Dynamic Memory Allocation in C

Computer Science Department
University of Central Florida

COP 3502 – Computer Science I

Dynamically Allocated Mem. in C

- Throughout the C course (COP 3223)
  - All variable declarations were statically allocated memory
  - The word “static” means “not changing”
  - The word “dynamic” means “changeable”
    - roughly speaking
  - So we essentially work with two types of memory:
    - Static
    - Dynamic

Dynamically Allocated Mem. in C

- Static Memory in C (review)
  - Memory requirements are known at compile time
  - After a program compiles, we can perfectly predict how much memory will be needed for statically allocated variables
  - Although the program may receive different input on different executions of the code
    - This does NOT affect how much memory is allocated
  - One serious consequence of this:
    - A statically allocated variable can only have its memory reserved while the function, within which it was declared, is running
    - Ex: if you declare an “int x” within function A, once function A has completed, the memory for x is not reserved.

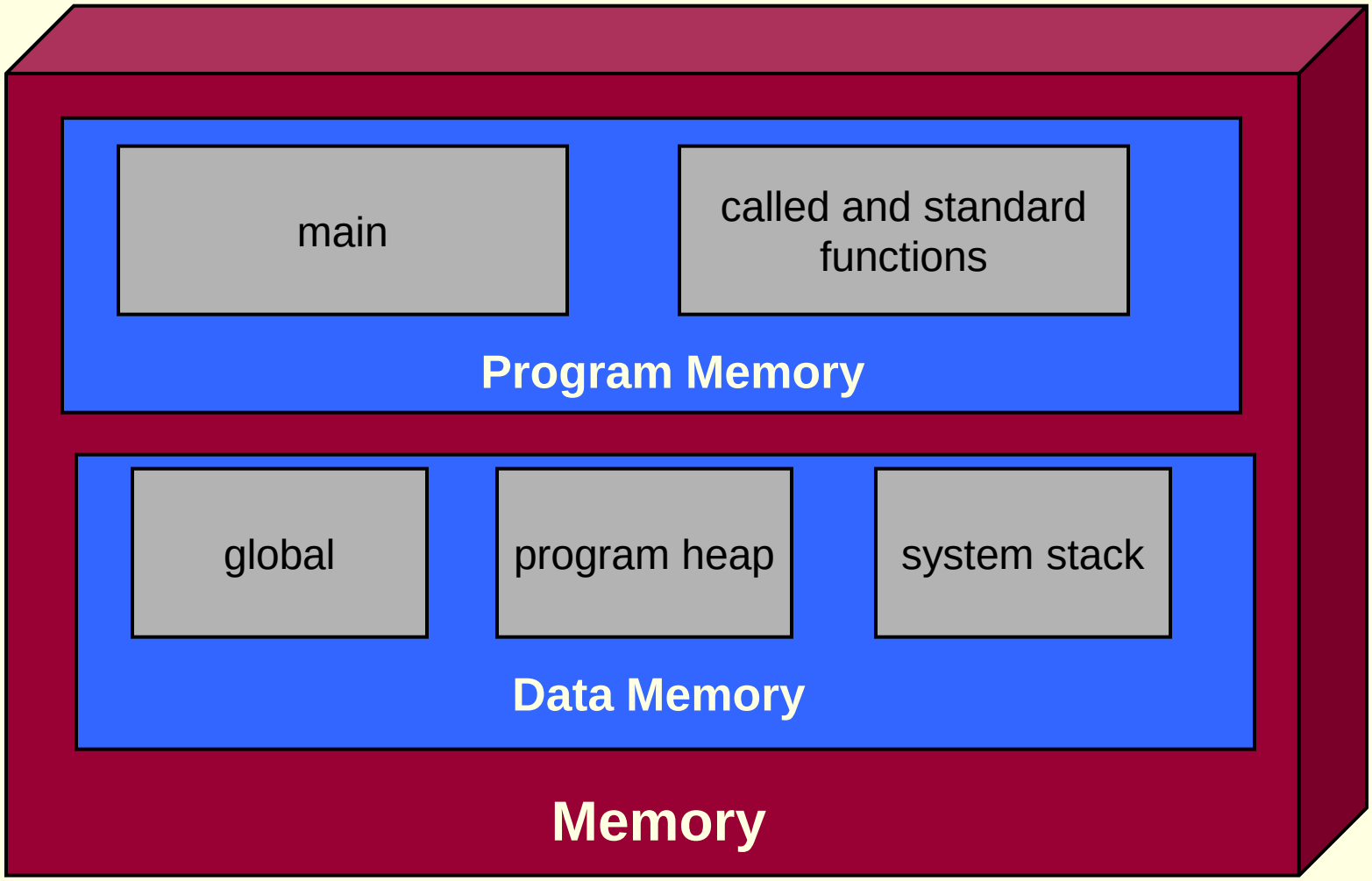
Dynamically Allocated Mem. in C

- Dynamic Memory in C
  - Memory requirements are NOT known (for sure) at compile time
    - Perhaps different amounts of memory are allocated on different executions of the program
      - That is, if the input affects the memory allocation
  - Benefit: memory allocated within a function can be made available outside the function
    - This memory must be allocated within the function
    - And it must be allocated dynamically
  - Caveat:
    - Dynamically allocated memory is NOT “freed” automatically
    - It is the programmers job to free memory
    - This is done with the free function

Dynamically Allocated Mem. in C

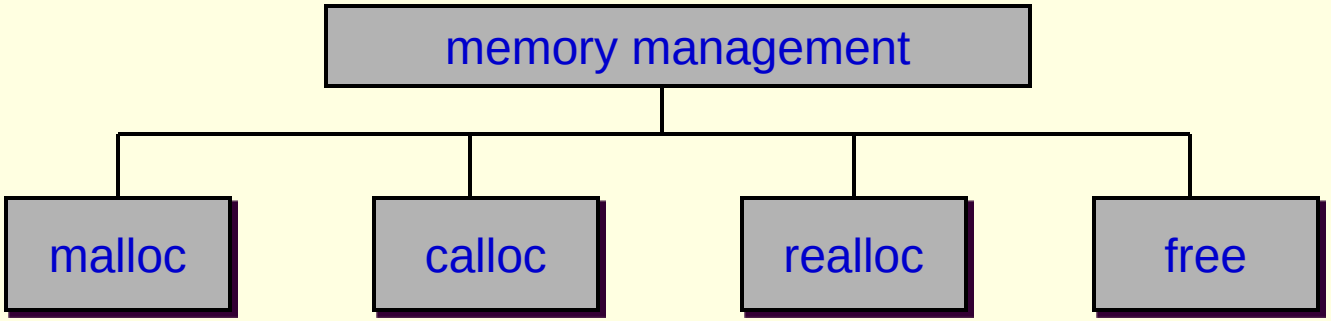
- Conceptually, memory is divided into:
  1) program memory which is used for main and all called functions, and
  2) data memory which is used for global data, constants, local definitions and dynamic memory.
  - Obviously, main must be in memory at all times.
    - Each called function must only be in memory while it or any of its called functions are active.
    - Since multiple copies of a function may be active at one time (recursion) the multiple copies of the variables are maintained on the stack.
    - The heap memory is unused memory allocated to the program and available to be assigned during execution.
  - The next page illustrates the conceptual view of memory.

Conceptual View of Memory



Dynamically Allocated Mem. in C

- Four memory management functions are used with dynamic memory in the C language.
  - malloc, calloc, and realloc are used for memory allocation.
  - free is used to return allocated memory to the system when it is no longer needed.
- All the memory management functions are found in the standard library header file <stdlib.h>.



Memory Management Functions

- `malloc`
  - Formal description:

```
// Allocates unused space for an object
// whose size in bytes is specified by size
// and whose value is unspecified, and
// returns a pointer to the beginning of the
// memory allocated. If the memory can't be
// found, NULL is returned.

void *malloc(size_t size);
```

Memory Management Functions

- `calloc`
  - Formal description:

```
// Allocates an array of size nelem with
// each element of size elsize, and returns
// a pointer to the beginning of the memory
// allocated. The space shall be initialized
// to all bits 0. If the memory can't be
// found, NULL is returned.

void *calloc(size_t nelem, size_t elsize);
```

Memory Management Functions

- `malloc` & `calloc`
  - Seem confusing?
    - Both descriptions basically say that you need to tell the function how many bytes to allocate
      - How you specify this to the two functions is different
    - Then, if the function successfully finds the memory
      - A pointer to the beginning of the block of memory is returned
    - If unsuccessful
      - NULL is returned

Dynamically Allocated Mem. in C

- An Example: Dynamically Allocated Arrays
  - Sometimes you won't know how big an array you will need for a program until run-time
    - So you dynamically allocated space for the array
      - Using a pointer
  - Consider the following program:
    - Simply reads from a file of numbers (integers)
    - Assume that the first integer in the file stores how many integers are in the rest of the file
    - What does the program do:
      - reads in all the values into the dynamically allocated array
      - and prints them out in reverse order

Dynamically Allocated Mem. in C

- An Example: Dynamically Allocated Arrays
  - Sometimes you won't know how big an array you will need for a program until run-time
    - So you dynamically allocated space for the array
      - Using a pointer
  - Consider the following program:
    - Let's say the program reads in a 10
      - Meaning, there will be 10 integers that we need to read in
      - So we will allocate space for those ten integers, read them in, and then print them in reverse order
    - If our ten integers are: 4 2 3 1 5 3 2 9 3 7
    - Our program should print: 7 3 9 2 3 5 1 3 2 4

```
#include <stdio.h>
int main() {

   int *p, size, i;
   FILE *fp;
   // Open the input file.
   fp = fopen("input.txt", "r");
   // First int read shows how many numbers
   fscanf(fp, "%d", &size);

   // Make memory and read numbers into array.
   p = (int *)malloc(size*sizeof(int));
   for (i = 0; i<size; i++)
        fscanf(fp, "%d", &p[i]);

   // Print out the array elements backwards.
   for (i = size-1; i>=0; i--)
        printf("%d\n", p[i]);

   // Close the file and free memory.
   free(p);
   fclose(fp);
   return 0;
}
```

Note the actual parameter passed to the `malloc` function.

We must specify the total # of bytes we need for the array.

This number is the **product** of the number of array elements and the size (in bytes) of each array element.

Dynamically Allocated Mem. in C

- An Example: Dynamically Allocated Arrays
  - Using `calloc` instead of `malloc`
    - It should be fairly easy to see how we can change the above code to use `calloc` instead of `malloc`
    - For this example, however, there is no need to initialize the whole block of memory to 0
      - Which is the benefit of `calloc`
    - So there's no obvious advantage of using `calloc`
    - But when you want to initialize all the memory locations to 0
      - `calloc` is the clear function of choice as it takes care of that for you

Dynamically Allocated Mem. in C

- Extra notes on pointers and dynamic arrays
  - The return type of `malloc` is void*
    - This means that the return type for `malloc` MUST be casted
      - To what?
        - To the type of pointer that will be pointing to the allocated memory
    - What is the reason?
      - `malloc` is used to allocate memory for all types of structures
      - If `malloc` only returned an int *, for example, then we couldn't use it to allocate space for a character array
    - So `malloc` simply returns a memory location
      - It doesn't specify what is going to be stored in that memory

Dynamically Allocated Mem. in C

- Extra notes on pointers and dynamic arrays
  - The return type of `mallic` is void*
    - Thus, the programmer should cast the return value, from `malloc`, to the type they want
    - All this does is specify what size “chunks” the memory should be broken down into
    - Once we know what we are pointing to, we know how many contiguous memory locations store a piece of data of the array

Dynamically Allocated Mem. in C

- Extra notes on pointers and dynamic arrays
  - The return type of `mallic` is void*
    - Example:
      - You want to create an array that is 800 bytes long
      - How many cells are in that array?
      - Well, it depends on what “size” each cell will be
      - If you want an array of integers, which are 4 bytes each, then you will have 200 cells (800 total bytes / 4 bytes)
      - But if you want an array of doubles, which are 8 bytes each, then you will have 100 cells (800 total bytes / 8 bytes)
      - So again, when you `malloc` your space, you need to “cast” that space to whatever type you want (int, float, double, etc)
      - That then determines how many chunks (and what size) the allocated memory is broken into

Dynamically Allocated Mem. in C

- Extra notes on pointers and dynamic arrays
  - `malloc` can fail to find the needed memory within the heap
    - If this occurs, `malloc` returns NULL
    - Good programming should check for this after each `malloc` call
  - Mind you, this should rarely happen
    - But the potential is there if you do not free memory when possible
    - When you are done using a dynamic data structure
      - Use the free function to free that memory!

Memory Management Functions

- `realloc`
  - What if your dynamically allocated array gets filled?
    - And now you want to “extend” it because more elements need to be stored
  - Based on what you know thus far, we could:
    - Allocate new memory larger than the old memory
    - Copy over all the values from the old memory to the new
    - Free the old memory
    - And now we can add new values to the new memory
  - `realloc` is a function that does all this for us!

Memory Management Functions

- `realloc`
  - `void *realloc(void *ptr, size_t size);`
  - Description for IEEE standards web page:
    - The `realloc()` function shall change the size of the memory object pointed to by *ptr* to the size specified by *size*. The contents of the object shall remain unchanged up to the lesser of the new and old sizes. If the new size of the memory object would require movement of the object, the space for the previous instantiation of the object is freed. If the new size is larger, the contents of the newly allocated portion of the object are unspecified. If *size* is 0 and *ptr* is not a null pointer, the object pointed to is freed. If the space cannot be allocated, the object shall remain unchanged.
      - Basically describes the various contingencies of what can possibly happen in atypical situations

realloc example

```
#include <stdio.h>
#include <time.h>

#define EXTRA 10

int main() {
   int numVals;
   srand(time(0));
   printf("How many random numbers do you want?\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");
   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;
}
```

Now let's just say that for some crazy reason, we now want 10 extra random numbers

Brief Interlude: Human Stupidity

Dynamically Allocated Mem. in C

- How to create a dynamically allocated array inside a function
  - Main idea is the same as if you did this in main
  - BUT, you MUST return a pointer to the newly created array
  - Otherwise you won't have access to the memory

Dynamically Allocated Mem. in C

- How to create a dynamically allocated array inside a function
  - Example of function:

```
int* readArray(FILE* fp, int size) {

        int *p = (int *)malloc(size*sizeof(int));
        for (i = 0; i<size; i++)
                fscanf(fp, "%d", &p[i]);
        return p;
}
```

Dynamically Allocated Mem. in C

- How to create a dynamically allocated array inside a function
  - Here's how we call this function from main:

```
fp = fopen("input.txt", "r");
fscanf(fp, "%d", &size);
int *numbers = readArray(fp, size);
```

  - What's going on:
    - Array is created while `readArray` is running
    - A pointer to the beginning of the array is returned
    - `numbers` (from main), which itself, is a **pointer**, is set to point to the newly allocated memory

Dynamically Allocated Mem. in C

- How to create a dynamically allocated structure from a function
  - We will create the following struct and return a pointer to it from a function:

```
struct bigInteger {
    int* digits;
    int size;
};
```

- Remember from COP3223
  - A struct is simply a “skeleton”
  - When you define a struct, you are not actually making an instance of that struct. It is just a skeleton that you will then reference later in the program.

  - The following function creates a random `struct bigInteger`, dynamically, and returns a pointer to it

Dynamically Allocated Mem. in C

```
struct integer* createRandBigInt(int numDigits) {

        struct bigInteger* temp;
        temp = (struct bigInteger*)malloc(sizeof(struct bigInteger));
```

- So what's going on here:
  - We make a pointer called temp, which is of type `struct bigInteger`
  - Then we `malloc` space for **one** actual `struct bigInteger`
  - We cast it appropriately, and we save that pointer into `temp`
  - And that is okay, right.?.
  - Because `temp` is a pointer of type `bigInteger`
    - Meaning, it expects an address, and at that address it expects to find a `struct bigInteger`

Dynamically Allocated Mem. in C

```
struct integer* createRandBigInt(int numDigits) {

        struct bigInteger* temp;
        temp = (struct bigInteger*)malloc(sizeof(struct bigInteger));
```

- Are we now ready to store stuff into this structure?
  - Normally, after you allocate a structure, you can store into it
  - And remember that we want to store an array of integers
  - But did our structure ("skeleton") specify a specific array size
    - Such as `int digits[10000]`
    - No, it did not!
  - The first member of our structure, int *digits, is a pointer (will be an array)
  - This means we must ALSO allocate space specifically for that array of ints

Dynamically Allocated Mem. in C

```
struct integer* createRandBigInt(int numDigits) {

        struct bigInteger* temp;
        temp = (struct bigInteger*)malloc(sizeof(struct bigInteger));

        temp->digits = (int*)malloc(numDigits*sizeof(int));
        temp->size = numDigits;
```

- So this is what we just did:
  - We malloced space for our array of digits
  - And then we also edited the `size` member of the struct
    - Making `size` equal the `numDigits`
  - And now, we simply randomly enter values into each digit in this array…

Dynamically Allocated Mem. in C

```
struct integer* createRandBigInt(int numDigits) {

        struct bigInteger* temp;
        temp = (struct bigInteger*)malloc(sizeof(struct bigInteger));

        temp->digits = (int*)malloc(numDigits*sizeof(int));
        temp->size = numDigits;

        temp->digits[numDigits-1] = 1 + rand()%9;

        int i;
        for (i=0; i<numDigits-1; i++)
                temp->digits[i] = rand()%10;

        return temp;
}
```

Dynamically Allocated Mem. in C

- How to create a dynamically allocated structure from a function
  - Note that there were **TWO separate mallocs**:
  - The first `malloc` allocates space for the struct itself
    - This space is ONLY enough for one integer **pointer** (small amount of space) and one actual integer
      - `int *digits` and `int size`
  - The second `malloc` allocates space for the integer array (that digits will point to) within the struct
    - Which is potentially a large amount of space
      - Depends on the variable `numDigits`

Dynamically Allocated Mem. in C

- How to create a dynamically allocated array of structs from a function
  - We will create the following `struct` and return a pointer to it from a function:
    - Works very, very similar to allocating an `int` array dynamically
    - Only difference is that instead of using `int`, you use the `struct`
    - We use the following `struct` for this example:

```
struct point {
    int x;
    int y;
};
```

Dynamically Allocated Mem. in C

This function creates an array of struct point, dynamically, fills it with random points, and returns a pointer to the front of the array:

```
struct point* createRandPoints(int size, int maxVal)  {

        struct point* temp;
        temp = (struct point*)malloc(size*sizeof(struct point));

        int i;
        for (i=0; i<size; i++) {
                temp[i].x = 1 + rand()%maxVal;
                temp[i].y = 1 + rand()%maxVal;
        }

        return temp;
}
```

Dynamically Allocated Mem. in C

- How to create a dynamically allocated array of structs from a function
  - Note that we only have one malloc
    - For the array itself
  - This allocates all the space we need in one step
  - Once space is allocated, we treat each array location as an individual struct
  - Notice we can use “.” to access the struct members

Dynamically Allocated Mem. in C

- How to create a dynamically allocated array of pointers to structs
  - Effectively, we accomplish the same general task as the previous example
  - This time, however, our array elements will only store a POINTER to the struct instead of the struct itself
    - We use the same `struct` for this example:

```
struct point {
    int x;
    int y;
};
```

Dynamically Allocated Mem. in C

- How to create a dynamically allocated array of pointers to structs

```
struct point** createRandPoints(int size, int maxVal)  {

        struct point** temp;
        temp = (struct point**)malloc(size*sizeof(struct point*));

        int i;
        for (i=0; i<size; i++) {
                temp[i] = (struct point*)malloc(sizeof(struct point));

                temp[i]->x = 1 + rand()%maxVal;
                temp[i]->y = 1 + rand()%maxVal;
        }

        return temp;
}
```

Dynamically Allocated Mem. in C

- How to create a dynamically allocated array of pointers to structs
  - Notice the double pointer
    - The first pointer is for the array
      - It refers to the array of pointers that we are making
    - The second pointers is for the contents of each array element
  - Notice we have two allocations
    - The first allocation is for the array of pointers
    - For each array element, we must allocate space for the individual struct that is being pointed to
  - **Notice the use of "->" since temp[i] is a pointer**

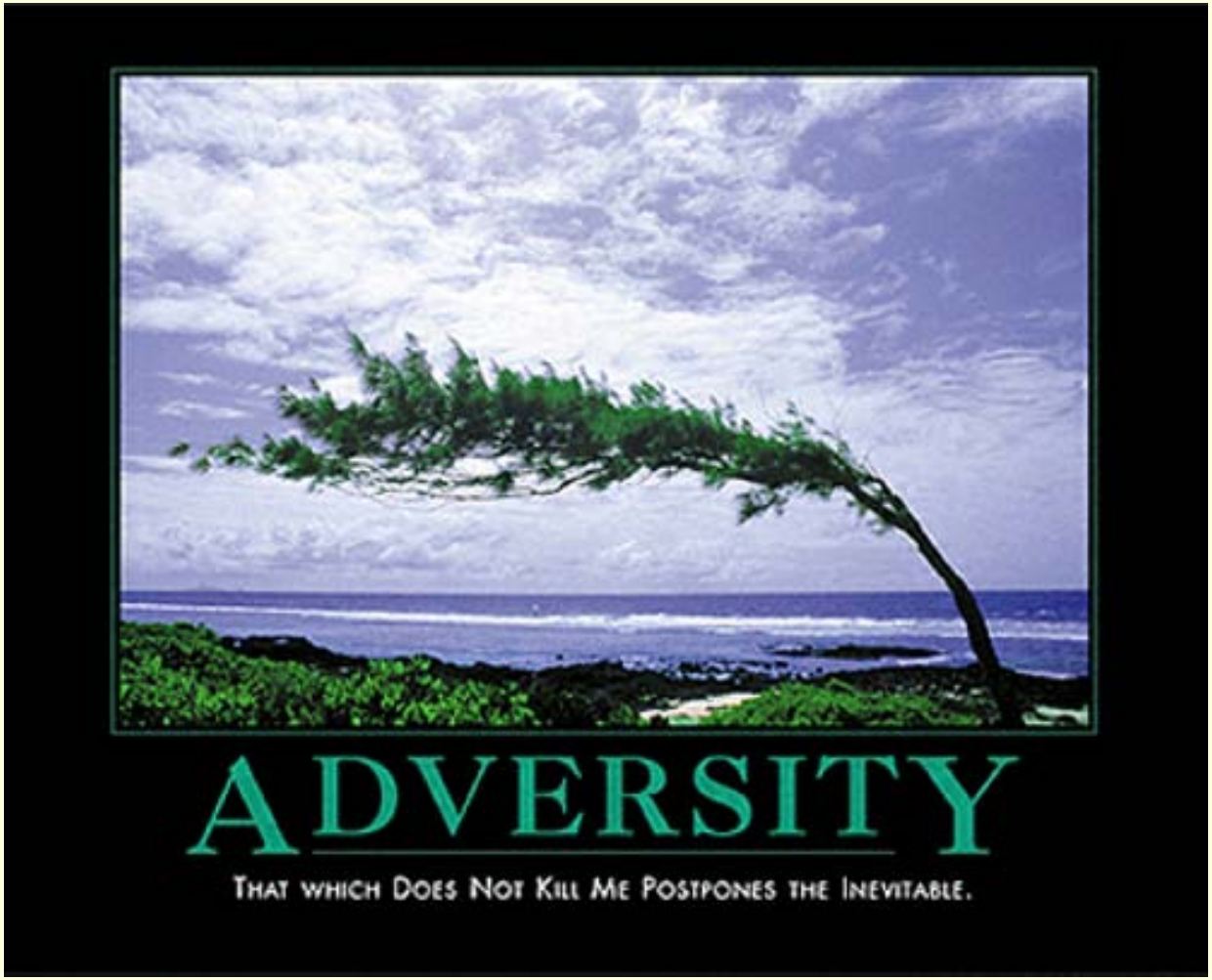
Dynamically Allocated Mem. in C

WASN'T

THAT

FUN!

Daily Demotivator



Dynamic Memory Allocation in C

Computer Science Department
University of Central Florida

COP 3502 – Computer Science I