Dynamic Memory Allocation

COP 3502

Dynamically Allocated Memory in C

- All throughout COP 3223, all examples of variable declarations were statically allocated memory.
- We will work with 2 types of memory in C:
  - **Static** –
    - “not changing”
  - **Dynamic** –
    - “changeable” (roughly speaking ;)

Dynamically Allocated Memory in C

1) Static

- The memory requirements are ***known*** at compile time.
  - Specifically, after a program compiles the compiler can perfectly predict how much memory will be needed and when for ***statically allocated variables.***
- A program can have different inputs on each execution of the code
  - But this does NOT affected the amount of memory allocated.
- The memory is only allocated for a static variable while in the function it was declared in is running.
  - For Example: if you declare `int x` within function `someFunc`, once function `someFunc` has completed, the memory for `x` is no longer saved.

Dynamically Allocated Memory in C

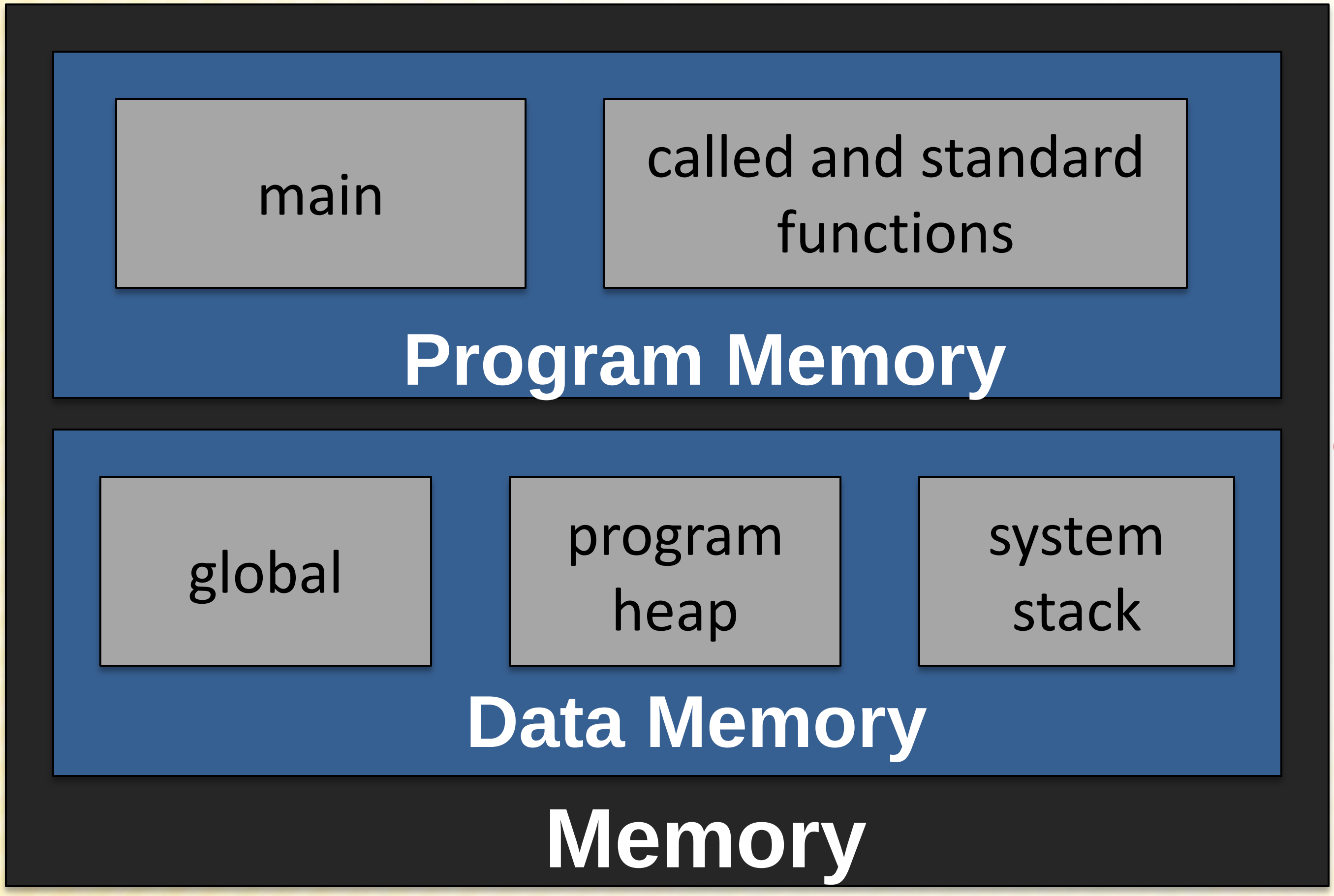
1) **Dynamic**

- The memory requirements **NOT *known*** at compile time
  - It may be the case that on different executions of the program, different amounts of memory are allocated;
    - the input may affect memory allocation.
- *If you want to allocate memory in one function,* ***AND*** *have that memory available after the function is completed,*
  - ***you HAVE to allocate memory dynamically in that function!!!***
- ***CAUTION:***
  - Since dynamically allocated memory isn't "freed" automatically at the end of the function within which it's declared
  - We (the programmers) have to free this memory!! :-O

program memory - used for main and all called functions

Each called function must only be in memory while it or any of its called functions are active.

Obviously, main must be in memory at all times.



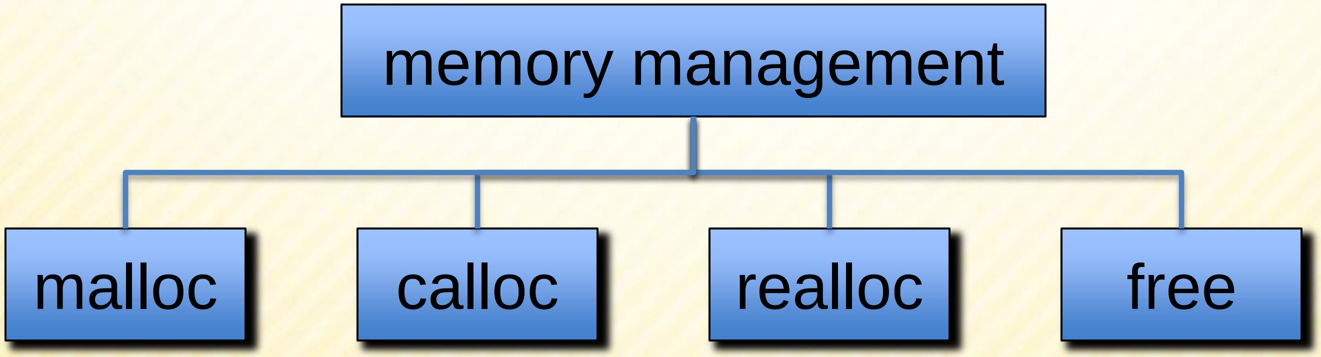
data memory - used for global data, constants, local definitions and dynamic memory.

The heap memory is unused memory allocated to the program and available to be assigned during execution.

Since multiple copies of a function may be active at one time (recursion) the multiple copies of the variables are maintained on **the stack**.

Dynamically Allocated Memory 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

- There are two functions we will typically use to allocate memory dynamically:
  - `malloc`
  - `calloc`

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
  - What's the difference?
    - ➢Both descriptions basically say that you need to tell the function <u>how many bytes to allocated</u>
      - – <u>How you specify this to the two functions is different</u>
    - ➢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 Memory in C

- An example:

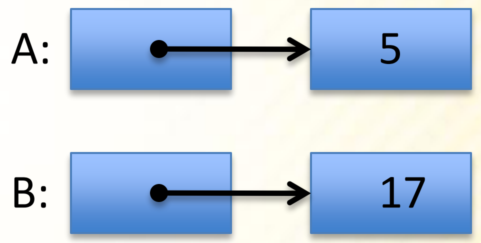
```
#include <stdio.h>
#include <stdlib.h>
int main() {

       // declare 2 pointers
       int *A, *B;

       // allocate memory for the pointers
       A = (int *)malloc(sizeof(int));
       B = (int *)malloc(sizeof(int));

       // Store the int 5 where A is pointing to
       *A = 5;

       // Store the int 17 where B is pointing to
       *B = 17;
}
```



Question & Answer Time

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

        // declare 2 pointers
        int *A, *B;

        // allocate memory for the pointers
        A = (int *)malloc(sizeof(int));
        B = (int *)malloc(sizeof(int));

        // Store 5 where A is pointing to
        *A = 5;

        // Store 17 where B is pointing to
        *B = 17;
}
```

- In C an int and a pointer to an int are different types.
  - And there's no automatic conversion to change the right side to a pointer
  - The C compiler won't compile the program, since this is a "type mismatch error"

Question & Answer Time

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

        // declare 2 pointers
        int *A, *B;

        // allocate memory for the pointers
        A = (int *)malloc(sizeof(int));
        B = (int *)malloc(sizeof(int));

        // Store 5 where A is pointing to
        *A = 5;

        // Store 17 where B is pointing to
        *B = 17;
        printf("B = 0x%x", B);
}       printf("*B = %d, *B);
```

B is the memory address of the int stored there.

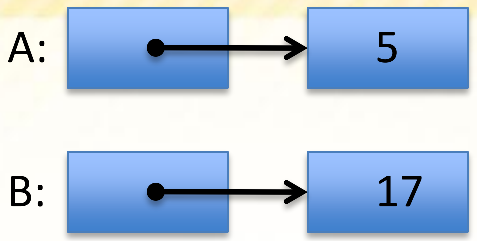
Suppose you try to print the value of the pointer stored in B?

- C permits pointer values to be printed.
- The `printf` statement prints an answer such as `B == 0xf6da`, (this is hexadecimal since %x prints values in hex.)

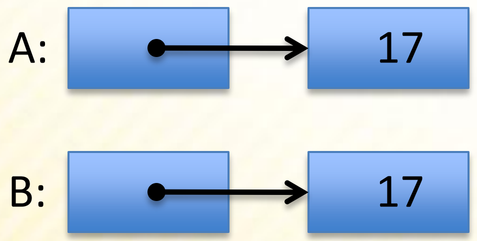
Question and Answer Time

- So what is a pointer?
  - A memory address!

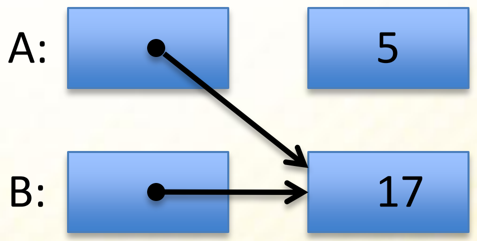
Question and Answer Time



- Starting with the situation above,
  - which of the following diagrams results if we perform the assignment: **`A = B;`** ?

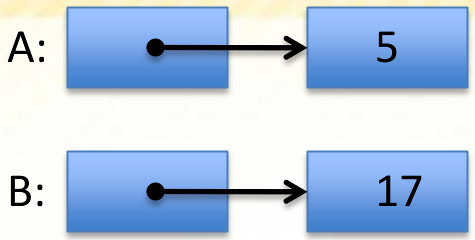


Left Diagram

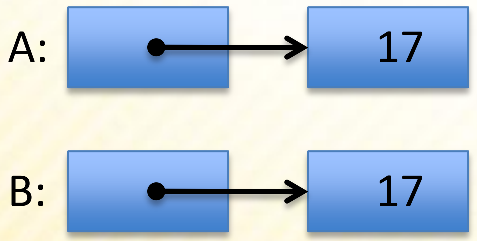


Right Diagram

Question and Answer Time



- Starting with the situation above,
  - What assignment statement would we perform if we wanted to create the situation shown in the left diagram?
    - ***A = *B**



Left Diagram

Dynamically Allocated Memory in C

- Another Example: Dynamically Allocated Arrays
  - Sometimes you won't know how big of an array you will need for a program until run-time
    - We can't do this:
    - `int inputArray[?];`
    - `int inputArray = {?, ?, ?, ?, …};`
  - So you dynamically allocate space for the array
    - Using a pointer at runtime
    - `int size;`
    - `scanf("%d", &size);`
    - `int *inputArray = (int*)malloc(size*sizeof(int));`

Dynamically Allocated Memory in C

- 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
  - 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>
#include <stdlib.h>
int main() {

    int *p, 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 parameters passed to **malloc**

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

Which is the ***product*** of the # of array elements AND the size (in bytes) of each array element.

Dynamically Allocated Memory in C

- Using `calloc` instead of `malloc`
  - We used: `p = (int *)malloc(size*sizeof(int));`
  - We could have done :
    - `p = (int *)calloc(size, sizeof(int));`
  - But for this example there was no need to initialize the whole block of memory to 0
    - Which is the benefit of `calloc`
  - So when you want to initialize all the memory locations to 0
    - `calloc` is the right choice since it does it for you
    - Thanks calloc! ☺

Dynamically Allocated Memory 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 will be stored there.

Dynamically Allocated Memory in C

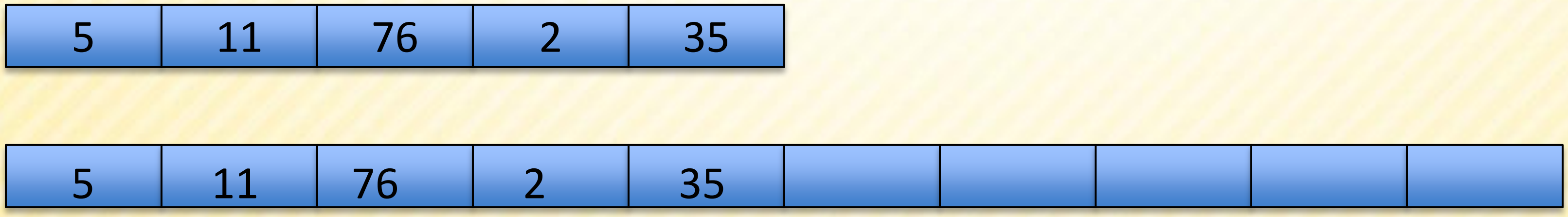
- Extra notes on pointers and dynamic arrays
  - The return type of **`malloc`** is void*
    - Example:
      - You want to create an array that is 800 bytes long
      - How many cells are in that array?
      - Well, it depends on what "size" each cell will be
      - If you want an array integers, 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 Memory 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
  - Rare… 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!

Dynamically Allocated Memory in C

- **`realloc`**
- Sometimes an array gets filled
  - but you want to “extend” it because more elements must be stored.
  - Based on dynamic memory allocation this could be solved by:
    1) Allocate new memory larger than the old memory.
    2) Copy over all the values from the old memory to the new.
    3) Free the old memory.
    4) Now we can add new values to the new memory.
  - Can avoid extra work through a function that does it for us: **`realloc`**
    - ➢ `void *realloc(void *ptr, size_t size);`
    - ➢ However, the steps above still happen behind the scenes, so is not used often because it is inefficient.



Dynamically Allocated Memory in C

- Short example of **`realloc`** …

realloc example

```
#define EXTRA 10
int main() {
    int numVals;
    srand(time(0));

    printf("How many numbers do you want to pick?\n");
    scanf("%d", &numVals);
    int* values = (int*)malloc(numVals*sizeof(int));

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

    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 we now want 10 extra random numbers.

Dynamically Allocated Memory in C

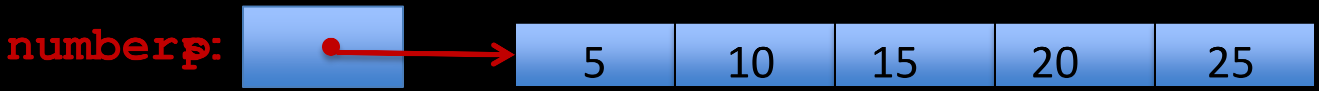
- How to create a dynamically allocated array in a function
  - The key idea is very similar to doing this task in main, but you have to return a pointer to the array created.
  - Program shown in class …

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

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

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

        return p;
}
```



```
int main() {

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

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

        return 0;
}
```

Dynamically Allocated Memory in C

- How to create a dynamically allocated structure in a function
  - AGAIN, the key idea is very similar to doing this task in main, but you have to return a pointer to the structure created.
  - Program shown in class …

Dynamically Allocated Memory in C

- Also shown in class,
- How to create a dynamically allocated array of structs from a function
- How to create a dynamically allocated array of pointers to structs