COP 3502 – Computer Science 1

Lecture 02

Dr. Sahar Hooshmand
sahar.hooshmand@ucf.edu

Department of Computer Science

Slides modified from Dr. Ahmed, with permission

Static Vs Dynamic Memory Allocation

- So far our example declared variables and array were statically allocated memory.
- It means there allocated spaces are not changing.
- In regards to memory allocation the word static and dynamic has the following meaning:
- Static:
  - the memory requirements are known at compile time.
  - after a program compiles, we can perfectly predict how much memory will be needed
  - Whether you take input or not, still that memory spaces will be used
  - Any statically allocated variable can only have its memory reserved while the function within which it was declared is running.
  - For example, if you declare an int x in function F1, if function F1 has completed, no memory is reserved to store x anymore.

Static vs Dynamic Memory Allocation

- **Dynamic:**
- the memory requirements are NOT known at compile-time.
- Memory allocation size may vary based on different execution as it may depends on input.
- Dynamically allocated memory isn't "freed" automatically at the end of the function within which it's declared (although it is NOT ACCESSIBLE automatically from another function)
- This shifts the responsibility of freeing the memory to the programmer.
- This can be done with the *free* function.

malloc, calloc functions

- Malloc or calloc function can be used to dynamically allocate memory
- Malloc:
  - *void *malloc(size_t size);*
  - It allocates unused space for an object whose *size in bytes is specified by size*
  - The value is unspecified in malloc
  - returns a pointer to the beginning of the memory allocated.
  - If the memory can't be found, NULL is returned.

malloc, calloc functions

- **Calloc:**
  - *void *calloc(size_t nelem, size_t elsize);*
  - It allocates an *array of size nelem with each element of size elsize*
  - returns a pointer to the beginning of the memory allocated.
  - The spaces shall be initialized to all bits 0
  - If the memory can't be found, NULL is returned.
- Basically in both function you have to say how many bytes to allocate (how to specify is different).
- Then, if the function successfully finds the memory it returns the pointer to the beginning of the block of the memory returned.
- If unsuccessful, NULL is returned.

Dynamically Allocated Arrays

- Sometimes you don't know how big an array you will need until-runtime.
- You can dynamically allocate memory in those cases.
- *int *ptr1 = (int*) malloc(10*sizeof (int));*
- Remember that *malloc and calloc* return void pointers (void*). So, if you want to use the allocated memory as an array, you must **cast** the array to the type you want. Why?
- The above line could be written in multiple lines:

```
int *ptr1;
ptr1 =  (int*) malloc(10*sizeof (int));
```

Now ptr1 can be treated as an array and you can iterate through it!

-What values are stored in the array?

- *ptr1 = (int*) calloc(10, sizeof (int));*
- How about the values of the ptr1 now if you use calloc?
- See the uploaded code.

Example malloc and calloc

```
void main()
{
    int *ptr1 = (int*)
malloc(10*sizeof (int));  //why are
we casting?
    int *ptr2;
    int i;
    if (ptr1 == NULL)
    {
      printf("Could not allocate
memory\n");
      exit(-1);
    }
    else
    {
        printf("Memory allocated.
Printing data: \n");
        for(i=0; i<10; i++)
            printf("%d  ",
ptr1[i]); //it will print garbage
values
    }
//due to lack of space, the next
lines are shown in the right side
```

```
ptr2 = (int*) calloc(10, sizeof
(int));
    if (ptr2 == NULL)
    {
      printf("Could not allocate
memory\n");
      exit(-1);
    }
    else
    {
        printf("Memory allocated. Printing
data after calloc: \n");
        for(i=0; i<10; i++)
        {
            printf("%d  ", ptr2[i]);
//it will print 0 for all elements
        }
    }
    free(ptr1);
    free(ptr2);
}
```

Some notes to remember

- malloc and calloc allocates specified amount of bytes
- returns void*
- So, <u>cast</u> them so that you can iterate through it or let it know what type of data are you going to refer with this.
- Additional notes: In recent compilers you can skip casting, because your program will cast it automatically based on the target pointer type
- Think about you have an array of structure type data?
- Let's say you have a structure called Student with *name* and *score*. How would you allocate and create an array of N students?
- *Struct Student *students;*
- *students = (struct Student*) malloc(N*sizeof (struct Student));*

There is a risk of creating memory leak!

- Consider the following lines of code:

```
int *ptr1 = (int*) malloc(10*sizeof (int));
int *ptr2 = (int*) malloc(10*sizeof (int));
ptr1= ptr2;
```

- Can you find any problem in the above piece of code?
- After the line ptr1=ptr2, you code does not have any access to the memory allocated by first malloc that was assigned to ptr1. But still that memory is being used by your code. It is called memory leak.
- So, we have to be careful while working with dynamic memory allocation so that our code does not create any memory leak.

realloc

- There might be cases when the allocated array size is not enough and you will need to resize the array.
- How would you approach this?
- Naïve approach would be:
  - Allocate new memory
  - Copy the old data to the new allocated array
  - Free the old array.
- We can avoid extra work through realloc function.
- void *realloc(void **ptr*, size_t *size*);
- So here, ptr is the old pointer, and size is the new size.
- It will allocate size amount of bytes and copy the content from the allocated data in ptr and return void pointer.
- Let's say *values* is an integer pointer and already allocated to numVals size. The following line will reallocate.
- *values = (int*)realloc(values,(numVals+EXTRA)*sizeof(int));*
- See uploaded example.

Example of realloc

```
#include <stdio.h>
#include <time.h>
#define EXTRA 10
int main() {
    int numVals;
    srand(time(0));
    printf("How many numbers?\n");
    scanf("%d", &numVals);

    int* values =
   (int*)malloc(numVals*sizeof(int));
    int i;
    for (i=0; i<numVals; i++)
        values[i] = rand()%100;
    for (i=0; i<numVals; i++)
      printf("%d ", values[i]);
    printf("\n");

//see the right side for remaining
lines of the code
```

```
values =
(int*)realloc(values,(numVals+EXTRA)*
sizeof(int));

    for (i=0; i<EXTRA; i++)
        values[i+numVals] =
                       rand()%100;

    numVals += EXTRA;

    for (i=0; i<numVals; i++)
        printf("%d ", values[i]);

    printf("\n");

    free(values);
    return 0;
}
```

```
//output
How many numbers?
5
97 62 7 74 48
97 62 7 74 48 44 23 75 61 64 69 92 39 23 58
Process returned 0 (0x0)   execution time : 3.112 s
```