Memory

Types

As you probably remember from your Intro to C class there was a few ways to represent you memory. You have all covered certain basic types,

- char
- int
- double
- float
- long
- byte
- short

Later you probably covered how to create your own data type. These custom data types were structs

```
struct newType {
   int number1;
   int number2;
   char character;
};
```

The last thing typically covered is pointers in the form of arrays.

```
int a[3];
a[0] = 5;
a[1] = 6;
a[2] = 1;
```

These methods lack some potentially desirable functionality,

1. Ability to change size
2. Ability to be used after the function that instantiated the memory is exited

This document will help you learn to do such memory management.

The Memory Stack

Certain variables are not accessible after a function is exited. Consider the following example,

```
void fun1();

int main() {
   fun1();
}

void fun1() {
   int tmp;
   scanf(“%d”, &tmp);
   printf(“%d\n”, tmp);
}
```

So if we tried the following what might we expect?

```
int *fun1();

int main() {
   int *ptr;

   ptr = fun1();
   printf(“%d\n”, * ptr);
}

int *fun1() {
   int tmp;
   scanf(“%d”, &tmp);
   printf(“%d\n”, tmp);
   return &tmp;
}
```

The above example might work, but ***it is very wrong***. With a few minor changes the program behavior gets a little bit strange. Consider the following example now,

```
int *fun1();

int main() {
   int *ptr1;
   int *ptr2;

   ptr1 = fun1();
   ptr2 = fun1();
   printf(“%d\n”, * ptr1);
   printf(“%d\n”, * ptr2);
}

int *fun1() {
   int tmp;
   scanf(“%d”, &tmp);
   printf(“%d\n”, tmp);
   return &tmp;
}
```

As it turns out the memory created this way is store on the STACK and after the code executes the portion of memory that is store there can be over written with other functions or variables. For this static memory you can think of it in the following way, when functions are called they have memory associated with them that are “lost” after the function returns.

Dynamic Vs Static

Typically when we refer to memory as static or variable we mean that the memory has the capability of changing its length. Some might say that dynamic memory refers to runtime memory, and this confusion is understandable but incorrect.

Motivation for dynamic allocation: Unknown. However, not in the sense that we don’t know why we would use dynamic memory, but because we might not necessarily know at compile time, or even at some (not necessarily all) parts in the program, the amount or the exact structure of our needed memory. Alternatively you might run on a platform with limited memory, and dynamic memory can allow you to efficiently manage the amount of memory created and used.

We can try to reassign memory with our tool, but you will see that we run into a problem.

```
int main()
{
   int i;
   int x[10];
   int y[11];
   for (i = 0; i < 10; i++)
      x[i] = 0;
   for (i = 0; i < 11; i++)
      y[i] = i;
   x = y;
   for (i = 0; i < 11; i++)
      printf("%d\n", x[i]);
}
```

The above code does not compile. Luckily there exists methods for us that can help us get the above code working as intended

The Memory Heap

The functions that we will use in this class is malloc, calloc, realloc, and free.

Malloc

Malloc allocates memory on a different portion of memory called the heap. This allocated memory is returned to the caller in the form of a void pointer (which means that it can point to many different memory types).

To specify how much memory is needed *malloc* takes in a single parameter representing the required number of bytes. This parameter type is size_t. (Note: As a common naming convention many variable types are suffixed with "_t" examples of which include int8_t and uint16_t). This size_t obviously should not be negative, so it is stored as an unsigned long int.

As an example suppose we want an integer array that is 11 size stored in some pointer x, but currently x has some memory already stored there. We could easily overwrite x by with the following line of code,

```
x = (int *) malloc(11 * sizeof(int));
```

The above line of code is doing a few things that should be addressed. The first thing that might confuse you is the "(int *)" in front of the *malloc* function. That snippet is used to tell our program that that section of code is supposed to be an int pointer. The next part is the sizeof(int). Since various platforms has different sizes for even the basic data structures, we should not use a hardcoded value for what we assume will be the size of an int. Instead our program asks the compiler/system "how many bytes are you going to use to store our data type?" By multiplying this result by 11 our program allows us to store 11 of the desired data type.

Two questions are created here:

1. What are the values for this variable after *malloc*?
2. What happened to the old memory?

Question 2 is a very important issue. That memory that was store at location x might be lost for the remainder of the programs execution (see the free section below). That means that the program won't reallocate memory where this value was stored. This issue is typically referred to as a memory leak. **If you have too much memory leakage, your program could seg fault or worse...** Tools such as valgrind are debuggers which can be used to help find such issues. Since valgrind is typically used in linux systems for windows users you can download Cygwin to run valgrind and find you problems.

The answer to *not as important* question 1 is that the newly allocated memory has not set to any value; whatever values were previously contained in that section of memory will be in the "new" array. These values are usually considered garbage, and you should initialize the array with some values or some other array.

Calloc

Due to the uninitialized memory generated via malloc (among other reasons) one popular alternative to malloc is *calloc*. *Calloc* is passed two parameters both are of type size_t. The first parameter is the number of items wished to allocate; the second parameter is the size of each item. *Calloc* like malloc returns a void pointer to the section of memory that can be used for our purposes. Unlike malloc, all bytes that are returned by *calloc* have their byte values set to 0. However, *calloc* can also leak memory if the pointers returned are not properly handled (see the free section below).

Realloc

Realloc as the name implies is a method for resizing a portion of allocated memory. *Realloc* like *calloc* takes in two parameters as well. But unlike *calloc* the first parameter *realloc* takes in is the pointer to the part of memory to be resized. The second parameter is the new size of the allocation. The memory in the old location will be handled properly assuming nothing horrible happened in the allocation.

All of the above functions are relatively safe as long as you don't allocate memory for something with zero bytes. Additionally if you don't have enough memory to allocate the section you may receive a "NULL" pointer.

Free

Free takes a pointer to memory and as long as it was allocated using malloc, calloc, or realloc. If a "NULL" pointer is passed to free nothing happens. If the pointer does not point to memory created using malloc, calloc, or realloc or the pointer has already had its memory freed, bad things can happen.

As a precaution you should set your pointers to null after freeing the associated memory.

```
int * ptr;
ptr = (int *) malloc(n * sizeof(int));
.
.
.
free(p);
ptr = NULL;
```

Ways to use Heap memory

Creating an array was shown earlier, but more uses exist. For creating memory to a singular basic data type such as int we can use malloc. As an example

```
int * x;
x = (int *) malloc(sizeof(int));
*x = 7;
printf("%d\n", x);
```

One struct

Both of these functions allow us to store even structs on heap, which can allow for structs to be used after a function finishes. When passing other variables to functions such that they could be updated we pass their pointers. If we have the pointer to a struct and we wish to access its member, we would have to dereference the pointer then access its member. A shortcut exists for this in c we can use "->" (e.g. p->x is the same as (*p).x).

Allocating memory for pointer to just a singular struct works the same way as allocating memory for basic data types. For an example refer to the following segement of code,

```
point_2d_t * p;
p = (point_2d_t *) malloc(sizeof(point_2d_t));
(*p).x = 7;
p->y = 4;
printf("%d %d\n", p->x, p[0].y);
```

Multiple structs

Arrays of structs can be used just as easily as arrays of the basic data types.

```
point_2d_t * p;
p = (point_2d_t *) malloc(sizeof(point_2d_t) * 2);
p[0].x = 7; p[0].y = 4;
p[1].x = 10; p[1].y = 11;
printf("%d %d\n", p[0].x, p[0].y);
printf("%d %d\n", p[1].x, p[1].y);
```

More than one dimension

These allocation functions can be used to create multidimensional arrays as well. To make things truly dynamic malloc should be used for each level. However, for each malloc/calloc/realloc used you need a corresponding free.

```
p = (float ***) malloc(sizeof(float *) * d1);
if (p == NULL)
{
    // Clean up
    return EXIT_FAILURE;
}
for (i = 0; i < d1; i++)
{
    p[i] = (float **) malloc(sizeof(float *) * d2);
    if (p == NULL)
    {
        // Clean up
        return EXIT_FAILURE;
    }
    for (j = 0; j < d2; j++)
    {
        p[i][j] = (int *) malloc(sizeof(float) * d3);
        if (p[i][j] == NULL)
        {
            // Clean up
            Return EXIT_FAILURE;
        }
    }
}
```

More Efficient Unbounded Arrays

Suppose we want to a structure that stores and array of value, but we want to append an arbitrary amount of items to the end. Statically sized arrays would not be able to meet this constraint. You could hope that with a significantly sized array you would not need to worry about running out of memory, but using dynamically allocated memory we can meet these conditions (in theory). We could store the maximum number of element a dynamic array can store, and when we need more room, we allocate additional memory and use that. However, how much additional memory has an effect on the efficiency of our unbounded array.

Consider the case where the array is allocated with exactly one extra value. If we used malloc each time all of the values need to be copied over which mean each insert past the default max size requires n operations, where n is the size of the array. A discussion on meaning of this will be in a later lecture. Instead we could allocate large portions imagine if we allocated ten extra slots each time we hit the max. Then only one out of ten inserts incurs an extra n operations. This is a ten times speed up. If we use an extra 100 or 1,000 as our gap we can achieve even better boosts.

How large of a gap should we go? If we use something like 1,000,000 is there any problems? Maybe. What if I need 100 of these array an each one will have no more than 100 values? It might have been better just to use a statically size 2d array. In this extreme case we have only .01% efficiency of memory, which is bad. Realistically I can understand 10% efficiency, but 90% is much more desirable. Let's split the difference and make a method that allows for 50% memory efficiency. This means that the array should be no more than twice the size of the number of values used. With this in mind when resizing the array we will make the array be twice as large as the previous maximum. ***I recommend coding this on your own as practice.***