THE UNIVERSITY COLLEGE
OF THE CARIBOO
Computing Science

**Lab Exercises #3 – C - *Memory***

Marks: 5 marks | Due: October 4th, 2004 (in lecture)

## Introduction

Modern computer programs are nothing more than a collection of instructions and data, called the von Neumann "stored program" model (see page 48 in the lecture text). This model allows for programs and data to be stored on the same medium (ex: disk) and loaded into contiguous memory (specific memory is not required for instructions and data, they can be mixed).

The reason is that, interestingly, programs themselves are only sequences of data to the Operating System and CPU. Although we like to think our instructions are important, they are just answers to the CPU asking the question, "What do I do next?"

So how is memory organised for programs, at least from a data-perspective?

## Compilers and Tools

Use the techniques and resources from the previous lab exercises to complete the following.

## Exercises / Programming Problems

1. Write a program that produces a display of the byte-sizes of the standard data types in C: **char, short, int, unsigned int, long, unsigned long, float, double, long double,** and **void**.

   Use the example statement below, but change accordingly for each type. Also include a statement that displays the size of a pointer: use **sizeof(int*)** – *is the size different if* ***double**** *is used*?

   Example statement showing the size of an **int**,

   ```
   printf("Size of int %d bytes\n",sizeof(int));
   ```

2. A common technique in memory programming is called "pointer arithmetic."

   Essentially, pointer arithmetic uses the condition that a pointer is declared to point to a memory block of a particular size (based on the type). Adding or subtracting to the pointer variable makes it point to a block *after* (adding) or *before* (subtracting) where it currently points to.

   Write a program that tests this technique:

   - declare a dynamic array of 5 **int** elements (using **malloc()** or **calloc()**, and fill the array with the values of the index (so first element =0, second =1, third =2, etc.);
   - display the values of in the array using a simple loop, and the standard array reference with [ ] *(using an array reference is also an example of "pointer arithmetic"; why?)*
   - declare a new pointer of type **int** (called, for example, **pitem**); point to the beginning of the array, and display the values in the array, but using the value of ***(pitem+i)** instead of an array reference
   - reset the pointer to the beginning of the array, and display the array values using ***(pitem++)**

   *What advantages are there of using these techniques rather than the array reference?*

3. Write a simple program that attempts to allocate all available memory (or at least as much as the OS gives to the program). Use the fact that C does not release memory unless specifically told to (through **free()** or until the program terminates), which means the program can keep allocating memory and returning the reference to the same pointer variable.

   Rather than allocating a single block of type **int** or **double**, try allocating an array's worth (50 or 100 blocks at a time). Display a continuous status of how much memory has been allocated until the program crashes.

   *No screen capture is required for this program, but the source code and final memory amount is required.*

**Challenge** (this program is a little strange!)

4. Write a program that shows the hexadecimal value of each *byte* that makes up an **int**, using a pointer to the **char** data type as the reference to the specific bytes. The goal of the program is to examine how integer values are stored in memory, at the binary level.

   Use the following suggestions and hints:

   - have the program execute in a loop until the user wishes to stop
   - ask the user for the **int** value to show (this way the program can be tested with multiple values, including simple, small ones first first: 0, 1, 2, 128,…)
   - memory is organised on the smallest type: the *byte level*
   - a pointer can point to any memory location (although the compiler may warn against an incompatible pointer reference, such as a **char**-pointer pointing to an **int**-block)
   - memory on Intel CPU-based systems defines types with the low-end byte coming first followed by the high-end bytes; which means that the 4-bytes in an **int** (1,2,3,4) are in memory as (4,3,2,1)
   - apply the concept of "pointer arithmetic" to determine all parts of the **int** value
   - to display the character value as a hexadecimal, use the formatting symbol: **%x**
     for example: **printf (" %x ", v);** where **v** is an integer variable (**char, short, long,** etc).
   - test your program with a series of values, expecially ones that are larger than 255 (implying a number larger than one byte); also try a few negative values.
   - *rather than assume that your final program works correctly, in your tests, determine what the integer values will look like in binary, then hexadecimal, before entering the value in the program, for example,*

     value $\mathbf{11}_{10}$ represented in binary as: 00000000 00000000 00000000 $00001001_2$
     the binary value will be stored as: 00001001 00000000 00000000 00000000
     or in hexadecimal as: 09 00 00 00 (or as shown in C: 9 0 0 0)

     value $-\mathbf{1}_{10}$ represented in binary as: 11111111 11111111 11111111 $11111110_2$
     the binary value will be stored as: 11111110 11111111 11111111 11111111
     or in hexadecimal as: FE FF FF FF (fffffffe ffffffff ffffffff ffffffff )

## Conclusion

Submit the well-documented source code for the program: #2, #3, and #4. Include output captures.

Attach a cover page, with: *course number, exercises/assignment title, your name, date of submission*.

> ***Note**: Once again, well-documented describes: complete* file documentation *(file name, author, date of last update, and purpose of program); proper comments for variable declarations; proper comments for major segments and statements (such as input / output sections, loops, and function definitions).*
>
> *Also, the programs must have clear and understandable user-interaction.*