Dynamically Allocated Memory in C

All throughout the C course (COP 3223), all examples of variable declarations were statically allocated memory. The word "static" means "not changing" while the word "dynamic" means "changeable," roughly speaking.

In regards to memory, what this means is as follows:

(1) static – the memory requirements are known at compile-time. Namely, after a program compiles, we can perfectly predict how much memory will be needed and when for statically allocated variables. The input the program may receive on different executions of the code will NOT affect how much memory is allocated. One serious consequence of this is that any statically allocated variable can only have its memory reserved while the function within which it was declared is running. For example, if you declare an int x in function A, if function A has completed, no memory is reserved to store x anymore.

(2) dynamic – the memory requirements are NOT known (for sure) at compile-time. It may be the case that on different executions of the program, different amounts of memory are allocated; thus, the input may affect memory allocation.

If you want to allocate memory in one function, and have that memory available after the function is completed, you HAVE to allocate memory dynamically in that function!!!

Secondly, since dynamically allocated memory isn't "freed" automatically at the end of the function within which it's declared, this shifts the responsibility of freeing the memory to the user. This can be done with the free function.

malloc, calloc functions

Here are the formal descriptions of the two functions we will typically use to allocate memory dynamically:

```
// Allocates unused space for an
object    // whose size in bytes is
specified by size // and whose value is
unspecified, and     // returns a pointer to
the beginning of the // memory allocated. If
the memory can't be // found, NULL is
returned.

void *malloc(size_t size);


// Allocates an array of size nelem with
// each element of size elsize, and returns
// a pointer to the beginning of the memory
// allocated. The space shall be initialized
// to all bits 0. If the memory can't be
// found, NULL is returned.

void *calloc(size_t nelem, size_t elsize);
```

Although these specifications seem confusing, they basically say that you need to tell the function how many bytes to allocate (how you specify this to the two functions is different) and then, if the function successfully finds this memory, a pointer to the beginning of the block of memory is returned. If unsuccessful, NULL is returned.

Dynamically Allocated Arrays

Sometimes you won't know how big an array you will need for a program until run-time. In these cases, you can dynamically allocated space for an array using a pointer. Consider the following program that reads from a file of numbers. We will assume that the first integer in the file stores how many integers are in the rest of the file.

The program on the following page only reads in all the values into the dynamically allocated array and then print these values out in reverse order.

Note that actual parameter passed to the malloc function. We must specify the total number of bytes we need for the array. This number is the product of the number of array elements and the size (in bytes) of each array element.

It should be fairly easy to see how we can change the code below to utilize calloc instead of malloc. In this particular example, since there is no need to initialize the whole block of memory to 0, there's no obvious advantage to using calloc. But, when you want to initialize all the memory locations to 0, it makes sense to use calloc, since this function takes care of that task.

```
#include <stdio.h>
int main() {

  int *p, size, i;
  FILE *fp;

  // Open the input file.
  fp = fopen("input.txt", "r");

  // Read in all the numbers into the array.
  fscanf(fp, "%d", &size);
  p = (int *)malloc(size*sizeof(int));
  for (i = 0; i<size; i++)
    fscanf(fp, "%d", &p[i]);

  // Print out the array elements backwards.
  for (i = size-1; i>=0; i++)
    printf("%d\n", p[i]);

  // Close the file and free memory.
  free(p);
  fclose(fp);
  return 0;
}
```

A couple notes about pointers and dynamic arrays

The return type of malloc is void*. This means that the return type for malloc must be casted to the type of the pointer that will be pointing to the allocated memory.

The reason for this is so that malloc can be used to allocate memory for all types of structures. If malloc returned an int *, then we couldn't use it to allocate space for a character array, for example.

Instead, all malloc does is return a memory location w/o any specification as to what is going to be stored in that memory.

Thus, the programmer should (the book says it isn't necessary, but the gcc compiler will give you a warning if you don't do this) cast the return value from malloc to the type they want.

All this cast really does is specify the memory to be broken into "chunks" in a particular way. (Once we know what we are pointing to, we know how many contiguous memory locations stores a piece of data of the array.)

Although I haven't specified above, it is possible for malloc to fail to find the necessary memory in the heap. If this occurs, malloc returns NULL. A good programming practice is to check for this after each malloc call.

I've never had a malloc call fail. But, the potential is there if you do NOT free memory when possible. Once you are done using a dynamic data structure, use the free function to free that memory so that it can be used for other purposes.

realloc

Sometimes, what may occur is that an array gets filled, but you want to "extend" it because more elements must be stored. Based on the method of dynamic memory allocation already discussed, this could be solved in the following manner:

1) Allocate new memory larger than the old memory.
2) Copy over all the values from the old memory to the new.
3) Free the old memory.
4) Now we can add new values to the new memory.

We can avoid this extra work through a function that does it for us, realloc.

```
void *realloc(void *ptr, size_t size);
```

Here's the description from the IEEE standards web page about this function:

The *realloc*() function shall change the size of the memory object pointed to by *ptr* to the size specified by *size*. The contents of the object shall remain unchanged up to the lesser of the new and old sizes. If the new size of the memory object would require movement of the object, the space for the previous instantiation of the object is freed. If the new size is larger, the contents of the newly allocated portion of the object are unspecified. If *size* is 0 and *ptr* is not a null pointer, the object pointed to is freed. If the space cannot be allocated, the object shall remain unchanged.

Roughly speaking, in most cases, realloc, when called appropriately, simply extends the size of an existing array. The description basically describes the various contingencies of what occurs in atypical situations.

Short Example of realloc

```
#include <stdio.h>
#include <time.h>

#define EXTRA 10

int main() {

    int numVals;

    srand(time(0));

    printf("How many numbers do you want to pick?\n");
    scanf("%d", &numVals);
    int* values = (int*)malloc(numVals*sizeof(int));

    int i;
    for (i=0; i<numVals; i++)
        values[i] = rand()%100;

    for (i=0; i<numVals; i++)
        printf("%d ", values[i]);
    printf("\n");
```

```
values = (int*)realloc(values,(numVals+EXTRA)*sizeof(int));

    for (i=0; i<EXTRA; i++)
        values[i+numVals] = rand()%100;
    numVals += EXTRA;

    for (i=0; i<numVals; i++)
        printf("%d ", values[i]);
    printf("\n");

    free(values);
    return 0;
}
```

How to create a dynamically allocated array in a function

The key idea is very similar to doing this task in main, but you have to return a pointer to the array created.

Here is an example of how we can take the first example and separate out the array allocation into a function:

```
int* readArray(FILE* fp, int size) {

  int* p = (int *)malloc(size*sizeof(int));
  for (i = 0; i<size; i++)
    fscanf(fp, "%d", &p[i]);

  return p;
}
```

Here is how we can call this function from main (or any other function):

```
fp = fopen("input.txt", "r");
fscanf(fp, "%d", &size);
int* numbers = readArray(fp, size);
```

Picture-wise, the array gets created while readArray is running, a pointer to the beginning of the array is returned, and numbers (from main), is set to point to this newly allocated memory. To free this memory, do the following in main later:

```
free(numbers);
```

How to create a dynamically allocated structure from a function

We will create the following struct and return a pointer to it from a function:

```
struct integer* {
    int* digits;
    int size;
};
```

The following function creates a random struct integer dynamically and returns a pointer to it:

```
struct integer* createRandBigInt(int numDigits) {
    struct integer* temp;
    temp = (struct integer*)malloc(sizeof(struct integer));

    temp->digits = (int*)malloc(numDigits*sizeof(int));
    temp->size = numDigits;

    temp->digits[numDigits-1] = 1 + rand()%9;

    int i;
    for (i=0; i<numDigits-1; i++)
        temp->digits[i] = rand()%10;

    return temp;
}
```

Notice the that there are two mallocs. The first allocates space for the struct itself. This space is just enough for one integer pointer (small amount of space) and one integer.

The second malloc allocates space for the array (this is potentially a large amount of space).

To properly free the memory from this whole structure, imagine we had a variable p of type struct integer* pointing to a struct integer. These two lines would be necessary to free all the memory for the structure:

```
free(p->digits);
free(p);
```

How to create a dynamically allocated array of structs from a function

This works very, very similar to allocating an int array dynamically. The only difference is that instead of using int, you use the struct, appropriately in those locations. We will be using the following struct in this example:

```
struct point {
  int x;
  int y;
};
```

Here is a function that creates an array of struct point dynamically (filled with random points) and returns a pointer to the front of the array:

```
struct point* createRandPoints(int size, int maxVal)  {

     struct point* temp;
     temp = (struct point*)malloc(size*sizeof(struct point));
     int i;

     for (i=0; i<size; i++) {
         temp[i].x = 1 + rand()%maxVal;
         temp[i].y = 1 + rand()%maxVal;
     }

     return temp;
```

```
}
```

Notice that we only have one malloc, for the array itself. This allocates all of the space we need in one step.

Once the space is allocated, we treat each array location as an individual struct, using . to access its components. To free this array, if we had a pointer my_pts pointing to the array, do the following: `free(my_pts);`

How to create a dynamically allocated array of pointers to structs

Effectively, we will accomplish the same general task as the previous example, but this time, our array elements will only store a POINTER to the struct instead of the struct itself.

Here is the function:

```
struct point** createRandPoints(int size, int maxVal)  {


     struct point** temp;
     temp = (struct point**)malloc(size*sizeof(struct point*));
     int i;

     for (i=0; i<size; i++) {

         temp[i] = (struct point*)malloc(sizeof(struct point));

         temp[i]->x = 1 + rand()%maxVal;
         temp[i]->y = 1 + rand()%maxVal;
     }

     return temp;
}
```

First, notice the double pointer – the first is for the array, the second is for the contents of each array element. (Note: This same declaration could be used for a 2-D array…)

Our first allocation is for the array. From there, for each array element, we must allocate space for each individual struct.

Finally, notice the use of the -> since temp[i] is a pointer.

We must free everything in the same fashion (each element first, then the array). Here is some code to show this process:

```
struct point** my_pts = createRandPoints(100, 1000);


// Do something with my_pts.


// Frees each individual point pointer.
int i;
for (i=0; i<100; i++)
    free(my_pts->temp[i]);


// Frees the memory that stores the main array.
free(my_pts);
```

How to create a dynamically allocate a two dimensional array

Here is some code that allocates a two dimensional array of integers (or rather, an array of an array of integers) with n rows and m columns. Assume that n and m are integer variables that have already been given meaningful values.

```
int** array = (int**)malloc(sizeof(int*)*n);
int i;
for (i=0; i<n; i++)
    array[i] = (int*)malloc(sizeof(int)*m);
```

Notice that the first cast is to int and second set of casts are to int*. These are the types of array and array[i], respectively. Also note that we need sizeof(int*) in the first malloc, since array will be an array of int*, but we need sizeof(int) for the second set of mallocs, since each array[i] will be an array of integers.**

Now, here is how we free this memory:

```
for (i=0; i<n; i++)
    free(array[i]);
free(array);
```