Dynamic Memory and Pointers

Pointers

Pointers in C are variables that store the memory address of some other variable or data. They have an associated type of what kind of value they reference.

Pointers are one of the most difficult concepts in the C language. However, mastering pointers allows you do have a deeper understanding of what actually happens when your program runs. Languages like Java do not have pointers, but they do use pointers "behind the scenes". Learning pointers in C can also help you understand what's going on in a Java program.

Declaring

The type of a pointer variable is:

```
type*
```

where `type` is the type of data this pointer will reference. For example:

```
int* intPtr;
```

`intPtr` can hold the address of an `int` variable. Another:

```
char* charPtr;
```

`charPtr` can hold the address of a `char` variable.

When you are declaring pointers, the `*` can go any where between the type and the variable name. For example, all of the following are acceptable:

```
int* intPtr;
int * intPtr;
int*intPtr;
int *intPtr;
```

Variables in C are not automatically initialized – and this includes pointers. After declaring a pointer, it holds some garbage value that was left in that spot in memory.

& Operator (Address-Of)

The & operator returns the memory address of a variable. For example, if we have:

```
int x = 4;
```

And the `x` variable is stored at spot `1714` in memory (every variable is given a certain spot in memory, and these spots have an associated number), then if we do:

```
    &x
```

this will give us the `1714` spot.

The address-of operator isn't very useful unless we're using pointers. Since pointers are supposed to hold memory addresses, we can initialize them to be the address of some other variable.

So, suppose we have the following variable declarations:

```
    int x = 4;        //x is given spot  1714 in memory
    int *xPtr;
```

We can make the `xPtr` variable reference `x`:

```
    xPtr = &x;        //Now xPtr "points to" x – it holds address 1714
```

Notice that when we DECLARE a pointer, we include the *. However, when we INITIALIZE the pointer, we don't include the *.

** Operator (Dereferencing)*
The `*` operator is an operator specifically for pointer variables. It returns the value of what is being pointed at. For example, if we have:

```
    int x = 4;        //x is given spot  1714 in memory
    int *xPtr;
    xPtr = &x;
```

Then saying `*xPtr` gets us the value pointed to by `xPtr`. `xPtr` holds memory address `1714`, so if I say `*xPtr`, then I get the value stored in spot `1714` – which is a `4`. I can also use this operator to modify the value at that spot in memory. For example:

```
    *xPtr = 6;
```

Now the value at spot `1714` is a `6`. The `x` variable is stored in spot `1714`, so now `x` has the value `6`.

Example
The following example illustrates how pointers work:

```
    int i;     //i gets a memory location, say 3245, and has some random value
```

```
int *ip;    //ip has some random address
i = 36;     //i has the value 36
*ip = 72;   //Most likely, causes a segmentation fault
ip = & i;   //ip references memory address 3245
*ip = 72;   //Memory address 3245 has value 72 (so i = 72)
```

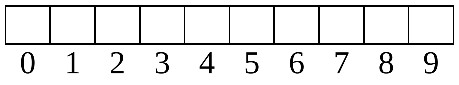
The reason "`*ip = 72`" will cause problems is that `ip` currently holds some random memory address, since it has not been initialized. When we say `*ip`, we're trying to access the memory at that random spot. This is most likely not the program's memory, so we will get a segmentation fault when we try to change it. (The other possibility is that we could end up overwriting one of the other program variables.)

Pointers v. Arrays

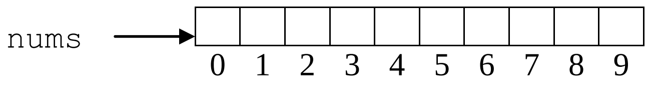
Arrays and pointers have a lot in common. When we do:

```
int nums[10];
```

Then we get a spot in memory that looks like this:



But what is `nums`? It is actually a *constant pointer* to the first spot in the array, `nums[0]`. So really, the picture looks like this:



So, `&nums[0]` (the address of the first element in the array) is the same thing as `nums`.

Pointer Notation

Because pointers and arrays are essentially the same thing (except array addresses cannot be changed), we can also access elements in an array by treating it as a pointer. In the above example, `nums` is a pointer to the first spot in the array. Space for arrays is reserved contiguously, so the second element in the array is physically next to the first element. This means that I can say:

```
nums+1
```

to get the memory address of the second element in the array.

Note: an integer uses 4 bytes of space. However, you don't say "`nums+4`" to move to the next integer. This is because pointers have a particular type associated with them – like an `int` – and the compiler will automatically move over the space of an `int` when you say `+1`.

Suppose now that you want to initialize the value at index `4` in the `nums` array to `7`. You could say:

```
nums[4] = 7;
```

However, you could do the same thing by treating `nums` as a pointer. You can get the address of the array element at index `4` by saying:

```
nums+4
```

This is a pointer, so if we want to change the contents of that location to `7`, we need to dereference it:

```
*(nums+4) = 7;
```

Example

Recall that an array is a constant pointer to a block of reserved memory. This means that we can change the values stored in the array, but we can't change the array itself (make it reference another piece of memory). Consider the following statements – which are legal?

```
int a[10];        //OK – a points to a block of 10 ints in memory
a++;              //NO – The address of an array can't change
int *xp = a;      //OK – Now xp also points to the beginning of the array
a = xp;           //NO – The address of an array can't change

int b[5];         //OK – b points to a block of 5 ints in memory
int *bp = b;      //OK – Now b also points to the beginning of the array
xp++;             //OK – Now b points to the second element in the array
*xp = 14;         //OK – The second element in the array is set to 14 (b[1] = 14)
```

Iteration with Pointers

This is how we have initialized array elements in the past:

```
int i;
int nums[10];
for (i = 0; i < 10; i++) {
     nums[i] = 0;
}
```

However, now that we can treat arrays like pointers, there is a different way to initialize array elements:

```
int *ip;
int nums[10];
for (ip = nums; ip < nums+10; ip++) {
     *ip = 0;
}
```

Here, `ip` is a pointer that starts by pointing to the first element in the array. We loop while the value of `ip` (the memory address) is less than `nums+10` – which is the address of the last element in the array. Each time, `ip++` advances `ip` to point at the next element in the array. Inside the loop, we dereference `ip` to get the current array element, and set that element to 0.

Pointers to Pointers

Just like a variable can be a pointer, we can also declare pointers to pointers. (We can take it even further than that, but it starts to get pretty confusing!) You can denote the "level" of the pointer by how many *'s you use in the declaration.

Here's an example of using pointers to pointers:

```
int i;              //declares the int i
int *ip;            //declares the int pointer ip
int **ipp;          //declares a pointer to a pointer to an int, ipp

i = 36;             //gives i the value 36
ip = &i;            //now ip points to i
*ip = 72;           // dereferences ip to get i, and sets it to 72 (now i=72)

ipp = &ip;          //ipp points to ip, which points to i
**ipp = 24;         //dereferences ipp to get ip, then dereferences again to get i,
                    //and sets it to 24 (now i = 24)
```

Call-by-Reference

C functions are naturally call-by-value, which means that we don't pass variables themselves – we pass their value. So, if we modify one of the parameters in our function, it does not modify the original variable passed to the function. Consider the following example:

```
//This example doesn't work!
void swap(int a, int b) {
     int temp = a;
     a = b;
     b = temp;
}
```

```
int x = 3;
int y = 4;
swap(x, y);
```

This code fragment is supposed to swap the values in `x` and `y`, so that `x = 4` and `y = 3`. However, when we call swap, only the VALUES `3` and `4` are passed – not `x` and `y` themselves. The values `3` and `4` get bound to the function parameters `a` and `b`. By the end of the function, we do have that `a = 4` and `b = 3`. However, `x` and `y` don't change because they are completely different from `a` and `b`.

If we do want to change `x` and `y`, we need to pass in the address of `x` and the address of `y`. Then, we can update the values at those memory locations. Here is our revised swap function:

```
//Take two memory addresses (pointers)
void swap(int *a, int *b) {
     int temp = *a; //Store the value pointed to by  a
     *a = *b;       //Update the contents of a to be the contents of b
     *b = temp;     // Update the contents of a to be temp
}
```

Now, when we call swap, we will need to pass the memory address of the variables we want to swap. This means we need to use the `&` operator:

```
int x = 3;
int y = 4;
swap(&x, &y);
```

sizeof

The `sizeof` function in C returns the number of bytes needed to store a specified type. It is needed for dynamic memory allocation because we need to know how many bytes we want to allocate.

Here is the prototype:

```
int sizeof(type)
```

where type is a defined type in C, like `char` or `int*`. Here are a few examples:

```
sizeof(int)     //evaluates to 4
sizeof(char)    //evaluates to 1
sizeof(double)  //evaluates to 8
sizeof(int*)    //evaluates to 4
sizeof(char*)   //evaluates to 4
```

Dynamic Memory

Currently, we can only declare arrays to be of a constant size (like 10). This is not always convenient – sometimes we want to make the size based on some user input. If we want to allocate a *dynamic* amount of space, we need to use C's dynamic memory functions. Each of these functions is in `<stdlib.h>`.

`malloc`

This function allocates a contiguous block of memory with the specifies size (number of bytes). It returns a *void pointer* to the block of memory. (This pointer will be automatically cast to the correct type when you store it.) Here is the prototype:

```
void* malloc(int numBytes);
```

For example, we could allocate an array like this:

```
int nums1[5];
```

Or we could do the same thing using `malloc`. If we use `malloc`, we need to specify the number of bytes to reserve. We want 5 ints, and each `int` takes up `sizeof(int)` bytes. So, the total needed is `5*sizeof(int)`:

```
//The result of malloc is automatically cast to an int*
int* nums2 = malloc(5*sizeof(int));
```

Now, we can treat `nums2` just like an array. For instance, if we wanted to initialize all elements in `nums2` to 0:

```
int i;
for (i = 0; i < 5; i++) {
    nums2[i] = 0;   //The compiler converts this to  *(nums2+i) = 0
}
```

Allocating arrays with `malloc` has several key difference from standard array allocation:

1) `malloc` can handle a variable for the desired size; a standard array cannot
2) The result of `malloc` is a pointer; the result of a standard array allocation is a *constant* pointer
3) `malloc` memory is allocated on the heap. If there is not enough space to do the allocation, `malloc` will return `NULL`. An array is allocated on the program stack – if there is not enough space, the program simply won't compile.

`calloc`

The `calloc` function is very similar to `malloc`. The only difference is that when arrays are allocated using `calloc`, all elements are automatically initialized to 0. Here is the prototype:

```
void* calloc(int numElems, int sizePerElem);
```

The prototype of `calloc` is also a little different than the one for `malloc`. It takes two arguments – the number of elements you want in the array, and the number of bytes needed for each elements. Like `malloc`, `calloc` returns a void pointer to the contiguous block of memory it allocated. This pointer will be automatically cast to the appropriate type when you store it.

Here's how to create an array of 10 ints, all initialized to 0:

```
int* nums = calloc(10, sizeof(int));
```

Now you can use `nums` just like an array. For example:

```
nums[0] = 4;
```

Like `malloc`, `calloc` will return `NULL` if there is not enough space to do the allocation. In both cases, it's a good idea to check if the pointer is `NULL` before you use it. For example:

```
int* nums = calloc(10, sizeof(int));
if (nums == NULL) {
     printf("Not enough space.\n");
}
else {
     //Use nums as usual
}
```

`realloc`

The `realloc` function allows you to easily expand and shrink the space allocated for an array. Here is the prototype:

```
void* realloc(void* origPtr, int newSize)
```

This function takes your original pointer and the desired new size in bytes. It looks for a contiguous block of memory with the desired size. If it can find one, it copies the contents of the old array into the new block of memory. Then it releases the space needed for the old array, and returns a `void` pointer to the new block of memory.

The `realloc` function doesn't always behave as you intend. Here are the possible return values of `realloc`:

- `NULL` (if not enough space is found)
- The original pointer (if there is enough space at that location)
- A new pointer to a different spot in memory

Suppose we allocate the `nums` array like this:

```
int* nums = malloc(10*sizeof(int));
```

Now we decide that we want `nums` to hold 15 elements instead of 10. Here's what we might try:

```
nums = realloc(nums, 15*sizeof(int));
```

Suppose that `realloc` could not find enough space to grant the request – so it returns `NULL`. This means that we assign `NULL` to our original `nums` pointer. Now `nums` does not reference the original array – in fact, nothing does. The original array is stuck in memory with no way to get at it – this is called a memory leak.

To fix this problem, assign a temporary pointer to the result of `realloc`. Then, if it's not `NULL`, reassign the original pointer. This keeps you from losing your array:

```
int *temp = realloc(nums, 15*sizeof(int));
if (temp != NULL) {
     nums = temp;
}
```

`free`

In Java, any memory that you're no longer using will be cleaned up by the garbage collector. C has no garbage collector, so you are in charge of releasing memory that you're done with. **If you never release any allocated memory, you will eventually run out of space.**

The C function that releases dynamic memory is called `free`. Here is the prototype:

```
void free(void* pointer)
```

Note that just because free takes a `void` pointer, it can take any type of pointer that has been dynamically allocated. Here's an example of using `free`:

```
int* nums = malloc(10*sizeof(int));
int i;
for (i = 0; i < 10; i++) {
     nums[i] = i;
}

//done using  nums
free(nums);
```

Multi-Dimensional Dynamic Arrays

We can also create multi-dimensional dynamic arrays using `malloc`. This section will focus on creating two-dimensional arrays. You can get more dimensions by adapting the following process:

1) Use `malloc` to create an array of pointers. Each pointer in the array will point to a row in the two-dimensional array. For example:

```
//Final array will have 3 rows
int **matrix = malloc(3*sizeof(int*));
```

2) Use `malloc` to allocate space for each row:

```
int i;
for (i = 0; i < 3; i++) {
    //Final array will have 4 columns
    matrix [i] = malloc(4*sizeof(int));
}
```

3) Now we can treat the pointer like a traditional two-dimensional array. For example, we could set every element to 0:

```
int j;
for (i = 0; i < 3; i++) {
    for (j = 0; j < 4; j++) {
        matrix [i][j] = 0;
    }
}
```

When we are done using a multi-dimensional array, we release the memory in reverse order of how we allocated it. So, first we release each row:

```
for (i = 0; i < 3; i++) {
    free(matrix [i]);
}
```

And then we release the top-level array of pointers:

```
free(matrix);
```