they are used.

Note that each function has its own *scope* of variables. A function cannot refer to a variable which was declared in the `int main()` function. In fact, you can re-use the same variable name in different functions!

This is the time we have seen variable scope, but we can illustrate the problem even without functions. Consider the following example -- it will not compile. The `int x` is defined inside an extra code block { . . . }. The scope of variable `x` only lasts for that code block; referring to this variable outside of that scope will produce a compiler error.

```
int main() {
     {
          int x = 0;
     }
     x = 1;
}
```

This is one reason why indentation is useful in source code.

4.1. Technical details

Simple function:

```
#include <stdio.h>

float g(float x) {
     return x*x + 2*x - 3;
}

int main() {
     float a = 3.2;
     float b = g(a);
     printf("g(%f) = %f\n", a, b);

     printf("g(5) = %f\n", g(5));


     // wait for a keypress
     getchar();
}
```

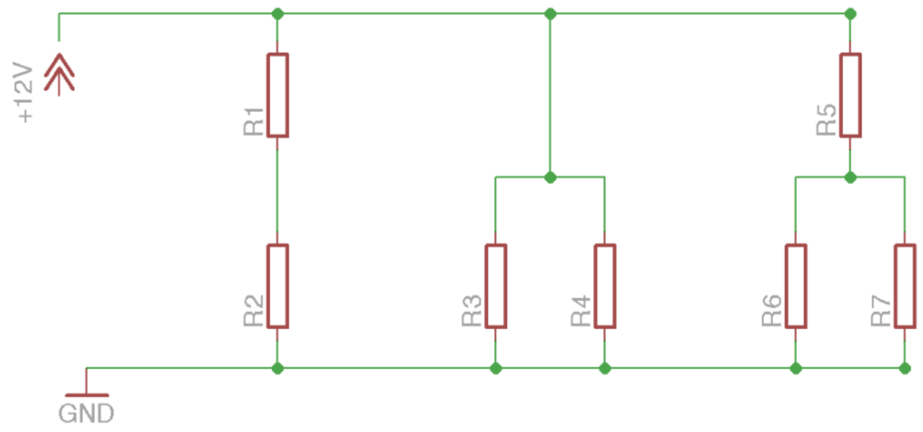
Note the different variables names in the above example!

Various functions:

```
#include <stdio.h>

int catYears(int human_years) {
     float y = human_years / 7.0;
     // this would be a good place for a comment
     // why did I write the next line?
     int rounded = y+0.5;
     return rounded;
}

float sumOfSquares(float a, float b) {
	return a*a + b*b;
}

void p(int years, float c, char animal[100]) {
	printf("If I were a %s, I'd be ", animal);
	printf("%i years old.\n\n", years);
	printf("The sum of squares ");
	printf("was %f units.\n", c);
}

int main() {
	int my_age = 18;
	float x = 3.3;
	char animal[100] = "cat";

	float years_a = catYears(my_age);
	float sum = sumOfSquares(x,7.4);

	p(years_a, sum, animal);

	// wait for a keypress
	getchar();
}
```

4.2. Exercise

As electrical engineering students, you are familiar with Ohm's Law (*V* = *IR*) and calculating the value of resistors in series (*R* = *R1*+*R2*) and parallel (*1/R* = *1/R1* + *1/R2*). Consider the following circuit:



Write a program to calculate the overall current in this circuit.

- You *must* use 3 functions: `Ohm_law(...)`, `series(...)`, and `parallel(...)`. A 4th function `parallel_three(...)` is optional. Do not write functions like `sum_r1_r2(...)`.
- Calculate the current for:

  ```
  int  r1=100,  r2=200,  r3=300,  r4=400,  r5=500,
  r6=600, r7=700;
  ```

  (you may copy and paste this into your .c file)

  The answer should be 124.6 mA.
- Calculate the current for:

  ```
  int  r1=123,  r2=234,  r3=345,  r4=456,  r5=567,
  r6=678, r7=789;
  ```

  The answer should be 107.6 mA.
- You may find it helpful to use intermediate variables, and to print debugging information.

(optional: you can call functions as inputs to other functions, i.e.

```
z = f( g(x,y), h(x,y) );
```

Using this, calculate the current from voltage and Rx values in a single line.)

Lab Session 2

5. While Loop

The simplest loops in C are `while` loops. These loops are very similar to `if` statements. The difference is that there is not `else` statement included in a `while` loop, and the `{ ... }` code block will be repeated as long as the statement is true.

There is no built-in `true` statement in C, so we use `1` for true and `0` for false.

To stop an infinite loop, use `ctrl-c`

A closely related form is the `do ... while` loop. Examine the code presented -- is there any difference in the output? Try setting `int i=10`

5.1. Technical details

```
 #include <stdio.h>
int main()
{
     while (1) {
         printf("Infinite loop is infinite.\n");
     }
    getchar();
}
```

```
 #include <stdio.h>

int main()
{
     int i = 0;
     while (i < 10) {
         printf("The loop has repeated ");
         printf("%i times.\n", i);
         i++;
     }
     getchar();
}
```

```
#include <stdio.h>
int main()
{
     int i = 0;
     do {
         printf("The loop has repeated ");
         printf("%i times.\n", i);
         i++;
     } while (i < 10);
     getchar();
}
```

5.2. Exercise

Save your guessing game from the previous exercise ("if conditionals") with a new file name; we will be modifying it, but you don't want to lose all your hard work.

Modify your guessing game:

- Instead of giving the user X chances, keep on offering guesses until the user is correct.
- Keep track of the number of guesses, and tell the player how many guesses it took.
- After the first game is finished, start a new game with a different answer to guess. Keep track of the number of guesses the player needed to win.
- After the second (and later) games are finished, tell the player their score (number of guesses) and the best score (the fewest number of guesses). If the player's score is better, store it as the best score.
- Keep on repeating until the player wins with 4 or fewer guesses. (or until the player hits `ctrl-c`)

(optional: randomly change the rules -- instead of always going from 1 to 32, let the computer randomly decide to do 100 to 131, -97 to -66, etc. If you change the size of the range, you can't compare high scores for different games. However, you could program the computer to randomly select "all numbers divisible by 4 between 32 and 156" and the like.)

6. For Loop

A loop that counts up (or down) is the most common type of loop, so most programming languages have a short-cut: the `for` loop.

This loop squishes the initial setting, conditional "do we continue", and ending "how to change the variable" into a single line. Given a statement of the form:

```
for (init; run_loop?; after_loop) {
    ...
}
```

- *init*: This is executed **once**, at the beginning of the entire loop. Generally, will be something like `i=0`.
- *run_loop?*: The computer tests this **before each** loop. Generally, will be something like `i<10`.
- *after_loop*: the computer executes this statement **after each** loop. Generally, will be something like `i++`.

6.1. Technical details

The most common form of loop is this:

```
#include <stdio.h>
int main()
{
     int i;
     for (i=0; i<10; i++) {
         printf("The loop has repeated ");
         printf("%i times.\n", i);
     }
     getchar();
     return 0;
}
```

Here's a weird loop:

```
#include <stdio.h>
int main()
{
     int i;
     for (i=2000; i>0; i=i/2-3) {
         printf("What is this loop ");
         printf("doing?  i is %i\n",i);
     }
     getchar();
     return 0;
}
```

6.2. Exercise

Back in the good old days, there were no fancy graphics in computer games; we used text to represent everything. Your task is to draw a room from a family of games called *Roguelikes* -- the player (represented by the `@` symbol) must explore a dungeon.

2x2 room, player at 0,1:

```
+--+
|..|
|@.|
+--+
```

5x3 room, player at 1,2:

```
+-----+
|.....|
|.....|
|.@...|
+-----+
```

14x8 room, player at 8,5:

```
+--------------+
|..............|
|..............|
|..............|
|..............|
|..............|
|........@.....|
|..............|
|..............|
+--------------+
```

Your task is to write a function (with extra "helper" functions) to draw such a room.

- How big are the rooms? What is the coordinate system?
- Your `int main()` must contain only:

```
int main() {
      drawRoom(2,2,0,1);
      drawRoom(5,3,1,2);
      drawRoom(14,8,8,5);

      getchar();
      return 0;
 }
```

- In addition to `drawRoom()`, write *three extra* functions. The first draw a horizontal `+----+` line, the second draws a line without the player `|....|`, and the third draws the line with the player `|..@.|`. The `drawRoom` function should call those other helper functions when appropriate.
  (examples of the three extra functions are not drawn to scale)
- Use `for` loops. You may *not* use `if` or `while` in this exercise.

(optional: combine this exercise with keyboard input -- let the player move around in the room, bump into walls, etc. Ask the user to turn on the *numlock* key and to press `enter` after every move, then you can read his moves by reading `int` from the keyboard. Use a while loop for this movement.).

Lab Session 3

7. Arrays

An *array* is an ordered sequence of values; you cannot rearrange values without changing the meaning. A two-dimensional array is just a normal "table".

In C, arrays are indexed (accessed) starting from 0. For example,

```
int array[4] = {3, -2, 987, 12};
```

creates an array in memory, storing each value at the appropriate index:

Index	0	1	2	3
Value	3	-2	987	12

The array's size can only be specified when you create it. If you want to store more information in that array, you need to create a new (bigger) array, then copy information from the old array to the new array.

Due to the way C handles arrays and pointers (coming later), you can modify an array inside a function without needing to return anything. The function needs to know the array's size, though!

7.1. Technical details

Initializing and displaying:

```
#include <stdio.h>

int main() {
    int array[8] = {1, 2, 8, 3, -5, -1, 1};
    int i;
    // display the array
    for (i=0; i<8; i++) {
        printf("%i ", array[i]);
    }
    printf("\n");

    // change the array
    for (i=0; i<8; i++) {
        array[i] = 2 * array[i] - 1;
    }

    // display the array again
    for (i=0; i<8; i++) {
        printf("%i ", array[i]);
    }

    printf("\n");
    getchar();
   return 0;
}
```

Arrays and functions:

```
#include <stdio.h>

void printArray(int arr[], int size) {
    int i;
    for (i=0; i<size; i++) {
        printf("%i ", arr[i]);
    }
    printf("\n");
}

void changeArray(int arr[], int size) {
    int i;
    for (i=0; i<size; i++) {
        arr[i] = arr[i] * arr[i] + 3;
    }
}

int main() {
    int array[8] = {1, 2, 8, 3, -5, -1, 1};
    printArray(array, 8);
    changeArray(array, 8);
    printArray(array, 8);

    getchar();
    return 0;
}
```

7.2. Exercise

The Fibonacci numbers are a famous sequence of numbers. They begin with 0 and 1, and then the next value in the sequence is the sum of the previous two values.

0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765

(`fib[8]` is 21 -- remember to start counting from 0!)

Write a program that calculates the Fibonacci sequence.

- Make a function that accepts n, which is the number of integers to generate.
- Declare an array, initialize it with only the first two Fibonacci numbers, then calculate the rest.
- Display the sequence for n=10 and n=20.
- Try generating output for n=50. If anything goes wrong in this step, you don't need to fix it. Just add a comment explaining what happens, and why.

8. Two-Dimensional Array

Just like a one-dimensional array is an ordered sequence of values, a two-dimensional array is an ordered sequence of one-dimensional arrays. By convention, we think of these 1-D arrays as being stacked vertically.

Array Number	Array
0	1-D array | 0 | 1 | 2 | 3 | 4
1	1-D array | 3 | 7 | 2 | 1 | 4
2	1-D array | 4 | 2 | 8 | 5 | 6

8.1. Technical details

Using 2-D arrays:

```
#include <stdio.h>

// we MUST specify the size for a 2-D array
// (unless you use pointers)
void printArray(int values[5][3]) {
    int i, j;
    for (i=0; i<5; i++) {
        for (j=0; j<3; j++) {
            printf("%i ", values[i][j]);
        }
        printf("\n");
    }
}

int main() {
    int array[5][3];
    int i, j;

    // fill array with some values
    for (i=0; i<5; i++) {
        for (j=0; j<3; j++) {
            array[i][j] = 3*i + j;
        }
    }
    printArray(array);

    getchar();
    return 0;
}
```

C does not check that you are only accessing an array within the bounds of the array. If you ask for `array[-1][-1]` then the computer will tell you what was at that memory location, even though that memory does not belong to that variable!

8.2. Exercise

Write a program which creates a "game board". We will use this in the next exercise to write a tic-tac-toe game (also known as "noughts and crosses", "tick tack toe", "X's and O's"). If you're not familiar with the game, see Wikipedia's page on tic-tac-toe, or ask a lab instructor.

- Your game board should be 3x3. Each square in the board can either be empty (print a dot "."), or have an "X":

```
. X .
X . .
. . .
```

- You *must* use a two-dimensional array to represent the game board in memory.
  (You can't have variables like
  `int squareUpperLeft, squareBottomMiddle; ...`)
- The board begins complete clear (all blank squares `"."`)
- The player may select any square by entering a number from the *computer keypad*. (your program should read it as an `int`, and you may assume that it is between 1 and 9 inclusive)
- Before each player takes a move, print out the current game board. You can remove board from the screen by doing

`printf("\n\n\n\n\n\n\n\n\n\n");` 2 or 3 times.

- If the player tries to select a square that is currently occupied, print a warning message and ask them to choose again.
- Continue playing the game until all squares are filled.
- You *must* create formulae to deal with "number <=> row and column". You *may not* use `if` statements or a `switch...case` statement for these formulae, but you can use them elsewhere in the assignment.
- Hint: create formulae to deal with "number <=> row and column" for these three cases:

```
(a)          (b)           (c)
easy         mobile        computer keypad
0 1 2        1 2 3         7 8 9
< row 0
3 4 5        4 5 6         4 5 6
< row 1
6 7 8        7 8 9         1 2 3
< row 2

^ ^ ^        ^ ^ ^         ^ ^ ^
0 1 2        0 1 2         0 1 2
column       column        column
```

  Your assignment needs to work with the *computer keypad*, but it is strongly recommended that you learn how to solve (a) and (b) before trying to do (c).

  - You may find the division `/` and modulus `%` (or "remainder") operators useful.
  - Solve case (a) first.
  - After solving (a), figure out how to transform the numbers in (b) into the numbers in (a).
  - Do the same for (c) into (b).

8.3. Bonus Exercise Tic-Tac-Toe

Write a tic-tac-toe game (also known as "noughts and crosses", "tick tack toe", "X's and O's"). If you're not familiar with the game, see Wikipedia's page on tic-tac-toe, or ask a lab instructor.

- Use your "game board" from exercise 2. The same constraints apply, namely:
  - You *must* use a two-dimensional array to represent the game board in memory.
  - You *must* create formulae to deal with "number <=> row and column". You *may not* use `if` statements or a `switch...case` statement. You can use those commands elsewhere in the assignment.
- Modify your "print the game board" function to print out `.XO` as required.
- Your game should be human-vs-computer, and human moves first.
- Computer moves randomly. Note that it must select an unoccupied square; you cannot simply pick a number from 1-9 at random without checking!
- Before each player takes a move, print out the current game board. You can remove board from the screen by doing `printf("\n\n\n\n\n\n\n\n\n");` 2 or 3 times.
- After each player takes a move, you must check to see if anybody won the game.

  (hint: write a function that checks if player `X` won the game, then call this function twice, for player 1 and player 2) (another hint: there are 8 possible winning combinations; a player wins if one of 8 combinations of 3 square all have his mark. You can either write out all 8 combinations in full, or use three `for` loops -- test the three vertical wins, three horizontal wins, and two diagonal wins.) (final hint: begin work on this point by changing the rules of the game. Pretend that you can only win by going horizontally or diagonally across the middle. Solve this problem first; once you have it working for the "fake rules", just add in the other 6 winning combinations.)

(optional: instead of having the computer move randomly, try to make it play intelligently.).

Lab Session 4

9. Pointers

Pointers: stuff from lecture notes.

To keep track of time, computers need something to measure it against. In C on the lab computers, we get the number of seconds since 00:00:00 UTC on 1 January 1970.

As you might imagine, this produces a fairly large number -- it exceeds the bounds of a normal `int`. We therefore use a `long int`, which has twice the number of bits of a normal `int`. Our `time.h` library also defines a special data type to store the time value: `time_t`. On our computers, this is the same as a `long int`.

9.1. Technical details

Swapping data with pointers:

```c
#include <stdio.h>

void swap(int *px, int *py) {
    int temp = *px;
    *px = *py;
    *py = temp;
}

int main() {
    int x = 1;
    int y = 2;
    printf("initial: %i %i\n", x, y);
    swap( &x, &y);
    printf("swapped: %i %i\n", x, y);

    getchar();
    return 0;
}
```

Watch out for the * and &

Getting the time:

```
#include <stdio.h>
#include <time.h>  // extra include

int main() {
    time_t now;
    now = time(NULL);

    long int a = now;
    printf("%li\n", now); }

    getchar();
    return 0;
}
```

9.2. Exercise

Calling `time(NULL)` gives you a large integer. Use this value to calculate today's year, date, hours, and minutes.

- Simplifying assumption: we pretend that each year has exactly 365 days. The 15th of March is therefore day 83 in year 2011. (use that date to figure out today's "day number") (hint: how many seconds are there in a day?)
- You *cannot* use `struct` for this assignment.
- You *cannot* use any time-based functions from the standard library such `asasctime` or `strftime`.
- You *must* use one function to calculate the year and day (pass in the total seconds, and a reference to the year and day).
- You *must* use a separate function to calculate the hours and minutes. (again, pass the total seconds, and a reference to the hours and minutes)

- You *must* use a separate function to print the time; this function can only take as arguments the year, day, hour, and minutes.

(optional: handle leap-years correctly, and print the month as well).

10. Bit manipulation

As engineers, you will (or *should* at least!) have more interest in bitwise manipulation than most other programmers. Writing software that controls lights, reacts to flipped switches, and generally anything that interacts closely with hardware will require bitwise operations. The main concepts here are *flags* and *bitmasks* (commonly known as "masks).

A *flag* is a particular bit with a special meaning. For example, when representing negative numbers, the 8th bit could be thought of a flag which indicated whether the remainder of the bits should be interpreted as a positive or negative number.

A *bitmask* is a way of accessing a particular bit (generally, but not always, a flag). The bitmask is a number which has 0 in all bits we don't care about, and a 1 for the bit(s) that we want to examine. By *and*ing the bitmask with the original number, we can "extract" the bit(s) -- if that bit was 0, then the new number will be completely zero; if the bit was 1, then the new number will be non-zero.

The same operation is done in reverse to set a flag -- by *or*ing a bitmask and the data variable, we can set a flag to be true. Setting a flag can be done by *and*ing the variable with the *not*ed bitmask.

10.1. Technical details

Various bitwise operations:

```
#include <stdio.h>

int main() {
    char x = 0x10; // number in hex
    char y = 63;

    printf("char is a one-byte number.  ");
    printf("Printing it with %%c will treat ");
    printf("it as\nan ASCII value, but we ");
    printf("can also print it as a number: ");
    printf("x=%i y=%i.\n", x, y);

    printf("Some operations: ");
    printf("%i\t%i\n", ~x, ~y);
    printf("%i\t%i\t%i\n", x&y, x|y, x^y);

    getchar();
    return 0;
}
```

Toggling individual bits:

```
#include <stdio.h>
#include <stdlib.h>  // extra includes!
#include <time.h>

float getRand() {
    return rand() / (RAND_MAX+1.0);
}

int main() {
    srand( time(NULL) ); // init random
    char x = 0;
    char bit, bitmask;
    int i;

    // Building a bitwise random number
    for (i=0; i<8; i++) {
        // set bit as 1 or 0?
        if (getRand() > 0.5) {
            bit = 1;
        } else {
            bit = 0;
        }
        // get bit ready
        bitmask = bit << i;
        // or them together
        x = x | bitmask;
        printf("After round %i, ",i);
        printf("bit is %i, and ", bit);
        printf("x is %i.\n", x);
    }

    getchar();
    return 0;
}
```

10.2. Exercise

Dougie Dog has invented an "encryption" technique to keep secrets from the Ferocious Kittens. Fortunately, cats are extremely intelligent, and have cracked the simple code:

1. Letters are grouped into pairs. Add a space to the end of the string if necessary to give it an even number of characters (here "character" means `char`.
2. Make an `int` from each pair by sticking the bits from the first letter in front of the bits from the second letter. You may assume that we are using 8-bit ASCII.
3. XOR the result with 31337.

   Here's two examples of encryption: "cats" and "kittens".

   1. Pairs: "ca ts"
      "ki tt en s_" (_ represents a space)
   2. into ints: 25441 29811
      27497 29812 25966 29472
   3. XOR with 31337: 6408 3610
      4352 3613 7943 2377

Decryption is performed with the same steps, but in reverse order.

- The Ferocious Kittens have intercepted two secret messages from Dougie Dog:

```
15643 6913 6916 23040 2377 6985 6408 3657 5638
3084 2119 15910 23079 13629 23101 10300 10557
23073 13092 23369
```

- Write a program that decrypts them.
  (hint: this will be a lot easier if you begin by writing a program to *encrypt* values -- you can check each step with "cats" and "kittens" to make sure you understand the process!)
- You *must* use a function to split a large integer into two separate letters. This function *may not* print anything to the screen.
  (hint: how can a function return two values?)