| Week 08, Lab 08 | Weight: 1% | Due: End of your stream's week 8 lab session (via `sync`) |
|---|---|---|

**Pre-lab Preparation:**

- **Week 1, 2, 3, 4, 5, 6, 7 Lectures, Week 1, 2, 3, 4, 5, 6, 7 Labs**
- **Week 8, Lecture 019**

---

**Lab Activities:**

Remember to **`sync`** to obtain the lab starting code.

Pointers provide benefits to C programmers:

- Data allocations at runtime.
- Memory sharing.
- Resizable data structures.

As you work through the lab exercises, if you encounter programming terminology that you do not understand:

- Refer to the Programming 1 lecture materials.
- Ask a Lab TA for clarification on what the term means.
- After attempting an activity, if you are stuck for an unreasonable amount of time, seek help from a Lab TA! Do not wait too long to seek help!

## Exercise 1: Introduction to Pointers

Complete the following tasks, in order:

- Declare a main function in **`lab08ex01.c`** which returns zero.
- Call **`printf`** with the output message: **`"Lab08: Exercise 1: \n"`**
- Declare an integer variable called **`my_age`**.
- Assign a literal to the **`my_age`** variable which represents your actual age.
- Declare a pointer to an integer, called **`my_pointer`**.
- Assign the address of **`my_age`** to the pointer variable **`my_pointer`**.
- Using **`printf`**:
    - o Call **`printf`** once to display the contents of **`my_age`**.
    - o Call **`printf`** once to display the contents of **`my_pointer`**.
    - o Call **`printf`** once to display the value pointed to by **`my_pointer`**.
- Call **`printf`** again with the output message: **`"Indirection test! \n"`**
- Change the integer value pointed to by the pointer **`my_pointer`**.
    - o Dereference the pointer, assign to the value-pointed-to the value of zero.
- Using **`printf`**:
    - o Call **`printf`** once to display the contents of **`my_age`**.
    - o Call **`printf`** once to display the contents of **`my_pointer`**.
    - o Call **`printf`** once to display the value pointed to by **`my_pointer`**.

The output should display as follows, however you will need to complete the missing parts (the underscores) with the runtime output of your program:

```
Lab08: Exercise 1:
my_age holds the value ________
my_pointer holds the value ________
my_pointer points to the value ________
Indirection test!
my_age holds the value ________
my_pointer holds the value ________
my_pointer points to the value ________
```

Without editing your program, run the executable again. Record the new output:

```
Lab08: Exercise 1:
my_age holds the value ________
my_pointer holds the value ________
my_pointer points to the value ________
Indirection test!
my_age holds the value ________
my_pointer holds the value ________
my_pointer points to the value ________
```

Answer the following questions in the handout:

| |
|---|
| *What is a pointer?* |

| |
|---|
| *What values change between the two consecutive executions of the program?* |

| |
|---|
| *Why do these values change between executions?* |

## Exercise 2: Sharing Stack Memory with Pointers

Pointers allow for pass-by-reference argument passing.

In **`lab08ex02.c`**, write a function call **`roll_dice`**, which rolls two six-sided dice. You will need to use **`rand()`** and modulus to generate the random numbers.

The return value of the function must be the sum of the face values of the two die. Additionally, using pointers, and pass-by-reference behaviour, the **`roll_dice`** function must take in two integer pointer parameters, called **`address1`** and **`address2`**. The first parameter represents one dice, the second parameter represents the other.

Declare a **`main`** function, with two local integer variables, called **`dice1`** and **`dice2`**.

Call **`roll_dice`** from the main function, passing the address of the two local variables from the main function into the **`roll_dice`** function.

Declare a third local variable in **`main`**, called **`total_roll`**, and assign the result of the **`dice_roll`** call to it.

Add **`printf`** calls to your program to achieve the following output (note the **`?`** marks will depend on your program's runtime evaluation). All print out lines prefixed with **`main`** are generated by the **`main`** function, likewise the **`roll_dice`** prefixed lines come from the **`roll_dice`** function:

```
main: Lab 08: Exercise 2:
main: Starting main function:
main: variable dice1 holds the value: 0
main: variable dice1 stored at: ????????
main: variable dice1 holds the value: 0
main: variable dice2 stored at: ????????
main: calling: roll_dice(????????, ????????);
roll_dice: Starting roll_dice function!
roll_dice: variable address1 holds the value: ????????
roll_dice: variable address2 holds the value: ????????
roll_dice: ROLLING TWO DICE!
roll_dice: Assigning first dice to caller's memory...
roll_dice: Assigning second dice to caller's memory...
roll_dice: Returning sum of two dice...
main: variable dice1 holds the value: ?
main: variable dice1 stored at: ????????
main: variable dice1 holds the value: ?
main: variable dice2 stored at: ????????
main: variable total_roll holds the value: ?
main: Returning zero to the operating system...
```

Answer the following questions in the handout:

| |
|---|
| *Which variables in this program were stored in the* **`main`** *function's stack frame, and what are their types?* |

| |
|---|
| *Which variables in this program were stored in the* **`roll_dice`** *function's stack frame, and what are their types?* |

| |
|---|
| *What is passed-by-reference to the* **`roll_dice`** *function?* |

| |
|---|
| *What is passed-by-value to the* **`roll_dice`** *function?* |

**Have a lab TA review your completed exercises 1 and 2 for this lab session. See the end of this document for the review questions**

## Exercise 3: Memory Sharing with Pointers

Type the following program source code into **`lab08ex03.c`**:

```
void zero_out_array(int* p_array, int num_elements)
{
    printf("zero_out_array called: \n");
    // Insert code here...
}

void print_array(int* p_array, int num_elements)
{
    printf("print_array called: \n");
    // Insert code here...
}

int main(int argc, char* argv[])
{
    int main_array[] = { 10, 20, 30, 40, 50 };

    // Insert code here...

    return (0);
}
```

Next, in the **`main`** function, call the **`print_array`** function passing in the **`main_array`** pointer, and an appropriate value for **`num_elements`**. Then from main, call **`zero_out_array`**, with the main_array and appropriate parameters. Then call **`print_array`** with **`main_array`** again.

The two functions, **`print_array`** and **`zero_out_array`** will require array iteration. You must use the **`[ ]`** bracket array index access notation in one of these functions, and the dereference pointer arithmetic notation in the other. You can choose which technique you apply where.

Write code in **`print_array`** which will iterate though the **`p_array`** printing each element.

Finally, write code in **`zero_out_array`** which will iterate through the array **`p_array`**, setting each element to zero.

| |
|---|
| *When pointer arithmetic is used on the* **`p_array`** *pointer, how many bytes does the address move per element?* |

| |
|---|
| *Why might it be useful to set the elements in an array to zero?* |

| |
|---|
| *Are the values in* **`main_array`** *passed by value, or passed by reference, to* **`print_array`**? |

| |
|---|
| *Is the value stored in* **`p_array`** *passed by value, or passed by reference, to* **`print_array`**? |

## Exercise 4: Heap Data Allocations with Pointers

In **`lab08ex04.c`**, write a program which queries the user for the number of rugby games played by a team. After obtaining the number of games played, the program must request from the user the score for each game played by the team.

Once the scores have been obtained, calculate and display the average score. Then display the list of scores entered, and state whether each score was above, below or equal to the average, followed by a tally of each category.

To program this, use a heap allocation, allocate a dynamic array of the size required by the user.

The program should output and function as follows:

```
How many games has the rugby team played? 8
-Enter score 1: 25
-Enter score 2: 35
-Enter score 3: 13
-Enter score 4: 33
-Enter score 5: 11
-Enter score 6: 18
-Enter score 7: 16
-Enter score 8: 24
The average score is: 21
Analysis:
-Score 1, 25, is above average.
-Score 2, 35, is above average.
-Score 3, 13, is below average.
-Score 4, 33, is above average.
-Score 5, 11, is below average.
-Score 6, 18, is below average.
-Score 7, 16, is below average.
-Score 8, 24, is above average.
-Number of scores above the Average: 4
-Number of scores equal to the Average: 0
-Number of scores below the Average: 4
```

Run your program with various input values, i.e.: different number of games played, and different scores.

| |
|---|
| *How can your program ensure that there are no memory leaks present when the program exits?* |

| |
|---|
| *What is the different between a "static array" and a "dynamic array"?* |

| |
|---|
| *Why is a dynamic array required in this application?* |

## Exercise 5: Resizable Data Structure

Type the following program source code into **`lab08ex05.c`**:

```
int* resize_dynamic_array(int* p, int old_size, int new_size)
{
    printf("resize_dynamic_array called: \n");
    // Insert code here...
}

int main(int argc, char* argv[])
{
    int* data = 0;

    printf("1) data starts at: %p \n", data);
    data = resize_dynamic_array(data, 0, 10);

    printf("2) data starts at: %p \n", data);
    for (int i = 0; i < 10; ++i)
    {
         main_array[i] = i * 2;
    }

    data = resize_dynamic_array(data, 0, 15);

    printf("3) data starts at: %p \n", data);
    for (int i = 5; i < 15; ++i)
    {
         data[i] = i * 3;
    }

    printf("4) data starts at: %p \n", data);
    for (int i = 0; i < 15; ++i)
    {
        printf("data[%d] is storing: %d \n", i, data[i]);
    }

    return (0);
}
```

Next, in the **`resize_dynamic_array`**, make a heap allocation with **`malloc`** based upon the **`new_size`** requirement. Then copy the elements from the existing array, **`p`**, into the newly malloc'ed array. Ensure **`resize_dynamic_array`** deallocates the old array. Finally, return the address of the newly allocated array.

Run your completed program. Note the output.

## Exercise 6: Three-way C String Joiner

Type the following program source code into `lab08ex06.c`:

```
char* join_three_strings(char* s1, char* s2, char* s3)
{
    // Insert code here...
}

int main(int argc, char* argv[])
{
    char hello[] = "HELLO";
    char p1[] = "Programming 1";
    char students[] = "STUDENTS!";

    char* joint = join_three_strings(hello, p1, students);
    printf("JOIN RESULT 1: %s \n", joint);
    free(joint);
    joint = 0;

    char* joint = join_three_strings(p1, hello, students);
    printf("JOIN RESULT 2: %s \n", joint);
    free(joint);
    joint = 0;

    return (0);
}
```

In the function, `join_three_strings`, allocate a `char` array on the heap which will contain the merged C string.

The output of your program should be as follows:

```
JOIN RESULT 1: HELLO-Programming 1-STUDENTS!
JOIN RESULT 2: Programming 1-HELLO-STUDENTS!
```

*Why does the* `main` *function call* `free(joint)` *twice?*

*Explain the difference between* `char* p = "Hello";` *and* `char p[] = "Hello";`*, use a diagram to enhance your explanation:*

## Week 08, Lab 08 Submission:

Run the **`sync`** command to submit your completed lab work.

Shutdown your Raspberry PI by pressing **`ALT-CTRL-DEL`**. Power-down and pack up your Raspberry Pi kit.

## Marking Criteria:

Have you completed each of the following? Have you submitted your code from lab?

| **Marking Criteria:** | **Week 08 Lab 08<br>Weight 1%** | **Yes** | **No** |
|---|---|---|---|
| Ex 1: | Required output matches executable's output? | | |
| | Questions answered correctly on handout? | | |
| Ex 2: | **`roll_dice`** function created with required signature? | | |
| | Implementation of **`roll_dice`** matches requirements? | | |
| | Required output matches executable's output? | | |
| | Questions answered correctly on handout? | | |
| Ex 3: | **`print_array`** implemented correctly? | | |
| | **`zero_out_array`** implemented correctly? | | |
| | Each function uses a different technique (bracket notation or pointer indirection) to access elements in the array? | | |
| | Main calls **`print_array`** and **`zero_out_array`** appropriately? | | |
| Ex 4: | Required output matches executable's output? | | |
| | User choose to enter any number of scores? | | |
| | Scores are classified correctly? | | |
| | Dynamic heap allocation used? | | |
| Ex 5: | **`resize_dynamic_array`** implemented correctly? | | |
| Ex 6: | **`join_three_strings`** implemented correctly? | | |

## Next activity: Homework 8 and Final Week 8 Lecture