Lab Session – C Programming II

Objectives.

- Write simple programs in C using the preprocessor directives
- Measure the execution time of blocks of code
- Compare Floating Point numbers
- Write simple programs using structs

Aim

The **aim** of this lab session is to further acquaint yourselves with programming.

C Routines

A function is a group of statements that together perform a specific task. Every C program has at least one function, which is main() function. A function declaration tells the compiler about a function's name, return type, and parameters. A function definition provides the actual body of the function. You can find more information about C routines in [5].

Task0: In slide 3, you are provided with a program that finds the maximum number (a separate routine is implemented for this). Your task is to implement a program that finds the maximum number in an array of integers by re-using the max function.

C Preprocessor and Preprocessor Directives

The preprocessor is an important part of C/C++ programming language. Before the C/C++ compiler compiles the code, the preprocessor makes a pass through the program looking for preprocessor directives. The output of this preprocessor step is then processed by the C/C++ compiler. All preprocessor commands begin with a hash symbol (#).

The two most commonly used preprocessor directives are the following

#include: The #include directive brings in code from another file, typically a header file or library, e.g., # include <stdio.h> loads the code from this library.

 # define: In the C Programming Language, the '***#define***' directive allows the definition of macros within your source code, e.g., #define N 10, will create a constant value N=10 and it will replace N with 10 in the entire program. This macro definition allows a constant value to be declared for use throughout your code. Macro definitions are not variables and cannot be changed by your program code like variables.

Task1: Amend the previous week's program found in '***maxmin_value_solution.cpp***' *so as the array's input can be specified by the user, by using the define directive.* A solution is provided in the '***maxmin_value_solution_ver2.cpp***'

Measuring the execution time of a block of code.

C/C++ provide us with several functions being able to measure the execution time of a block of code. These functions are known as timers. The process of measuring the execution time of a block of code is the following:

```
start_time = timer();

Function();

end_time = timer();

Print(end_time-start_time);
```

Further understanding how these timers work is out of the scope of this module.

To accurately measure the execution time of a block of code, it should run for about 10 seconds (not less) and without using your computer at that time, e.g., do not open windows, watching videos, typing etc. This is because apart from our process, other processes use the hardware resources (e.g., cache, CPU) too. If the execution time of our process is much higher than the execution time of all the other processes, then the error is minimized. We will further analyze this later on.

What if the execution time of our block of code is less than 10 seconds? In this case, we must run this block many times and then divide the overall time by the number of iterations (average time).

Task2: Study the program in '***measure_time.cpp***' file. Measure the execution time of 'find_max()' function, when 'TIMES_TO_RUN=1'. Repeat the experiment many times and you will realize that the execution time you get is different every time. Increase the 'TIMES_TO_RUN' value so as the execution time is about 10seconds (e.g., TIMES_TO_RUN=200). You will realize that the execution time value you measure is more accurate now.

Task3: Measure the execution time of the previous function in Task2 when **a)** using the 'Start Debugging' option, **b)** Start without Debugging' option. The code runs slower in the first case because Visual Studio (VS) runs diagnostic tools and stores more information about the program's execution.

Task4: Run the previous program again and measure the execution time under Debug mode and under Release Mode. The code always runs faster under release mode because the compiler further optimizes the code in this case. How faster is the code now?

Task4: Run the previous program again and measure the execution time for different N values. Create a graph in Microsoft Excel showing the results (execution time in Y-axis and N in x-axis). Make sure you get accurate execution time values.

Floating Point (FP) arithmetic

'Squeezing infinitely many real numbers into a finite number of bits requires an approximate representation. Although there are infinitely many integers, in most programs the result of integer computations can be stored in 32 bits. In contrast, given any fixed number of bits, most calculations with real numbers will produce quantities that cannot be exactly represented using that many bits. Therefore,

the result of a floating-point calculation must often be rounded in order to fit back into its finite representation.' taken from https://docs.oracle.com/cd/E19957-01/806-3568/ncg_goldberg.html .

As you have already been taught, 0.1+0.2 does not make 0.3 in most programming languages. This is because computers cannot accurately represent these numbers. If you are confused, then read this article https://techtalkbook.com/why-0-1-0-2-does-not-equal-0-3/ .

The real question for us (programmers), is how can we compare two FP numbers? Are 0.000011 and 0.000012 considered equal or not? How can we know? A solution which works well in most cases is using the following formula (relative error):

If (abs(a-b) / abs(b)) < specific.value.defined.by.the.programer

//then the two numbers are considered equal

Else

//the two numbers are not considered equal

To better understand the above formula, let us consider the following two numbers, a=0.0002 and b=0.0001. Although both numbers' values look very close to each other and a-b is a very small number (a-b=0.0001), a is two times larger than b (a==2*b). Therefore, a and b differ a lot. This is why the absolute value of (a-b)/b=1. On the other hand, if we consider a=10.0002 and b=10.0001, then these two numbers' values are very close with each other; this is why the above formula gives (a-b)/b=0.00001. In the first case, the above formula gives a number close to 1, while in the second case a number much smaller than 1. This is why the specific value in the aforementioned formula should be small. For the rest of this module you can use the value of 0.00001.

The above code becomes problematic when b is zero or both a and b are zero, but we will not further investigate this issue. If you are curious you can read this article, but this is out of the scope of this module https://floating-point-gui.de/errors/comparison/.

Task5: In this task we will change the order of the executed instructions of a program that does FP calculations, and we will check whether the results are correct or not (propagation error). First, study the code in the '***FP.cpp***' file. Then, run the program for N=10, N=100 and N=4000. Use EPSILON=1e-7 (1x10-7 or 0.0000001). By increasing N value, the FP error is increased. This is because more FP calculations occur in this case (larger arrays), and the FP error is propagated from one calculation to another. Repeat the experiment for EPSILON=1e-6 (1x10-7 or 0.000001); there is no error in this case.

Structs

A structure is a user defined data type in C/C++. A structure creates a data type that can be used to group items of possibly different types into a single type. While arrays allow us to define multiple data items of the same type, a **structure** is another user defined data type available in C that allows to combine data items of different types.

A struct is defined as

```
struct Datetime {
```

```
    unsigned short int year;
    char month[10];
    char day[10];
    unsigned short int hours;
    unsigned short int minutes;
    unsigned short int seconds;
};
```

As you can see, structure members are of different type. Keep in mind that each datetime object occupies 28bytes of memory (consecutive memory locations).

Task6. Study the 'structs.cpp' program.

Build structs.cpp program and run it. Make sure you understand how it works.

Task7 (Optional - advanced). This tasks is only for those who want to stretch their programming skills. Drawing upon struct.c program, try to implement a C program that does the following. Create a struct named 'student' whose members are 'char student_name[20]', 'int module_code', 'float mark'. This struct contains the student's name, the module code, e.g., comp1001 and the mark achieved. Now, create an array of structs. Write a piece of code that initializes and prints the array of structs. A solution is found in 'structs_advanced_task.cpp' file.

What is a pointer?

Pointers is an advanced topic. However, there are many C library functions that their operands are pointers and therefore you need to understand what they are. **You will not write any programs with pointers. You just need to understand how they work.**

Task8 (optional). Study **'*pointers_1d_array_initialization.cpp*'** file.

This file has a routine that initializes a one-dimensional array and three routines that print the array's values, in three different ways, which are explained below. Note that the array elements are always stored in consecutive memory locations (always).

1. printf("\n element %d equals to %d",i, A[i]);
2. printf("\n element %d equals to %d",i, *(A+i));
3. printf("\n element %d equals to %d",i, *(ptr+i));

The three bullets above are equivalent. What does ***(A+i)*** mean? A is an array, thus A is the memory address of the first array element. ***(A+i)*** is the memory address of the ith array element. * refers to contents of memory address. Thus, ***(A+i)*** means : give me the content (value) of the ith array element. The same holds for the 3rd bullet as ***'ptr=&A[0]***', which means that the pointer shows to the memory address of A[0] (& means memory address of).

Compile and run the program using different 'N' values, to make sure that all the three routines print the right values.

Keep in mind that in C language, all the routines must be declared before main function.

It is always a good practice to assign a NULL value to a pointer variable in case you do not have an exact address to be assigned. A pointer that is assigned NULL is called a null pointer. This is done by using

*int *ptr = NULL;*

Further Reading

If you want to learn more about C programming you can study the following links:

1) Tim Bailey, 2005, An Introduction to the C Programming Language and Software Design, available at: http://www-personal.acfr.usyd.edu.au/tbailey/ctext/ctext.pdf
2) C examples, Programiz, available at https://www.programiz.com/c-programming/examples
3) C Programming examples with Output, Beginners book, available at https://beginnersbook.com/2015/02/simple-c-programs/
4) C Programming Tutorial, from tutorialspoint.com, available at https://www.unf.edu/~wkloster/2220/ppts/cprogramming_tutorial.pdf
5) C functions, available at https://www.tutorialspoint.com/cprogramming/c_functions.htm