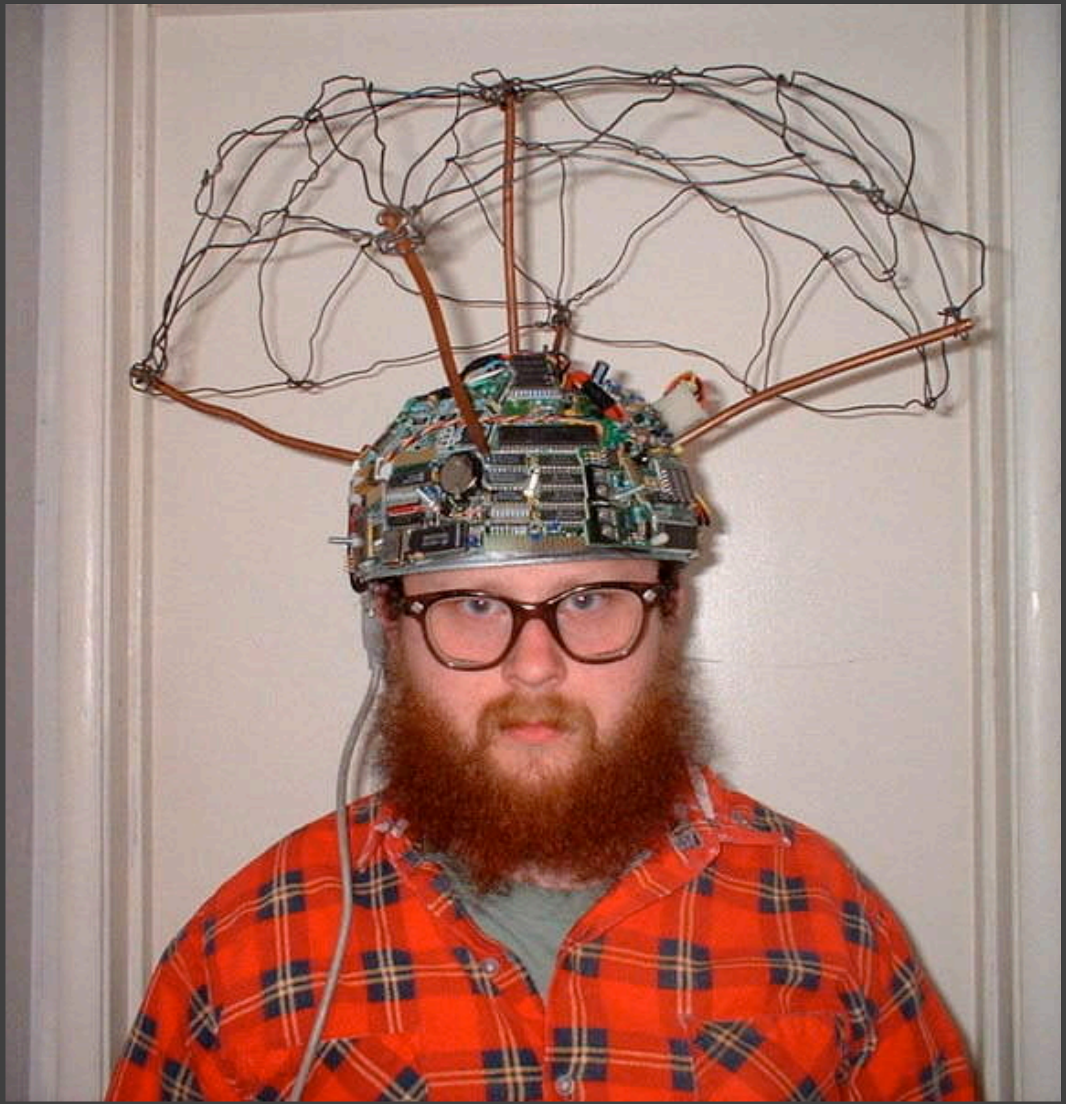
Dynamic Memory Allocation

- **Dynamic memory is memory that is allocated at run-time.**
- **It is allocated from a region of memory that we call *the heap*.**

What is **dynamic memory allocation?**

- The act of allocating memory for variables on the heap (or called free store) during program run-time.
- Quite different than compile time (static) allocation:
  - Compile time allocation (CTA) means memory for named variables is allocated by the compiler at compile time,
  - Dynamic memory allocation (DMA) means memory is allocated on-the-fly during run-time.
  - CTA requires the exact size and type of storage at compile time, DMA is calculated and allocates the exact memory it needs during run-time.



Why is dynamic memory allocation good?

- Stack space is limited.
- We sometimes want variables to last beyond the lifetime of its current scope:
  - Variables on the stack don't last beyond its current scope.
  - Variables dynamically allocated on the heap can be accessed beyond the current scope.
- We don't always know exactly how much memory is needed to run a program.
  - Solution: dynamically allocate memory when it's needed, however much is needed.
  - But how do we dynamically allocate memory?

Dynamic memory allocation

- There are three standard **C** library functions to allocate memory, all of which are defined under `stdlib.h`
  - `malloc()`, `calloc()`, and `realloc()`
- These three functions dynamically allocate memory on the heap and returns a pointer to the allocated memory
- Allocated memory must be freed using `free()`.
  - Not freeing allocated memory can lead to depletion of system resources.
  - This is called a *memory leak*.

The Heap

- A large region of unmanaged, anonymous memory.
- Only limitations are your computer’s physical limitations.
- Slower to read from/write to due to the need for pointers.
- Variables using heap memory can be accessed globally with access to the pointer.
  - A benefit of using pointers; much easier to pass around pointers for large data structures.
- Possible memory fragmentation can occur over time as blocks of memory are allocated and deallocated.

`malloc()`

- Defined as:
  - `void *malloc(size_t size)`
- Returns a pointer to `size` bytes of <u>uninitialized memory</u> allocated on the heap.
- The memory allocated by `malloc()` may contain junk data.
- What happens when `size == 0` is implementation defined (avoid doing this).
- Doesn’t check for overflow of `size` if it’s the result of an arithmetic operation, unlike `calloc()`.

Example of malloc()

```c
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>

//
// Allocates memory for an array of 10 ints.
// Sets each array index to the value of the index.
//
int main(void) {
  int *arr = malloc(10 * sizeof(int));
  assert(arr);

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

  return 0;
}
```

`calloc()`

"c" means that the allocated memory is cleared to zeroes.

- Defined as:
  - `void* calloc(size_t nmemb, size_t size)`
- `nmemb` denotes the number of objects and `size` the size of each object.
- Returns a pointer to `nmemb` × `size` bytes of allocated memory on the heap, in which each byte has been initialized to zero.
- Like `malloc()`, behavior when `nmemb` × `size` is zero is implementation defined.
- Generally slower than `malloc()`, but the tradeoff is that the contents of the allocated memory are known since it's zeroed out.

Example of calloc()

```c
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>

//
// Functions like previous malloc() example.
// Takes advantage of zeroed-out array to reach same result.
//
int main(void) {
  int *arr = calloc(10, sizeof(int));
  assert(arr);

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

  return 0;
}
```

Example of `malloc()`

```c
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>

//
// Allocates memory for an array of 10 ints.
// Sets each array index to the value of the index.
//
int main(void) {
  int *arr = malloc(10 * sizeof(int));
  assert(arr);

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

  return 0;
}
```

Example of `calloc()`

```c
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>

//
// Functions like previous malloc() example.
// Takes advantage of zeroed-out array to reach same result.
//
int main(void) {
  int *arr = calloc(10, sizeof(int));
  assert(arr);

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

  return 0;
}
```

`free(ptr)`

- If we allocate memory, we must also be able to deallocate (or free) memory.
- Another standard C library functions specifically for deallocating memory allocated by `malloc()`, `calloc()`, or `realloc()`.
- Defined as `void free(void *ptr)`
- Deallocates the memory space pointed to by `ptr`.
- Memory leaks occur if allocated memory isn't freed.
- Segmentation faults/core dumps can occur if a program tries to access (previously freed) memory locations that it isn't allowed to access.
- Pointers that have been freed should be set to `NULL` to mitigate use-after-free vulnerabilities.

"`ge_alloc()`" and "`ge_free()`"

- In my first position as a software engineer, I had to use special functions instead of standard `malloc()` and `free()`.

```
char *s = ge_alloc(10);
ge_free(&s);
```

- Our `ge_free()` function cleared `s` to NULL.
  - So using a stale pointer causes a segmentation fault instead of a malfunction.
- These functions also added "begin guards" and "end guards" before and after the allocated memory.
  - `ge_free()` would report if either of the guards had been overwritten.
  - Alerting us that there had been an out-of-bounds array access.