CSC 2400 – Understanding the C memory model

One of the hallmarks of a good C programmer (and really any programmer in general) is having a strong mental model of how C handles memory. In this part homework, we'll go through a series of exercises to strengthen our understanding of how our code manipulates memory.

Memory diagrams

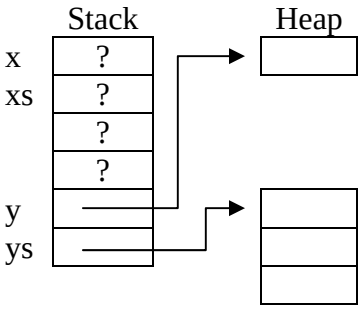
First, we'll step away from coding for a bit to drill down on how each instruction we issue modifies memory. We'll instead be creating memory diagrams that show what memory looks like after we execute our programs. Outside of an academic exercise, memory diagrams are a great tool to understand complex code. If you're ever stuck trying to debug a memory leak, segmentation fault, or some otherwise ungodly piece of C code, best thing to do is bust out a piece of paper and trace what's going on.

You've already seen the diagrams in lectures, but for sake of clarity, we'll step through the basics here so you can get a feel for how to create these diagrams on your own.

Recall that the memory of our program is divided into the stack, the portion of memory dedicated to local variables and function stack frames, and the heap, the portion of memory dedicated to dynamic allocation via malloc. Thus whenever we declare a local variable, we are actually allocating space on the stack (and giving that space a name) and when we call malloc, we are actually allocating space on the heap (and receiving a pointer to that memory).

```
int x;
int xs[3];
int *y = (int *)malloc(sizeof(int));
int *ys = (int *)malloc(3 * sizeof(int));
```

After this block of code executes, we will have the following stack and heap:



Some things to note in this diagram:

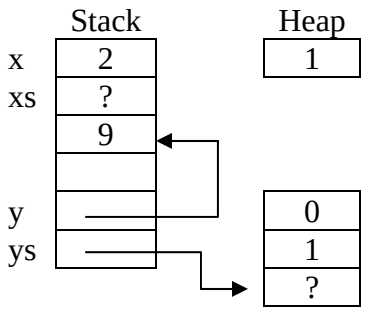
- Each declaration allocates some storage on either the stack or heap. In the case of xs and ys, we allocate contiguous storage (i.e., arrays).

- Stack allocation itself is contiguous. That is, the memory locations referenced by x, xs, y, and ys all follow after one another. Contrast this with heap allocations which are not necessarily contiguous in memory, so we draw them as such.
- Names are not embedded in memory. They are simply aliases for us to reference storage locations in memory. Strictly speaking, they are aliases for the memory addresses of said storage locations.
- By default, memory that is allocated comes uninitialized which we note with '?'. Really, the values of uninitialized memory correspond to whatever the bits that memory was previously set to, e.g., the value of the previous inhabitant of that memory block.

Here is an example of manipulating the declarations we've made. Before looking at the corresponding memory diagram, start with the previous diagram and trace through the code, updating the diagram for each instruction you "execute". This is generally how you should go about the memory diagram problems coming up.

```
x = 0;
for (x = 0; x < 2; x++) {
    ys[x] = x;
}
ys = ys + 2;
*y = 1;
y = &(xs[1]);
*y = 9;
```

Here is the final memory diagram after the above code executes.



With this in mind, here are a pair of problems to tackle. For each of the commented points in the code snippets below (e.g., /* 1 */), draw a memory diagram representing the current state of memory at that point in the program. You may assume any code not in a function is in main.

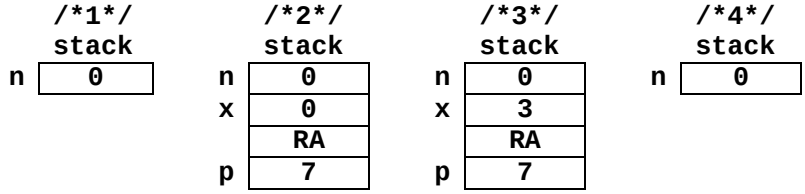
```
/** Problem #1 **/
int x = 0;
int y = 1;
int z = 2;
int *px = &x;
int *py = &y;
/* 1 */
z = *px;
*px = *py;
*py = z;
px = &z;
py = &(*px);
/* 2 */
```

```
/** Problem #2 **/
int i = 0, j = 0;
int *p = NULL;
int arr[5] = {3, 1, 0, 5, 4}
int *parr[5];
for (i = 0; i < 5; i++) {
    parr[i] = arr + i;
}
/* 1 */
for (i = 0; i < 5; i++) {
    p = arr + i;
    for (j = i; j < 5; j++) {
        if (arr[j] < *p) {
            p = arr + j;
        }
    }
    parr[i] = p;
    /* 2 (only after the first iteration, i = 0 */
}
/* 3 */
```

When we deal with functions, we need to understand how the caller of a function passes its arguments to the function in question, or callee. By default, these arguments are copied to the callee. This style of parameter passing is called call-by-value.

```
void f(int x) { int p = 7; /* 2 */ x = 3; /* 3 */ }
...
int n = 0;
/* 1 */
f(n);
/* 4 */
```

And the corresponding memory diagrams for points 1, 2, 3, and 4 in the program.

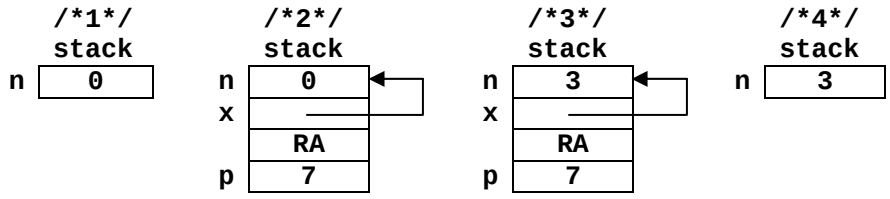


The symbol RA on the stack denotes the saved return address of the function call of f. That is, it is the location of the next instruction to execute after f returns. The particular value itself is irrelevant for our purposes, so we just denote it as RA.

Alternatively, we may want the callee to modify the values passed to it in a way that the caller can observe. This is called call-by-reference. In C, we explicitly accomplish this by using pointers.

```
void f(int *x) { int p = 7; /* 2 */ *x = 3; /* 3 */ }
...
int n = 0;
/* 1 */
f(&n);
/* 4 */
```

And the corresponding memory diagrams for the given points.



Here is a more complicated problem for you to work through that uses call-by-reference and call-by-value. Don't be daunted by the mutual recursion; be systematic in how you trace through the code.

```
/** Problem 3 **/
/* Forward declarations: to support the mutual definitions of f and g */
int f(int n, int *p);
int g(int n, int *p);

int f(int n, int *p) {
    int r = 0;
    if (n <= 0) { return -1; /* 3 (if we ever enter this branch) */}
    *p = *p * n;
    /* 2 (only the second time f is called) */
    return g(n-3, p);
}

int g(int n, int *p) {
    if (n <= 0) { return 7; /* 3 (if we ever enter this branch) */}
    *p = *p - n;
    return f(n - 2, p);
}

int x = 9, r = 0;
int *px = (int*)malloc(sizeof(int));
*px = 1;
/* 1 */
r = g(x, px);
/* 4 */
```

Now that we have the basics down, let's put that knowledge to the test by using diagrams to predict when our code will produce memory errors. There are four common kinds of memory errors:

- Memory leak: when we lose access to memory allocated in the heap but forget to free it.
- Buffer overflow: when we write over the end of a chunk of memory, usually a buffer allocated on the stack.
- Dangling pointer: when we try to access or write to a pointer whose value has been "freed", e.g., a pointer to malloc'ed memory that was already free'ed or a pointer to memory allocated on the stack that was reclaimed because its enclosing function returned.
- Double free: when we try to free malloc'ed memory that has already been free'ed.

For the code snippet below, give the memory diagrams at each of the following points. Also for each snippet, state what memory errors the program contains and why (e.g., for Problem #3 in this handout, "there is a memory leak because we did not free the memory referenced by px").

```
/** Problem #4 **/
int *fib5()
{
    int arr[5] = {0, 1};  // the remaining elements are 0
    int i = 2;
    for (; i < 5; i++) {
        arr[i] = arr[i-1] + arr[i-2];
    }
    /* 1 */
    return arr;
}

int * fibs = fib5();
/* 2 */
```

Deliverables

You will need to submit printed copies of the following diagrams:

- Problem #1, part 1 and 2
- Problem #2, part 1, 2, and 3
- Problem #3, part 1, 2, 3, and 4
- Problem #4, part 1 and 2

Your understanding of memory allocation will be tested in class, using similar exercises.