# L11 - The Heap

November 1, 2019

## 0.1 The Heap and Dynamically Allocated Memory

All the variables we have discussed are stored on the *stack*. Stack memory is highly ordered (specifically, it is last-in first-out); we must specify exactly how big an item is before we allocate it on stack (note that we always have to tell C how big our array is at initialization, or declare a type). Memory is provided to these variables automatically and it is released automatically when the program (or function) exits.

A C program also has access to a region of memory known as the *heap*. The heap is *not* used for variables; we cannot specify that a variable is stored on the heap. Heap memory is allocated *dynamically* - you must call a function to allocate memory on heap. The heap is also free on all limits of size - OS software generally imposes stack size limits.

We manage memory on the heap with functions `malloc`, `calloc`, `realloc` and `free`. Variables on the heap are accessed using pointers. (Note that these pointers are themselves stored on the stack.) Because we use pointers, we can store state in heap variables regardless of the scope where these variables are modified.

**tl;dr**

- the heap is most useful for
  - very large arrays and structs
  - variables, structs and arrays that must remain accessible to all scopes
  - arrays and structs that can change size (not possible on stack!)
- otherwise, use stack - it's easier and accesses faster

In [5]:
```
// allocate a block of memory on the heap to hold one int

#include <stdlib.h>
#include <stdio.h>

    int main(){

    int *p;

    p = malloc(sizeof(int));

    if ( p != NULL )
    {
```

```
            *p = 3;
        }
        else
        {
            abort(); // failed to allocate memory
        }

        printf("heapy p = %d", *p);

    }
```

```
heapy p = 3
```

#### 0.1.1 `malloc(size_t size)`

- allocates a block of memory of specified size, from the heap
  - the memory is *uninitialized*
- returns a pointer to the allocated block
  - if memory *cannot* be allocated, returns NULL (the null pointer value)

#### 0.1.2 `sizeof(type_name)`

- calculates the amount of memory required to hold one value of the specified type

#### 0.1.3 `assert( expression )`

- if `expression` is true, continue program execution as normal
  - if false; display a descriptive message and terminate the program
- must include `<assert.h>` to use

#### 0.1.4 `free(*ptr)`

- de-allocate memory from a variable stored on the heap
- memory is not automatically de-allocated, as on the stack
  - you must call `free()` to be able to re-use heap memory
  - failure to call `free()` on heap variables results in a *memory leak*

In [6]:
```
#include <stdlib.h>
#include <assert.h>

int main()
{
    int x = 1;
    int y = 2;

    assert(x == y); // lol nope
```

```
    return EXIT_SUCCESS;
}
```

```
tmp87f9prn1.out: /tmp/tmp4kfwiey6.c:10: main: Assertion `x == y' failed.
[C kernel] Executable exited with code -6
```

In [10]:
```
// allocate a block of memory on the heap to hold one int, assert-style

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

int main()
{

    int* p = malloc(sizeof(int));
    assert(p != NULL); // not the null pointer

    *p = 3;

    printf("heapy p = %d", *p);

}
```

```
heapy p = 3
```

In [23]:
```
// redo makepoint by allocating point_t on the heap
//%cflags:-lm

#include <stdlib.h>
#include <stdio.h>
#include <assert.h>
#include <math.h>

typedef struct {
    int x;
    int y;
    int z;
} point_t;

point_t *makepoint( int x, int y, int z )
{
    point_t *ptr = malloc(sizeof(point_t));
    assert(ptr != NULL);
    ptr -> x = x;
    ptr -> y = y;
    ptr -> z = z;
    return ptr;
```

```
        }

        int main()
        {
            point_t* pointy = makepoint( 1, 2, 3 );

            float norm = sqrt( pow(pointy->x, 2) + pow(pointy->y, 2) + pow(pointy->z, 2) );

            printf("norm pointy = %f", norm);

            free(pointy);

            return EXIT_SUCCESS;
        }
```

```
norm pointy = 3.741657
```