More on Memory allocation in C

Overview of different memory segments

- As we discussed earlier there are three types of memory allocation for a C/C++ program's data:
  1. Stack-allocated memory or also know as automatic allocation.
  2. Static-allocated memory.
  3. Heap-allocated memory.
- The first and second types are allocated at the compilation time.
- The third type is allocated at the runtime.

Automatic allocation

- Automatic allocation is the type of the allocation that is managed on the stack.
- It is called automatic because it will allocated when a function is called and will be automatically de-allocated when function terminates.
- Their scope is the function scope and their lifetime is the function life time.
- Basically what we have mostly learned so far in the course:
  - Function arguments
  - Function local variable
  - In the following example x, y, z, and m are all automatically allocated on the stack when function fun is invoked, and de-allocated when function terminates:

```
void fun(int x, double y, struct something z)
{
    int m;
    …
    …
}
```

Static Allocation

Static Allocation

- Static allocation happens before main starts.
- Static variables are initialized to zero automatically. Or they can be initialized manually.
- Their lifetime is the life time of the program.
- Their scope depends on the type of the declaration.
- Major Examples of static allocation
  - String constants
  - Global variable
  - Local static variables
  - Others which won't be discussed in ENSF 337

Dynamic Allocation of Memory

Dynamic Allocation of Memory

- Lifetime of this type of allocation is either the entire program or up to the programmer to de-allocate them whenever it is no longer needed. In other words:
  - Can be allocated at the exact time that they are needed and then to be de-allocated at the exact time that they are not needed.
  - It's the programmers responsibility to de-allocate them (return memory back to the pool of available storage on the heap).
  - Heap-allocated memory can be also survived beyond the lifetime of a function call.

C Library Function `malloc`

- There are several library functions to be called to allocate memory at the runtime.
- One commonly used function is malloc. Here is the prototype of this function:

```
void *malloc(size_t n);
```

  - A void pointer denotes a generic pointer type.
  - Allocates a block of n bytes.
  - Returns a pointer to the block. If it fails to allocate memory returns NULL.
  - You can always check the returned value and handle the error, if possible.
- The following statement allocates 20-bytes of memory, and returns the address of the first byte.

```
char *p = malloc(20);
```

Using Library Function malloc

- The following simple C program allocate (**n * sizeof(int)**) bytes of memory, on the heap, and returns the address of the first byte of this block of memory.
- **Note:** In this example the program terminates when memory is unavailable. However, in some cases you may choose to handle the error in a different way, if possible.
- Draw a memory diagrams for points 1 and 2 and write the definition of function print.

```
#include <stdlib.h>
int* createIntArray(unsigned int n)
{
  int * array;
  array = malloc (n * sizeof(int));
  if (array == NULL){
   printf("\nMemory Unvailable.");
   exit(1);
  }
  // POINT 1
  return array;
}
```

```
#include <stdio.h>
int main()
{
  int *a;
  unsigned int size = 5, i;
  a = createIntArray(size);
  for(i = 0; i < size; i++)
     a[i] = i+1;
  // POINT 2
  print(a, 5);
  return 0;
}
```

De-allocating Memory from Heap

- Memory allocated by malloc or calloc can be de-allocated by library function free:

```
void free(void * ptr);
```

- Making the de-allocated memory available again for further allocations.
- If **ptr** is moved from the beginning of the memory space allocated by **malloc** it can cause your program to crash the program.
- In other words **ptr** must have the address returned by malloc or NULL pointer to be properly passed to library function free.

De-allocating Memory from Heap

- Here is an example:

```
char* s = malloc(1024);
strcpy(s, "Apple Orange Grape");
// Advance the pointer…
s += 6;
printf("%s\n", s);
free(s);
```

- Result of freeing a pointer that is not pointing to an address returned from one of the C-library functions, malloc, calloc, or realloc, is an "**Undefined Behavior**". It means implementations running on different systems or compilers may behave differently.
- As an Example: Here is the program output and the error message from running this code segment on the Mac-Xcode

```
Orange Grape
(2387,0x7fff7882b300) malloc: *** error for object 0x100801206:
pointer being freed was not allocated*** set a breakpoint in
malloc_error_break to debug
```

How to show released memory spaces on the heap

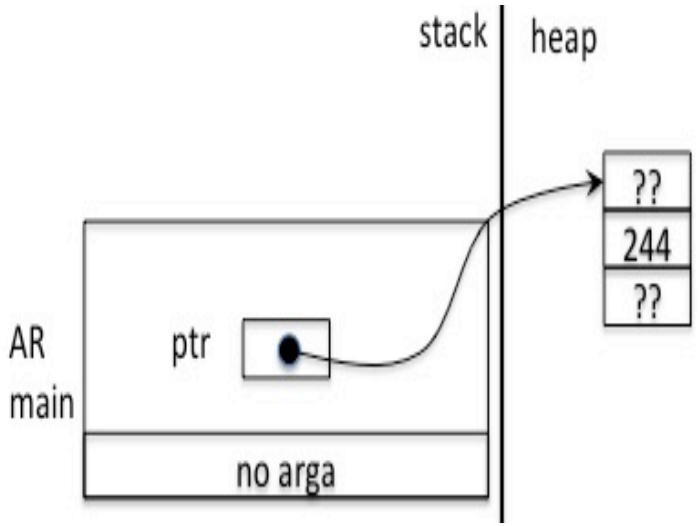
```
int main(void) {
  int* ptr = malloc (3 * sizeof(int));
  ptr[1] = 244;
  // Point 1

  free(ptr);
  // Point 2
  ptr = NULL;
  // Point 3
  return 0;
}
```

At point 3, you should put a zero into ptr and remove the arrow:

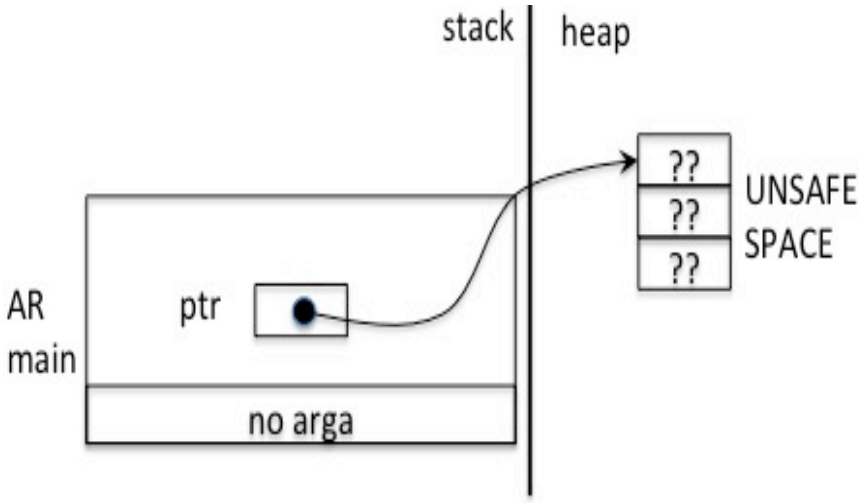
ptr | 0

Also you don't have to show the array on the heap.

POINT 1



POINT 2



Dynamic Allocation of Memory Using calloc

- Another library function to allocate memory is **calloc**. Similar to malloc, the library function calloc can be used to request allocating a block of memory on the heap:

```
void *calloc(size_t count, size_t  size);
```

  - The size of allocated block is obtained by multiplying the two arguments, **count** and **size**.
- Unlike **malloc**, the allocated memory by **calloc** will be automatically initialized to zero.

- Example:

```
int count = 4;
size_t size = sizeof(int);
int * arr;
arr = (int*) calloc (count , size);
      …
```

C Library Function realloc

- The other commonly used memory allocation function is realloc. This function is used to resize an existing dynamically allocated memory space.

```
void *realloc(void *ptr, size_t new_size);
```

- Assume a previously allocated block starting at `ptr`.
- realloc will:
  - change the block size to `new_size`,
  - return pointer to resized block, or NULL if fails to allocate memory.
  - The content of the memory block is preserved up to the lesser of the new and old sizes.
  - If the new *size* is larger, the value of the newly allocated portion is indeterminate.

C Library Function realloc

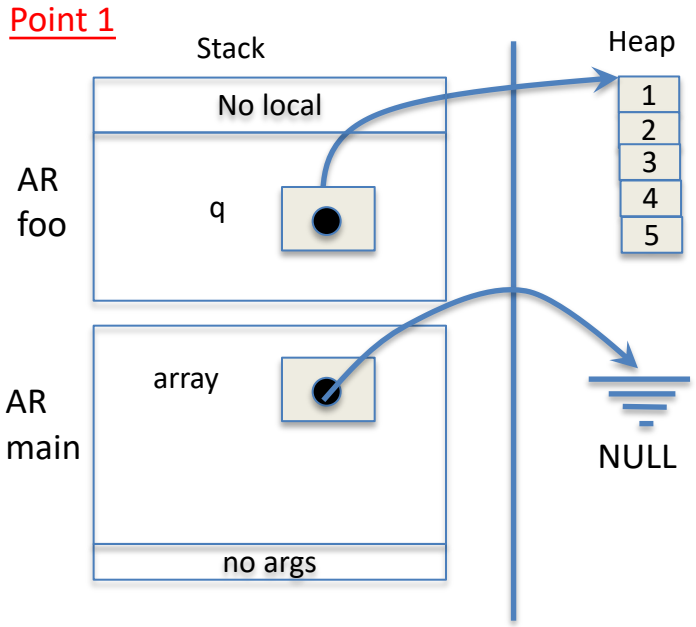
- Here is an example of using **realloc** C-library function. Draw a memory diagram at point 1.

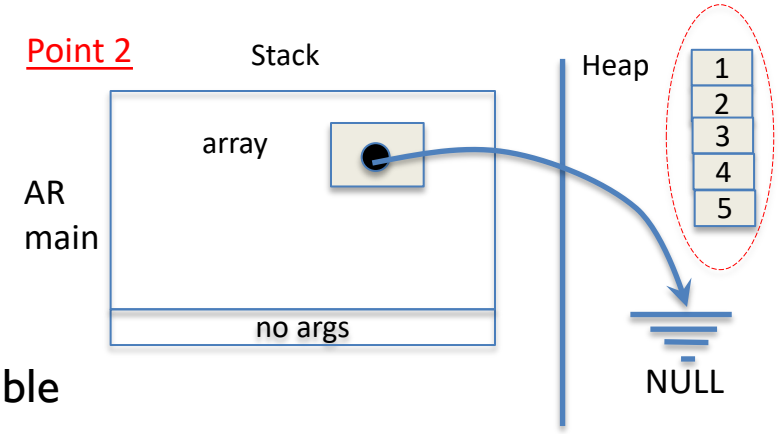
```
int main(void){
    int *a;
    unsigned int size = 5, i;

    a = createIntArray(size);

    for(i = 0; i < size; i++)
       a[i] = i+1;

    a = realloc(a, ++size * sizeof(int) );
    a[size-1] = i+1;
    // Point 1
    return 0;
}
```

Faulty heap management

- What if anything is wrong in the following program

```
void foo(int* q) {
  int i;

  q = malloc(sizeof(int) * 5);

  for(i = 0; i < 5; i++)
    q[i] = i+1;
  // Point 1
}

int main() {
  int* array = NULL;
  foo(array);
  // Point 2
  // MORE CODE
}
```





- The memory created by **malloc** is no longer accessible (called leaked memory)

Objects that Contain Allocated Memory

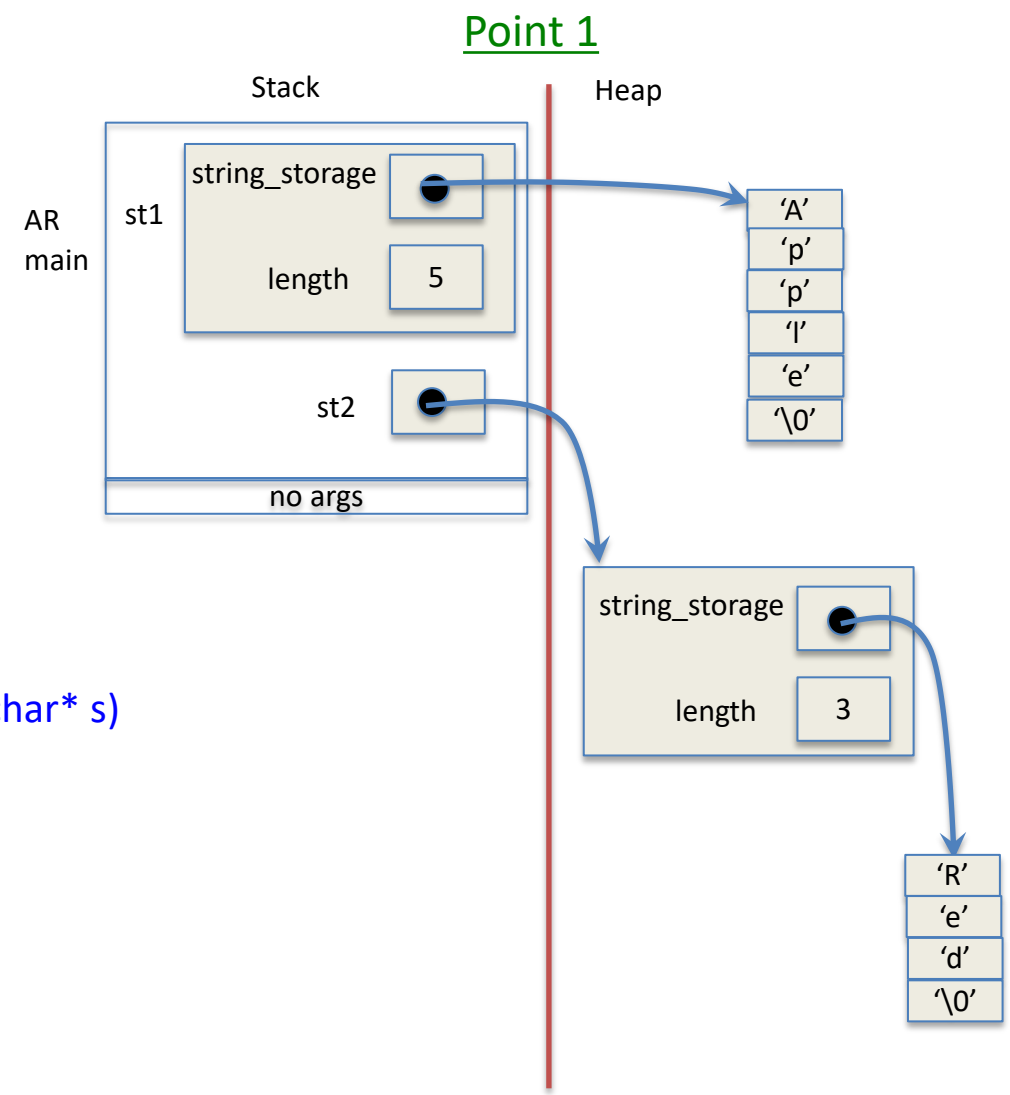
Objects that Contain Allocated Memory

- Dynamic allocation of memory is not limited to direct allocation of simple data types or user-defined types. The source of allocation can be wrapped and nested within a composite object, using user defined data types such as struct type.
- Lets assume we would like to build a user-defined type called String that not only knows about the elements of a C-string (an array of char), but also knows about the length of a string objects:

```
typedef struct String{
    char *string_storage;
    int length;
} String;
```

Assume all header files and function prototypes are declared:

```
int main() {
   String st1 ;
   String *st2;
   construct_String_object (&st1, "Apple");
   st2 = malloc(sizeof (String));
   construct_String_object(st2, "Red");
   // POINT 1
   return 0;
}


void construct_String_object (String* str, const char* s)
{
   int length = strlen(s);
   str -> string_storage = malloc(length +1);
   str -> length = length;
   strcpy(str -> string_storage, s);
}
```



Note: this example is an overly simplified version and this type of implementations require special considerations when you need to make copies of this objects. More functional version of this topic will be discussed in C++.

```
int main()
{
   String st1 ;
   String *st2;

   construct_String_object (&st1, "Apple");
   st2 = malloc(sizeof (String));
   if(st2 == NULL) {
      printf(" Error: Memory NOT Allocated");

   construct_String_object(st2, "Red");

   // POINT 1

   st1 = *st2;
  // Point 2

   return 0;
}
```

Point 2