ECE 220 Computer Systems & Programming

Lecture 18 – Dynamic Memory Allocation

ECE ILLINOIS

ILLINOIS

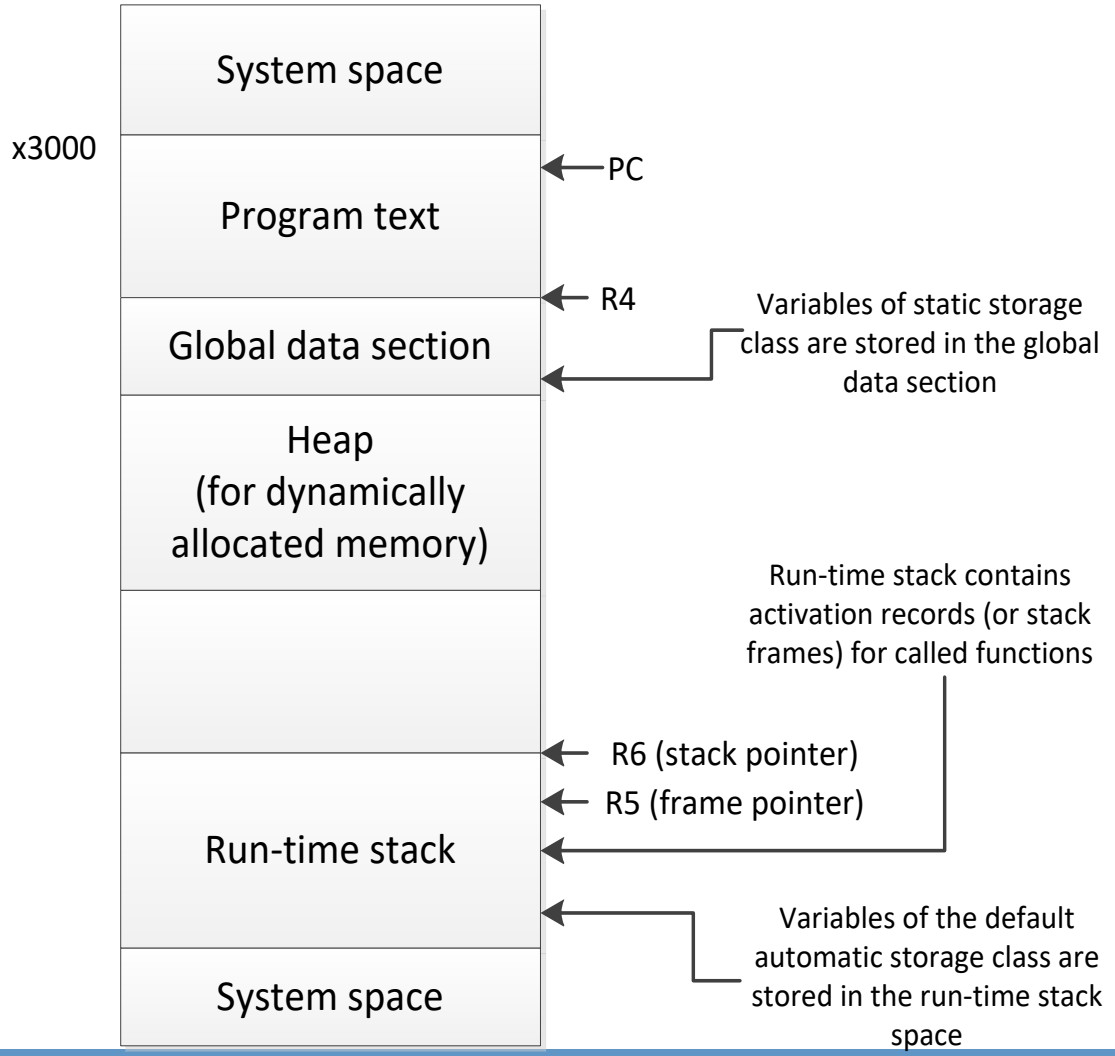
- **Observation from last lecture:**
  - `#define NofS 360`
    `student ece220[NofS];`
  - We allocated an array of student records consisting of 360 elements.
  - But what if next semester ECE 220 class will have only 10 students? Or 400 students?
  - Ideally, we want to allocate as much memory as needed rather than some pre-set amount. We want to dynamically adapt the size of array based on the actual size of class.
  - This can be achieved using the concept of dynamic memory allocation.
- **Dynamic memory allocation**
  - A piece of code called *memory allocation manager* that belongs to the OS manages an area of memory called *heap*.
  - During the execution, a program makes a request to the memory allocator for a contiguous piece of memory of a particular size.
  - The allocator reserves the memory and returns a pointer to it.

We interact with the memory allocation manager by using malloc/free functions.

Dynamic memory allocation concept

As a reminder

- Variables of static storage class are stored in global data section
- Variables of automatic storage class are stored in the run-time stack



malloc/free

- malloc function can be used to allocate some number of bytes of memory in the heap. It reserves a chunk of memory in the heap and returns a pointer to it. The memory is not initialized.
  - malloc prototype
    - void *malloc(size_t size);
    - size is the number of bytes of memory to be allocated
    - the function returns a generic pointer to a generic data type. User can cast this to an appropriate data type. On error, this function returns NULL.
  - Example:
    - char *name;
      name = (char *)malloc(22*sizeof(char));
- 22 bytes of memory will be allocated and a pointer to it will be returned, casted to type char.

free function frees the memory space pointed to by ptr.

- Free prototype:
  - void free(void *ptr);
  - ptr must have been returned by a previous call to malloc. Otherwise, or if free(ptr) has already been called before, undefined behavior occurs. If ptr is NULL, no operation is performed.
  - The free function returns no value.
- Example:
  - free(name);

- this will free the memory allocated by previous calls to malloc.

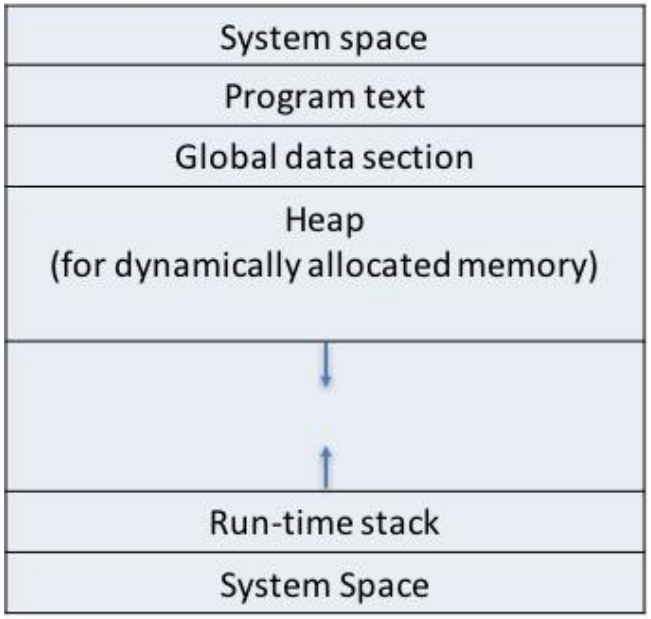
```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main()
{
    char name[20];
    char *address;

    strcpy(name, "Harry Lee");
    address = (char*)malloc( 50 * sizeof(char) );
/* malloc allocating memory dynamically -typecast char */
    strcpy( address, "Lee Fort, 11-B Sans Street");

    printf("Name = %s\n", name );
    printf("Address: %s\n", address );
    free(address);
    return 0;
}
```

Automatic vs. Dynamic Memory Allocation



	Automatic	Dynamic
Mechanism of allocation	automatic	use malloc()
Lifetime of memory	programmer has no control - it ends when exit function/block	programmer has control - use free() to deallocate
Location of memory	stack or global data area	heap
Size of memory	fixed	adjustable

malloc & free

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

- allocates a contiguous region of memory on the heap
- size of allocated memory block is indicated by the argument
- returns a generic pointer (of type void *) to the memory, or NULL in case of failure
- allocated memory is not clear (there could be left over junk data!)

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

- frees the block of memory pointed to by ptr
- ptr must be returned by malloc() family of functions

calloc & realloc

```
void *calloc(size_t n_items, size_t item_size);
```

- similar to malloc(), also sets allocated memory to zero
- n_item: the number of items to be allocated, item_size: the size of each item -> total size of allocated memory = n_items * item_size

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

- reallocate memory block to a different size (change the size of memory block pointed to by ptr)
- returns a pointer to the newly allocated memory block (it may be changed)
- Unless ptr == NULL, it must be returned by the malloc() family of functions
- if ptr == NULL -> same as malloc()
- if size = 0, ptr != NULL -> same as free()

calloc Function

- Like malloc, calloc also **allocates memory at runtime** and is defined in **stdlib.h**. It takes the number of elements and the size of each element(in bytes), initializes each element to zero and then returns a void pointer to the memory.

Its syntax is

void *calloc(n, element-size);

Here, **n** is the number of elements and **element-size** is the size of each element.

```c
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int n,i,*p;
    printf("Enter number of elements: ");
    scanf("%d",&n);
    p=(int*)calloc(n, sizeof(int));
    if(p == NULL)
    {
        printf("memory cannot be allocated\n");
    }
    else{
      printf("Elements of array are\n");
      for(i=0;i<n;i++)
      {
        printf("%d\n",*(p+i));
      }
    }
    return 0;
}
```

```c
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int n,i,*p;
    printf("Enter number of elements: ");
    scanf("%d",&n);
    p=(int*)calloc(n, sizeof(int));
    //memory allocated using malloc
    if(p == NULL)
    {
        printf("memory cannot be allocated\n");
    }
    else
    {
      printf("Enter elements of array:\n");
      for(i=0;i<n;++i)
      {
        scanf("%d",&*(p+i));
      }
      printf("Elements of array are\n");
      for(i=0;i<n;i++)
      {
        printf("%d\n",*(p+i));
      }
    }
    return 0;
}
```

realloc Function

- **void *realloc(pointer, new-size);**

Changes the size of the memory block pointed to by pointer to size bytes. The contents will be unchanged in the range from the start of the region up to the minimum of the old and new sizes.

- If the new size is larger than the old size, the added memory will not be initialized.
- if size is equal to zero, and ptr is not NULL, then the call is equivalent to free(ptr).

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main()
{
    char *p1;
    int m1, m2;
    m1 = 10;
    m2 = 20;
    p1 = (char*)malloc(m1);
    strcpy(p1, "Codesdope");
    p1 = (char*)realloc(p1, m2);
    strcat(p1, "Practice");
    printf("%s\n", p1);
    return 0;
}
```

Exercise:

```
typedef struct studentStruct
{
    char *NAME;
    int UIN;
    float GPA;
}student;
```

1. allocate memory for 200 student records, assuming the you need an array of 100 char to hold each name
2. initialize name to “To be set”, UIN to -1 and GPA to 0.0 for all the records
3. Add 200 more student records
4. free up memory space for all the records

Solution is on github