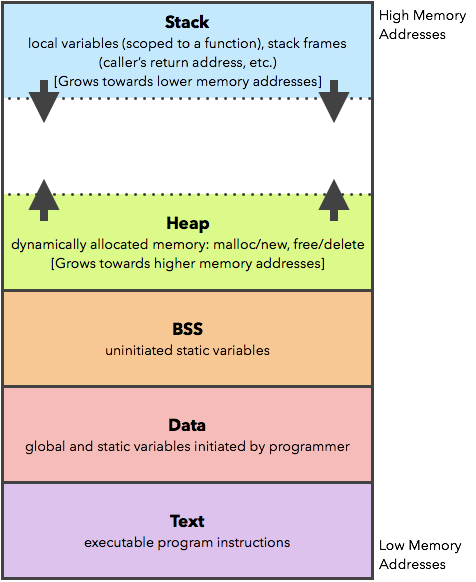
**ECE 260: Fundamentals of Computer Engineering – Lab #2  
Introduction to Computer Architecture**

**Name \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Lab Partner(s) \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**1. Introduction**: This lab provides a hands-on overview of computer architecture and organization topics. Students will explore how variables and code are stored in memory.

**2. Background**: Both data (variables) and control (instructions) are stored in the same physical memory within a computer. The computer executes the instructions found in memory and performs the instructions specified. This is known as the “stored program” concept.

In C++, the memory address of any function or variable can be found using the ‘**&**’ operator. For example, the memory address of a variable declared as ‘**int a**’ can be found by writing ‘**&a**’. Likewise, the memory address of a function can be found with ‘**&functionName**’. For example, the operation ‘**&main**’ will return the memory address where the instructions for the ‘**main**’ function are stored. Although variables and functions are placed in the same physical memory, they are stored in different regions of the memory (see figure to the right). Functions and all other executable program instructions are stored in a memory region called the ‘**Text**’ section. Global and static variables that are initialized by the programmer (including constants) are stored in the ‘**Data**’ section. Static variables that are declared by the programmer but not initialized are stored in the ‘**BSS**’ section. Variables and data structures that are allocated using ‘**new**’ or ‘**malloc**’ are stored in the ‘**Heap**’. Variables that are locally scoped to a method or function are allocated in the ‘**Stack**’ section.

Calls to ‘**new**’ or ‘**malloc**’ allocate blocks of memory to the program. Each block of memory that is requested and allocated is allocated as a contiguous block of memory. The size of the memory block is determined by the arguments that are passed to the ‘**malloc**’ or ‘**new**’ functions. For example, the code ‘**new int**’ would allocate sufficient memory to store a single integer on the heap. The code, ‘**new int[5]**’ would allocate enough memory to store five integers on the heap.

Memory addresses are typically expressed in hexadecimal. Hexadecimal is a base-16 numbering system that replaces decimal values 0-15 with 0-F notation. You can use the Windows Calculator (in Programming Mode) or Google to help you convert between decimal and hexadecimal values (or develop some mad skills and do it in your head … clearly, the superior choice).

**3. Procedure:** The program you have been provided is largely blank but contains several helper functions that will enable you to complete this lab. Throughout the lab you will be asked to print information to the console. You can use whatever print method you like (‘cout’ or ‘printf’). A helper function ‘print\_ptr\_info(string text, void\* ptr)’ has been provided that prints out the address and value of a pointer, along with some descriptive text that you can provide as the first argument to the function. Here’s an example of printing the variable ‘int myInt’:

print\_ptr\_info("myInt", &myInt);

Throughout the lab you will be asked to print out the address of several variables and functions. Table 1 is provided for your convenience to organize this information.

**3.1 Programming Tasks**

|  |
| --- |
| **TODO Task #1:** Use ‘new’ to create an array of five integers such that ‘long\* array‘ points to the head of the new memory block (i.e. create an array that can store five integers and assign it to an ‘long\*’ variable named ‘array’). Also, print the memory address and value of ‘array’ using the ‘print\_ptr\_info’ function. |
| **TODO Task #2:** At this point, the memory for ‘array’ has been allocated but it has not yet been initialized. The memory contains unknown garbage data. Iterate over ‘array’ and print out the memory address and the value of each index of the array using the ‘print\_ptr\_info’ function. **HINTS:** The first argument to the ‘print\_ptr\_info’ function is a ‘string’ value. You can concatenate ‘string’ values together using the ‘+’ operator. Additionally, you can convert an integer into a ‘string’ using the function ‘to\_string(val)’. Finally, recall from the background section that you can get the memory address of a value using the ‘&’.  **TODO Task #3:** Iterate over ‘array’ again. During this iteration, initialize each element to ‘100 + index’. That is, index 0 should contain the value 100, index 1 should contain the value 101, etc. This will overwrite the garbage data that was previously present in ‘array’. |
| **TODO Task #4:** After initialization, iterate over ‘array’ a third time. Once again, print out the memory address and the value of each index of the array using the ‘print\_ptr\_info’ function. You should see different values than those that you saw in task #2. |
| **TODO Task #5**: Create a new ‘long\* ptr’ and set that pointer equal to the ‘array’ created in task #1. Print out the memory address and value of ‘ptr’ using the ‘print\_ptr\_info’ function. Does that value of ‘ptr’ match the address of ‘array[0]’? Next, use the ‘+’ operator and add 3 to your ‘ptr’. This is called pointer arithmetic and will cause ‘ptr’ point to ‘array[3]’. Print out the memory address and value of ‘ptr’ once again. Does that value of ‘ptr’ match the address of ‘array[3]’ now? |
| **TODO Task #6**: Use ‘print\_ptr\_info’ to print out the memory address and value of ‘main()’, ‘print\_ptr\_info()’, and ‘get\_num\_processors()’. Note that you will need to cast your function names as ‘(void\*)’ to pass them to ‘print\_ptr\_info()’. Here’s an example:  print\_ptr\_info("main()", (void\*)&main); |
| **TODO Task #7**: Not all variables types consume the same amount of memory. Use the function ‘sizeof(<type>)‘ to determine the size (in bytes) of the following variable types: bool, char, short, int, long, long long, float, double, char\*, int\*, and long\*. Print their sizes using either ‘cout’ or ‘printf’. The size of a data type defines how much information can be stored in that data type. Larger data types can store larger or more precise numbers. Keep in mind, however, that the size of these data types may vary from machine to machine!  **TODO Task #8**: The function ‘get\_num\_processors()’ is defined in the ‘functions.cpp’ file. Use ‘get\_num\_processors()’ to determine the number of processors in the system and print your result. |

|  |
| --- |
| To compile and run your program, simply type ‘**make**’ from within your ‘**Lab02\_Computer\_Architecture**’ and then **‘./main.exe**’. A sample of what your finished program should look like is shown below. Of course, the memory addresses may be different than those shown. Use your console output to fill in the tables and answer the questions on the following pages. |
| Figure 1: Example output of program |

Table 1: Location in memory of functions and variables

|  |  |  |
| --- | --- | --- |
| **Variable / Function** | **Address (Hexadecimal)** | **Value (if applicable)** |
| long\* array (from task #1) | 0x00000000ffffcb70 | 0x0000000600012910 |
| array[0] (from task #2) | 0x0000000600012910 | 0x00000001802fa0c8 |
| array[1] (from task #2) | 0x0000000600012918 | 0x00000001802fa0c8 |
| array[2] (from task #2) | 0x0000000600012920 | 0x000005010000004c |
| array[3] (from task #2) | 0x0000000600012928 | 0x0000001505000000 |
| array[4] (from task #2) | 0x0000000600012930 | 0xbe8a98870abc9388 |
| array[0] (from task #4) | 0x0000000600012910 | 0x0000000000000064 |
| array[1] (from task #4) | 0x0000000600012918 | 0x0000000000000065 |
| array[2] (from task #4) | 0x0000000600012920 | 0x0000000000000066 |
| array[3] (from task #4) | 0x0000000600012928 | 0x0000000000000067 |
| array[4] (from task #4) | 0x0000000600012930 | 0x0000000000000068 |
| long\* ptr (beginning of task #5) | 0x00000000ffffcb68 | 0x0000000600012910 |
| long\* ptr (end of task #5, after adding 3) | 0x00000000ffffcb68 | 0x0000000600012928 |
| **main()** | 0x0000000100401350 | 0x0000d8ec81485355 |
| **print\_ptr\_info()** | 0x0000000100401080 | 0x000138ec81485355 |
| **get\_num\_processors()** | 0x00000001004012c1 | 0x30ec8348e5894855 |

Table 2: Variable types and their lengths

|  |  |
| --- | --- |
| **Variable Type** | **Length (bytes)** |
| **bool** | 1 byte |
| **char** | **1 byte** |
| **short** | **2 bytes** |
| **int** | **4 bytes** |
| **long** | **8 bytes** |
| **long long** | **8 bytes** |
| **float** | **4 bytes** |
| **double** | **8 bytes** |
| **char\*** | **8 bytes** |
| **int\*** | **8 bytes** |
| **long\*** | **8 bytes** |

**3.2 Free-response Questions (alter the space as needed)**

|  |
| --- |
| 1) Describe the relationship between ‘long\* array’ and ‘array[0]’. Are they the same location in memory or different? What is the relationship between the *value* of ‘array’ and the *address* of ‘array[0]’?  The long\* array and array[0] are linked because long\*array’s value is pointing to the address of array[0]. |
| 2) In task #5, how do you know that ‘ptr\*’ is correctly pointing to ‘array[3]’ after you added 3? What elements of the console output tell you that your operations are correct?  I know it’s in the correct spot because there is an 24-byte difference between values stored inside the ptr\* because the length of a long\* is eight bytes the “+ 3” operation is moving the value 3 long\* lengths ahead in value. |
| 3) Examine the addresses of the functions in Table 1 (i.e. ‘main()’, ‘print\_pointer\_info()’). Are they near or far from the memory locations of the variables? Why might this be so?  They are far because they are functions and functions are stored in the TEXT section of the memory and the variables are stored in the stack. |
| 4) Examine the processor for your computer. Click the magnifying glass icon in the Windows taskbar and search for ‘**about**’. Select ‘**About your PC**’ from the search results. In the ‘**About**’ dialog box, identify the processor for your system (see Figures 2 and 3 on the next page for an example). Use the Intel ARK site ([link](https://ark.intel.com/" \l "@Processors)) to get information about your specific model. For the example shown in Figure 3 below, you could search for ‘**i7-7700T**’ on Intel’s website. Answer the following questions:   |  |  | | --- | --- | | How many cores and threads are supported by your processor? | 4 cores, 8 threads | | What is the ***base*** operating frequency? | 2.9GHz | | What is the maximum ***turbo*** operating frequency? | 3.80GHz | | If your processor supports more threads than cores, identify the Intel technology (under Advanced Technologies) that enables your processor to do so. What is it called? | Hyper-threading Technology | | Is your processor a 32-bit or 64-bit processor? | 64-bit | |

|  |  |
| --- | --- |
| Figure 2: Pulling up My Computer properties | Figure 3: Determining your processor architecture |

**4. Submission**

When you have finished your lab, demo your program for your instructor. Write your answers to the above questions electronically in this document. To submit your lab assignment, make sure your files have all been saved (*including this file*). In a Cygwin window type the commands:

cd h:  
cd ECE260  
cd Lab02\_Computer\_Architecture  
make submit

Enter your Marmoset username and password (which you should have received by email). Note that your password will not be echoed to the screen. Make sure that after you enter your username and password, you see a message indicating that the submission was successful. Log into [Marmoset](https://cs.ycp.edu/marmoset/login) via the web to check the files you submitted to ensure they are correct.

**DO NOT MANUALLY ZIP YOUR PROJECT AND SUBMIT IT TO MARMOSET. IT WILL NOT BE GRADED!!!  
YOU MUST USE THE make submit COMMAND.**