## Understanding the C memory model

One of the identifiers of a good programmer is that they will have a strong mental model of how the language handles memory. To help, here is an explanation with some examples about how the "C" programming language handles memory.

# Memory diagrams

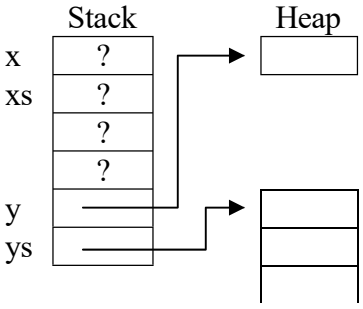
First, let's take a look at how instructions [lines of code] we write modify memory in the computer. These memory diagrams show what memory looks like after we execute our programs. In addition to being a good academic exercise, memory diagrams are a great tool to help you understand complex code. If you're ever stuck trying to debug a memory leak, a segmentation fault, or some otherwise gnarly piece of "C" code, the best thing to do is find a whiteboard and draw a diagram of what's going on.

Remember that the memory of a 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 function calls like '***malloc()***']. Thus whenever we declare a local variable, we are actually allocating space on the stack and letting the compiler know what name we will use to refer to that memory location in the stack area of our program. Conversely, when we call '***malloc()***', we are actually allocating space on the heap and are receiving a pointer to that memory as the value returned from the call to the function, which we are then storing in a variable

Remember also that when we make a function call to '***malloc()***', we must CAST the return value to the appropriate data type for what will be stored in that memory. What is a 'cast' you ask? You will see in the following code:

```
int x;
int xs[3];
int *y = (int *)malloc(sizeof(int));  // the "(int *)" is the 'cast'
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 mentally "execute" as you step through. This is generally how you should go about the memory diagram problems you encounter.

```
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.