University of Nottingham
UK | CHINA | MALAYSIA

EEEE1042/1032 - **Practical 2**
Tokens, Keywords, Identifiers.

This practical session gives you hands-on practice of the theories, concepts and codes taught in the lectures. It is best to work through the practical session while having the class notes beside you. You should submit your `.c` code to Moodle at the end of the session. Answer the questions in this practical **directly** in your `.c` file using comments and/or `printf()` to print to the screen.

For example, in your C code, include comments and printf()'s so I can tell which question your code is answering like this:

```
1 // Q1.1: What is the result when you add together a signed and an unsigned int?
  // Answer to 1.1: type in your answer here.
  printf("**************\nAnswer to 1.1\n***************\n");
```

Name your `.c` file as: `yourName_practical2_EEEE1042.c`. You can put all your answers into 1 huge `.c` file, or alternatively you can put each answer into separate `.c` files and name your files `yourName_practical2_EEEE1042.1.1.c` instead. In section 4, you should have one `.c` file per question, since the entire file will be modified for each question in that Section. Your C program should be compilable (no errors, warnings are OK, but good if you can get rid of warnings too) when you submit it.

Now you know how to answer the questions and how to submit your work in properly named `.c` files to Moodle, let's move on to the main part of the practical session.

1 Data Types: ints, floats, doubles, signed and unsigned

1. Write a C program to test how C performs when you add two `int`s, two `float`s and two `double`s together. Print the result as the corresponding type.

2. Add an `int` and a `float` together and try to print the result, once as an int and once as a float.
   Q1: What is the behaviour in C when you add an `int` and a `float` together? Take note of any warnings during compilation, what do these tell you?

3. Declare a variable as a `signed int`. Print it as an `unsigned int`. Make your variable negative and repeat.
   Q2: Are you able to print `signed` as `unsigned` and vice versa? Why do you get this result?

4. Add together a `signed` and `unsigned int`. Print as both `signed` and `unsigned`, try with different initial values including with negative numbers.
   Q3: What is the result? Why do you get this result?

5. Declare a variable as `double`. Print it as an `int`.
   Q4: What do you get and why?

To understand what you are getting here, think about how computers store and process numbers: as a sequence of bits. Those sequences of bits can mean different things as you change how the computer interprets the bits.

2 Data Types: chars and strings

A `char` is a type that (typically) holds 1 byte corresponding to an ASCII character. A string is an array of chars. You can declare `c` as a single char with:

2
```
char c;
```

and

3
```
char string[5];
```

as a string of 5 consecutive chars (they will be uninitialized). You can initialize and declare a string in a single command:

4
```
char string[]="Hello World";
```

which tells C to get enough memory to hold the entire string `Hello World`, plus the end-of-string character '\0' (also known as the NULL character) and assign it to the variable `string`. The end-of-string character indicates the position of the end of a string during printing, copying, comparing... functions.

Characters are printed with the `%c` conversion character to `printf`, while strings are printed using `%s` as the conversion character to `printf`. To get a little more technical, a string, is just a **pointer** to an array of characters. To make sense to `printf` (and other functions), that **pointer** should point to some memory holding something meaningful. To access an element of the array, you dereference it using the `[ ]` square brackets, so `string[3]` is the fourth element in the array pointed to by `string`. C and C++ begin their arrays from zero, so `string[0]` is the first element in the array pointed to by `string`. The square brackets in box 4 above indicate that `string` is actually a pointer to an array of chars. Another way to dereference the **pointer** is using the `*` operator. So `string[0]` is the same character as `*string`. You can think of a **pointer** as a memory address "pointing to" the location of the string in memory.

1. Declare a `char` as in box 2 above. Assign it a character, print it out using the `%c` conversion character as a `char`, and the `%d` conversion character as a digit.
   Q5: What do you get? Explain what you are observing.

2. Add one to `c`, and print it as above again. Add ten and repeat. Add 1000 and repeat.
   Q6: What do you get in each case? Why?

3. Declare a string as in box 4 above. Print it out using the `%s` conversion character.

4. Add one to `string[6]` and print it out again (this can be done with the `++` operator).
   Q7: What do you get and why?

5. Set `string[5]` equal to zero (NULL). Print the string out again.
   Q8: What do you get and why?

3 Simple math

1. Write a C program to convert from Farenheit to Celcius using the formula:

$$C = \frac{5}{9}(F - 32)$$

   where F and C are single precision. As we have not yet learnt about loops in the lectures, do this exercise manually without using them. Set F equal to 0, 20, 40, 60, 80 and 100, and print out both F and C in each case. You can do this using copy-paste in Emacs. `M-w` is the keyboard shortcut to copy, `C-w` is for cut and `C-y` is to paste. Use the `printf` formatting to make the output nicely aligned as:

F	C
0	-17.78
20	-6.67
40	4.44
60	15.56
80	26.67
100	37.78

   Make sure you have 2 decimal places for C and no decimal places for F in your output. In your formula for converting between Farenheit and Celcius try using each of the two formulas shown below:

```
5 C=5/9*(F-32)
```

   and

```
6 C=5.0/9.0*(F-32)
```

   Q9: Explain the difference in outputs you observe between the formulas used in box 5 and box 6.

2. Test out the relational operators `==` `<` and `>` by declaring two `int`s `x` and `y` and assigning them some values of your own choice. Print out the value of `(x==y)`, `(x<y)` and `(x>y)` using the `%d` conversion character to print.
   Q10: What numeric value does your computer use to represent TRUE and FALSE?

3. Test out the logical AND/logical OR by declaring two `int`s `x=8` and `y=5`. Print out the values of `(x&&y)` and `(x||y)` printing with `%d`. Repeat the printing after setting `x=0`. Experiment some more with different values of `x` and `y` until you understand what it is doing.

Q11: Explain your observations on what these two operators do and explain how they behave.

4. Test out the bit-wise operators `&` and `|`. Declare two `int`s `x=8` and `y=5`. Print out the values of `(x&y)` and `(x|y)` printing with `%d`. Try with different values of `x` and `y`.
   Q12: What do you observe? Record your understanding of how these two bit-wise operators behave here.

5. Test out the bit-wise operators `<<` and `>>`. Declare `int x=43`. Print out the values of `(x<<1)`, `(x<<2)`, `(x>>1)`, `(x>>2)` and `(x>>3)` using `%d`. Try with different values of `x`.
   Q13: What do you observe? Record your understanding of how these two bit-wise operators behave here.

4 Simple functions: declaration and definition

In this section, because we are testing functions, write a separate `main` function for each part and put them each into their own `.cpp` file labeled `yourName_practical2_EEEE1042.5.1.c` etc... Note: only one `main` function can be defined per `.c` or `.cpp` file. Write a simple program with a single subfunction called `mymax` as given in the class notes together with the standard `main()` function. Note that C++ already has a function named `max` defined, within its standard libraries, therefore you must use a function with a different name: `mymax`. Review the class notes to fully understand the difference between a **function declaration** and **function definition**. The function `mymax` declared and defined in the lecture notes, takes two `int`s as input and returns one `int` as output. Write a `main` function that declares 3 `int`s `x, y` and `z` and use them to test your function `mymax` by calling it as follows:

```
7 z = mymax(x,y);
```

and print out the values of `x, y` and `z` after that. Explore the following three situations and report what you find in each case:

1. Scenario 1: put the `main` function definition first, followed by the `mymax` function definition second. Compile and execute.
   Q14: Can you compile and execute? Explain what you observe.

2. Scenario 2: switch the order of the `main` and `mymax` function definitions, ie: have `mymax()` defined first in the file and `main()` second. Compile and execute again.
   Q15: Can you compile and execute now? Explain what has happened.

3. Note that up until this point, you should not yet have explicitly **declared** any function. You have only used **function definitions**, which can actually indeed serve to also **declare** the function to the compiler.
   Scenario 3: Revert back to the original order with `main` function defined first, and `mymax` function defined second. Now also put a **function declaration** just above the `main()` function as follows:

```
8 int mymax(int x,int y); //This is a function declaration
```

Try to compile and execute a third time.
Q16: Now are you able to compile and execute? Explain why you are able to compile and execute in one scenario, but not in the other.

If you are having difficulty understanding/explaining what is going on, review once again the class notes on function definitions vs function declarations and imagine what the compiler is doing as it is coming through your source code line-by-line.

Note that scenario 3 above is the more standard accepted way of declaring and defining functions in C/C++, with the **function declarations** sitting above the `main` and **function definitions** sitting either below or somewhere else. As the number of functions grows, **function declarations** are then typically shifted into a `.h` file which are `#include`'d into any program that wants to use them. The **function definitions** can reside in several places.

1. They could sit below the `main` function in which case the compiler will compile them into object code when it reaches them. After that all the object codes are linked together to form the executable.

2. They could sit in another `.c` or `.cpp` file. In this case you need to provide the name of that `.c` or `.cpp` file to the `gcc` or `g++` compile line. Otherwise the linker will not know where to find the needed **function definitions**. During compilation, the compiler will go into those extra supplied files, compile them into object code and link them with the `main` function.

3. They could sit in another `.o` or `.obj` file. In this case, somebody else has already done the job of compiling the source to object code, and you will find your compilation stage is faster. The name of the object code `.o` file must be supplied to `gcc` or `g++`. The compiler itself doesn't have to do anything to compile the **function definition** as it already exists as object code, but the linker just has to go into those supplied object code files to find the compiled **function definitions** and link them.

4. They could sit in a library. People will sometimes gather similar function definitions together into one big file, such as the math library having all the math **function definitions**. These libraries contain the object code for each function and therefore needs to have been compiled. When using such libraries, you must tell `gcc` or `g++` which library to use. This is done using the `-l` flag such as `-lm` for the math library. During linking, the compiler will search the library for the needed function definitions.