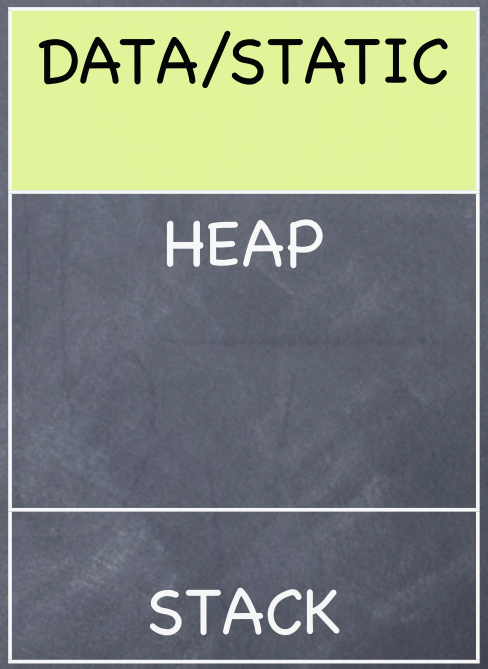
Heap Memory Management with malloc() and free()

Recall the Memory Model

- DATA/STATIC area: stores the compiled program in machine language form, global variables, and "static variables" (see text for explanation of static). Unlike the other memory areas it has an unchangeable/static size.

- STACK: holds a "stack frame" for each function call which stores function parameters, local variables, a program counter, and a return value (if any). Grows as functions are called (starting with main()), and shrinks as they return. Allocation and deallocation of stack memory is handled "automatically" when functions are called and exited. One a function returns, its stack frame is out-of-scope. A function should never return the address of a local variable.

- HEAP: is actively managed by the programmer via special function calls to allocate memory (malloc(), calloc()) and deallocate memory (free()). Java's rough equivalent to malloc() is the new operator. Java has not equivalent to free(); instead it has garbage collection.



Why Use Heap Memory?

- Arrays have limitations:
  - fixed size
  - the only way to free up its memory is to exit the function in which it was declared.
- The heap is flexible and convenient.
  - Ask for memory when you need it (with malloc() or calloc())
  - Ask for any # of bytes (e.g. 1, 200 * sizeof(int), 5000 * sizeof Packet)
  - Free up the memory when you're done (with free()).
- Suitable for data structures that grow, shrink (e.g. linked lists, trees)
- Heap memory persists across function calls (it's not local to a function)
  - NOT OK for a function to return the address of a local variable.
  - OK for a function to return a heap address

Easy as 1, 2, 3

1. Ask for memory with malloc() or calloc(). They're the same except:
   - malloc() uses one parameter for # bytes requested, calloc() uses two
   - calloc() initializes the memory it delivers (all to 0's); malloc() doesn't
2. Always check the return value of malloc()/calloc
   - If it's NULL, it failed (e.g. ran out of memory). Recover or quit.
3. Free memory with free()
   - Only valid argument: an address returned by malloc()/calloc() that hasn't already been freed

Ask for it, use it, free it

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

int main(){
  int *p = NULL;

  // Ask for memory to hold an int using malloc().
  // malloc() does not initialize memory.
  // It takes 1 argument (# bytes that you want)
  p = (int *)malloc(sizeof(int));
  if (p == NULL)
    return EXIT_FAILURE;
  *p = 12;
  free(p);


  // Ask for memory to hold an int using calloc().
  // calloc() initializes memory all to 0's.
  // It takes 2 arguments, # of items and size of an item
  p = (int *)calloc(1, sizeof(int));
  if (p == NULL)
    return EXIT_FAILURE;
  free(p);
  return EXIT_SUCCESS;
}
```

Memory for multiple structs

```
#include <stdio.h>
#include <stdlib.h>
#define MAX_DATA 1000000

typedef struct {
    char name;
    int id;
} Person;

int main(){
  Person *p = NULL;

  p = (Person *)malloc(MAX_DATA * sizeof(Person));
  if (p == NULL)
    return EXIT_FAILURE;
  int i = 0;
  for (i = 0; i < MAX_DATA; i++){
      p[i].name = '?';
      p[i].id = 0;
  }
  free(p);
  return EXIT_SUCCESS;
}
```

Variety of Operator Choices

```
#include <stdio.h>
#include <stdlib.h>
#define MAX_DATA 1000000
void initPeople(Person *, int num);

typedef struct {
    char name;
    int id;
} Person;

int main(){
  Person *p = NULL;
  p = (Person *)malloc(MAX_DATA * sizeof(Person));
  if (p == NULL)
    return EXIT_FAILURE;
  initPeople(p, MAX_DATA);
  free(p);
  return EXIT_SUCCESS;
}

// Write this function 3 different ways using *, ->, ., and []
void initPeople(Person *p, int num){


   for(          ;          ;       ){
                name = '?';
                id = 0;
   }
}
```

FYI function declarations

- Note that the return type of both malloc and calloc is void *
- size_t is defined to be some kind of int (compiler/processor dependent)

```
// function declarations

void *malloc(size_t size);
void *calloc(size_t nmemb, size_t size);
void  free(void *ptr);
```

What does static mean?

- In general static means "fixed memory" and "known at compile time". As opposed to dynamic which refers to run time. For example, the stack and heap are dynamic - they may grow and shrink during run time..
- In Java, static variables and methods belong to the class as a whole, not to individual (dynamically created) objects.
- In C, all static variables are stored in the DATA/STATIC memory area and initialized to 0 by default.
- In C, a static global variable or static method is private to the file. (Non-statics may be accessed by code in other files via the extern keyword.)
- In C, a static local variable retains its value from one function call to the next. Like "state" for a function. Precursor to objects.