Dynamic Data Structures

CSCI 112: Programming in C
It’s all about **flexibility**

- In the programs we’ve made so far, the compiler knows at **compile time** exactly how much memory to allocate for each variable or array.
- These variables are only “alive” during the specific block of code in which they were created.
- This isn’t always efficient or flexible: what if we don’t know the size of a list at compile time? What if we want more **control** of when a variable is created and deleted?
- Enter **dynamic memory allocation**, a C feature which gives us explicit control over memory management.
What are the drawbacks of letting the compiler do the allocation?

```c
int main(void) {
    char arr[400];
}

void my_function() {
    int x;
    ...
}
```
What are the drawbacks of letting the compiler do the allocation?

```c
int main(void) {
    char arr[400];
}

void my_function() {
    int x;
    ...
}
```

• **arr** will take up exactly 400 bytes, no matter how many elements we use. What if we need more than 400? What if we only actually use 300?

• **x** is no longer “alive” after **my_function** ends.
Dynamic Memory Allocation in a nutshell

• Dynamic memory allocation lets us obtain more memory when we need it, and to release memory back to the system when we don’t need it.

• It lets us do this while the program is running…we can base memory needs off of user input, for example.
First things first: the **heap** and the **stack**

- **The stack** is where C allocates memory for variables it creates automatically, within functions. (This is what we’ve done up until now.)
- When a function is called, the local variables for that function are allocated on the “top” of the stack, so the function can read and write to them.
- When the function terminates, those variables are **deallocated**. (Erased from the stack.)
First things first: the **heap** and the **stack**

- The **heap** is a more “stable” part of memory. Variables that are created here remain in existence for the duration of the program.
- Variables are not tied to a specific function like those in the stack are.
- **This is where dynamically allocated variables are stored.**
Make sure to include `stdlib.h` in order to use the following memory management functions!
Allocating space for a variable: `malloc`

- `malloc` allocates a specified amount of space on the heap, and returns a pointer to where that space starts.
- It takes one argument: an integer representing how many bytes to allocate.
- Let’s say we want to allocate space for an `int`:
  ```c
  malloc( sizeof(int) )
  ```
- The above call gives us back a `void pointer` (address) to the allocated block of memory.
- The `sizeof` function gives us exactly how many bytes we need for a given type.
Allocating space for a variable: `malloc`  

- We can cast this void pointer to a specific pointer type, allowing us to use the newly allocated space in our code:

```c
// Create some pointers
char *letp;
int *nump;

// Dynamically allocate memory blocks
// Assign the above pointers to point to those blocks
// We need to cast the void pointer returned by malloc
// to the right type of pointer.
letp = (char*) malloc( sizeof(char) );
nump = (int*) malloc( sizeof(int) );
```
Allocating space for a variable: `malloc`

- Here’s how this code affects memory: (Notice that the pointers are on the stack, since they were created in a function. Dynamically allocated blocks are on the heap.)
Allocating space for a struct: `malloc`

- It’s just as easy to dynamically allocate memory for a struct:

```c
typedef struct planet {
    int radius;
    char name[5];
    ...
} planet_t;

planet_t *p = (planet_t*) malloc( sizeof(planet_t) );
```
Allocating space for a struct: `malloc`

- Once again, here's what that looks like in memory. Notice that `malloc` creates enough space for all components of the planet structure.
Allocating space for an array: \texttt{calloc}

- \texttt{malloc} allocates a single block of memory for a built-in or user defined type (such as a struct).
- We can use \texttt{calloc} (contiguous allocation) to allocate an array of blocks.
- It takes two arguments—\texttt{number} of elements and \texttt{size} of each element.
- Besides allocating space, it also \textbf{initializes each element to 0}.
- This is the main difference between \texttt{malloc} and \texttt{calloc}. While \texttt{malloc} could allocate the same amount of space as a call to \texttt{calloc}, it would not initialize array elements to 0.
Allocating space for an array: `calloc`

- **In code:**

```c
int number_of_elements = 4;
int *int_array = (int*) calloc(number_of_elements, sizeof(int));
```

- **In memory:**

![Diagram showing stack and heap with a pointer to an allocated array on the heap]
Returning cells to the heap: **free**

- **free** let us “deallocate” a cell, or indicate that we no longer need it.
- The cell can then be allocated for another variable.
- It’s what the Garbage Collector does automatically in Java.
- To free a block of memory associated with a pointer \( x \), just do this: **free(x)**
Returning cells to the heap: `free`

- In this code, we dynamically allocate space for a double, assign it a value, make a copy of it, then free that space.

```c
double *xp, *xcopyp;

xp = (double *)malloc(sizeof (double));
*xp = 49.5;
xcopyp = xp;
free(xp);
```

- The cell containing 49.5 may be reallocated after the call to `free`. So we can’t use it anymore. Even though `xp` and `xcopyp` still exist as pointers, they cannot be used to reference the cell.