**Think-Pair-Share Activity 1**

1. **Name the 3 pools for memory and what kind of variables will be stores in each pool?**

The Heap, Stack and Static. Static memory stores global variables. The heap stores dynamic memory allocation. The Stack stores local variables.

1. **Open mem.c”:**
   1. **How many variables are declared?**

There are 3 variables.

* 1. **How many of them are pointers? What type of data does each pointer point to?**

There are 2 pointers. **handle** points to **ptr**, and **ptr** points to **num**.

* 1. **Which pool of memory are these variables stored in?**

Stack since they are all local variables.

* 1. **Which pool of memory will the pointer ptr point to in line 12?**

It is going to point to the heap because of **malloc().**

1. **Using a piece of paper (or a drawing app), draw the 3 pools of memory and indicate the locations (in which pool?) of the variables in mem.c using boxes(like what we did in lecture). Label the boxes with variable names, their content, and their addresses. You will need to insert extra code to obtain the addresses of these variables.**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **S**  **T**  **A**  **C**  **K** | |  |  | | --- | --- | | **num** | **xxx** | | **ptr** | **addr1** | | **handle** | **addr2** | |
| **H**  **E**  **A**  **P** | |  |  |  | | --- | --- | --- | | **int1** | **xxx**  **(ptr points here)** |  | | **addr1**  **(int2 points here)** | **addr+4** | | **int2** | **addr1**  **(handle is pointing here)** |  | | **addr2** |  | |
| **S**  **T**  **A**  **T**  **I**  **c** |  |

1. **In the same drawing, use arrows to connect each pointer to its destination.**

**I color coded my table so that you could tell where each variable points to, instead of using arrows to point to the specific addresses.**

**TPS Activity 2: Structures in C**

1. **What does this program do?**

It creates a Node and the node creates an integer a float and points to the next space**.**

1. **Insert extra code to print out addresses of** head**, value of** head**, addresses of** iValue, fValue, **and** next **pointed by** head.

**printf("Address of head: %p\n", &head);//address of head**

**printf("Value of head: %d\n", head);//value of head**

**printf("Address of iValue: %p\n", &head->iValue);//address of iValue**

**printf("Address of fValue: %p\n", &head->fValue);//address of fValue**

**printf("where head points next: %p\n", &head->next);//next pointed by head**

1. **Based on the addresses of the members of Node structure, what do you observe about how structures are stored in memory? What is the relationship between the pointer (head) and its destination (the Node structure)?**

They’re stored in a linked-list like structure 4 bytes apart from one another since they are integers. The relationship between head and it’s destination is simple: it is just like a pointer which stores both the value and the address of iValue.

**Arrays and pointers**

1. This program will store integers entered by a user into an array. It then calls **bubbleSort** to sort the array. Study the code in **bubbleSort** to refresh your memory on Bubble Sort algorithm and answer the following questions:

a. Why do we need to pass the size of array to the function?

We need to pass the size of the array to the function so that bubble sort will not compare elements that are out of bounds. This is also done in order to allocate memory for the new array where we will be copying our values.

b. Is the original array (the one being passed into the function) changed at the end of this function?

No, the array that is passed into this function never changes, is is just used to copy into the other pointer array.

c. Why do you think a new array (**s\_array**) is needed to store the result of the sorted values (why not update the array as we sort)? Hint: look at what the **main** function does.

We use **s\_array** to store our sorted array, and the original array is left as it is because we want to show our user what the unsorted array looks like compared to the sorted arrays.

2. Once you remember how Bubble Sort works, **re-write** the code so that you are accessing the array’s content using **pointer notations(\*s\_arr).** i.e. you cannot use **s\_arr[j]** anymore. Comment out the original code so the algorithm won’t be run twice.

for (i = 0; i< n-1; i++){

for(j = 0; j<n-1;j++){

if(\*(s\_arr+j) > \*(s\_arr+j+1)){

temp = \*(s\_arr + j +1);

\*(s\_arr + j+1) = \*(s\_arr+ j);

\*(s\_arr+j) = temp;

}

}

}

3. After the array is sorted, the program will ask user to enter a key to search for in the sorted array. It will then call **bSearch** to perform a Binary Search on the array. Complete the **bSearch** function so that it implements Binary Search recursively (no loop!) You must use pointer notations here as well. Pay attention to what is written in main so your bSearch will return an appropriate value.

if (b >= a) {

int mid = a + (b - a) / 2;

if (\*(arr+mid) == key)

return mid;

if (\*(arr+mid) > key)

return bSearch(arr, a, mid - 1, key);

return bSearch(arr, mid + 1, b, key);

}

return -1;// We reach here when element is not present in array

}