COMP 206 – Introduction to Software Systems

Lecture 12 – Taking control of memory dynamically!

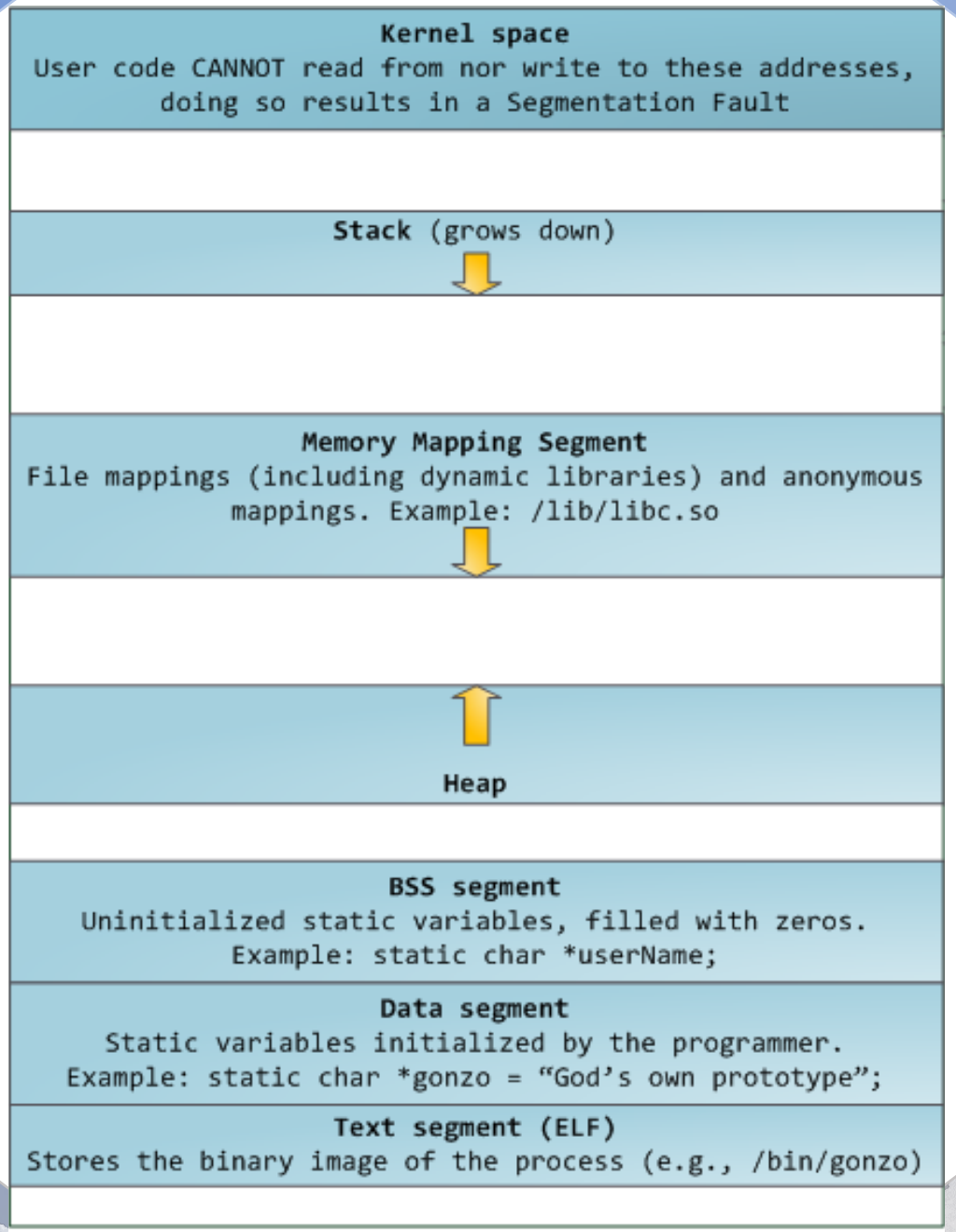
October 10th, 2018

Key thoughts on memory

- Your a.out is a file, it takes space on disk.
  - What is inside?
- Each running process requires system resources.
  - You have likely experienced your computer running slowly when you start too many programs, when a large video plays, gaming, etc.

- When we create systems programs, we have control over how the memory is used.
  - How-to guide today!

A process in memory



Elements of a running process:

- Text space
- Data segment
- BSS segment
- Heap
- Stack
- Kernel space
- Memory mapping

Statically Sized Parts that come from gcc

- Text space:
  - Our runnable machine code.
  - Stored in the a.out
  - The processor loads these instructions and steps through line by line
- Data segment:
  - Elements of fixed size that are known at compile time
  - Stored in the a.out
  - E.g. string literals
  - Values cannot be changed (we will see an unexpected gotcha in a few slides!)
- BSS (Block Started by Symbol) segment:
  - Space to hold “static” variables without initial values.
  - Filled with zeros at run time
  - Not actually stored in a.out, but only the size needed, since there is no initial data
  - Changeable

Elements that change in size

- Stack:
  - Space to store the local variables within each function.
  - Stack “frames” are created when the function is called, and removed when the function returns.
  - NOTE: This means stack variables, including arrays are temporary and we should not keep pointers to them (another gotcha incoming!)
- Heap:
  - Space for the programmer to control dynamically
  - Where we are allowed to request the most available resources
  - Grows and shrinks at our request (we will see the functions in C soon)
  - In object oriented languages, used for ”new” objects

We will leave these for later…

- Kernel
  - Yes, we get to see a copy of a portion of the kernel in each process.
  - This is a ”trick” of the operating system to give us low-level functionality
  - To be covered in a few more weeks
- Memory mapping:
  - Likewise, a bit more advanced than where we’re at
  - It’s for access to files and libraries that the Operating System connects us to

Static (global), stack, and heap memory

- 3 file in the ExampleCode folder on GitHub:
  - auto_array_alloc.c
  - static_array_alloc.c
  - dynamic_array_alloc.c

- We'll look through them in order

Automatically Sized Arrays

- These arrays live on the stack
- The size limit is typically quite small. Good for small stuff, but we need a way to ask for more memory!

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

int main(){

        long int automatic_array_size = 2;
        while(1){

                char automatic_array[automatic_array_size];

                printf( "I managed to allocate a string of size %ld.\n", automatic_array_size );

                automatic_array[automatic_array_size-1] = '2';
                printf( "The last element holds %c.\n", automatic_array[automatic_array_size-1] );

                automatic_array_size *= 2;
        }

        return 0;
}
```

Automatically sized arrays

- Typical output:

I managed to allocate a string of size 262144.
The last element holds 2.
I managed to allocate a string of size 524288.
The last element holds 2.
I managed to allocate a string of size 1048576.
The last element holds 2.
I managed to allocate a string of size 2097152.
The last element holds 2.
I managed to allocate a string of size 4194304.
The last element holds 2.
Segmentation fault: 11

Stack memory limited, and there is another “gotcha” to keep in mind

- When creating pointers, we must think about the stack push-pop behaviour. Similar example on Github: “stack_pointer_gotcha.c”

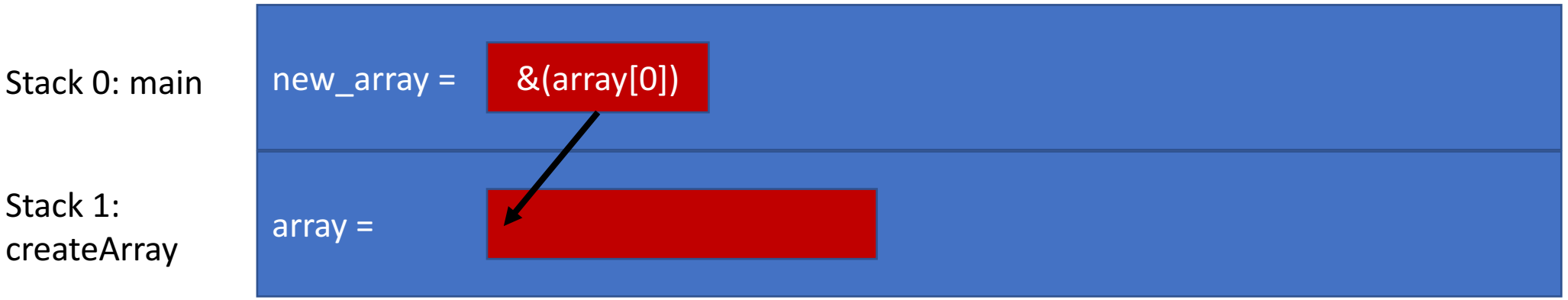
```
char* createArray( size ){
  char array[size];
  return array;
}
int main() {
  char *new_array = createArray( 10 );
  new_array[0] = ‘a’; // THIS LINE CAN SEGFAULT
}
```

What's happening here?

- The common error "returning pointer to stack memory"

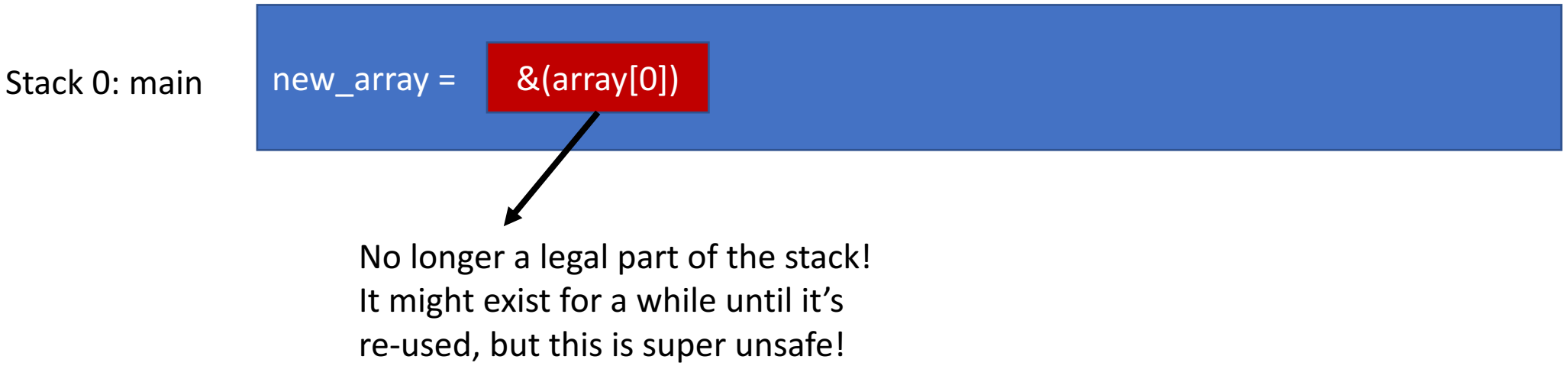
What’s happening here?

- The common error “returning pointer to stack memory”



What's happening here?

- The common error "returning pointer to stack memory"



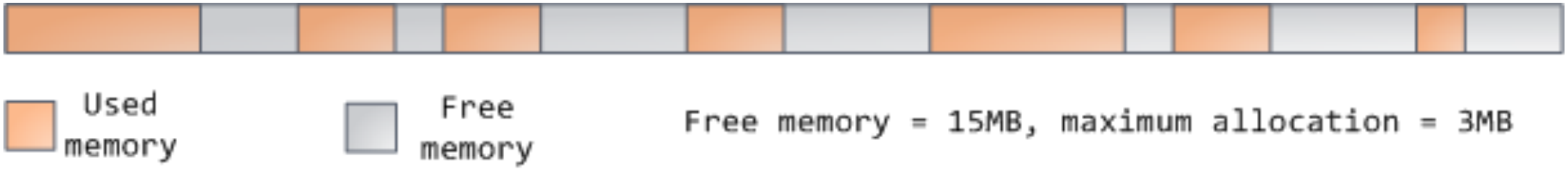
One way to fix: Statically Sized Arrays

- Test with “gcc –DSTATIC_ARRAY_SIZE=number
- You can typically get away with much larger sizes, even to the point where you cannot practically use the memory provided!
- But what if we didn’t know the size in advance?

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

static char static_array[STATIC_ARRAY_SIZE];

int main(){

        printf( "I managed to statically allocate a string of size %ld.\n", (long int)STATIC_ARRAY_SIZE );

        for( long int pos=0; pos<(long int)(STATIC_ARRAY_SIZE); pos++ )
                static_array[pos] = '2';

        printf( "The last element holds %c.\n", static_array[(long int)(STATIC_ARRAY_SIZE-1)] );
        int input;

        // This loop "pauses" execution so we can see the impact on our computer
        while( (input=getchar()) != 'q' ){
                usleep(10);
        }

        return 0;
}
```

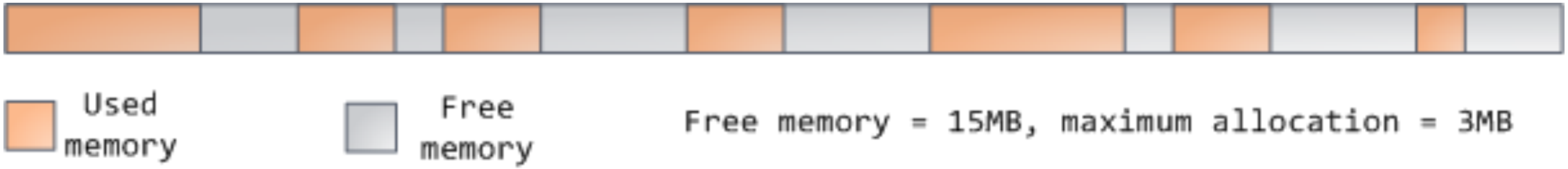
The Heap: Dynamically Allocated Memory

- Provides flexible, persistent memory across function calls
- Example: function “char* createImageData(int width, int height)” should return a pointer to new memory that can hold pixel data.
  - Incorrect to point to a stack variable!
  - If we don’t know the size of our bunny ears in advance, cannot use static!
- Our next low-level programming tool: ask for heap memory through the C dynamic memory functions
  - We will follow this path right to the Linux kernel today!



The Heap: Dynamically Allocated Memory

- How does the OS decide where to locate our new memory within the heap? How does it keep things organized (or does it?) What is the most memory we can possibly use and why is it larger than what my computer's maker reported (e.g., 4GB RAM)?
- These are interesting questions that we will ignore for now
  - Covered in COMP 310 where you often have to code a heap allocator/de-allocator



Asking for Heap Memory

- Request for N bytes of heap memory (not initialized):

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

- Request for an array of `N` elements each with `size` bytes, and initializes the values all to 0:

```
void *calloc(size_t num, size_t size_of_each);
```

Allocating Useful Types

- malloc and calloc return a void pointer `(void *)`
- It must be cast before it can be de-referenced:

```
int *a = (int *) malloc( sizeof(int) * 40 ); // OR
int *a = (int *) calloc(40, sizeof(int));
```

- The `sizeof()` function simplifies the allocation of memory by calculating the size of the provided data type.

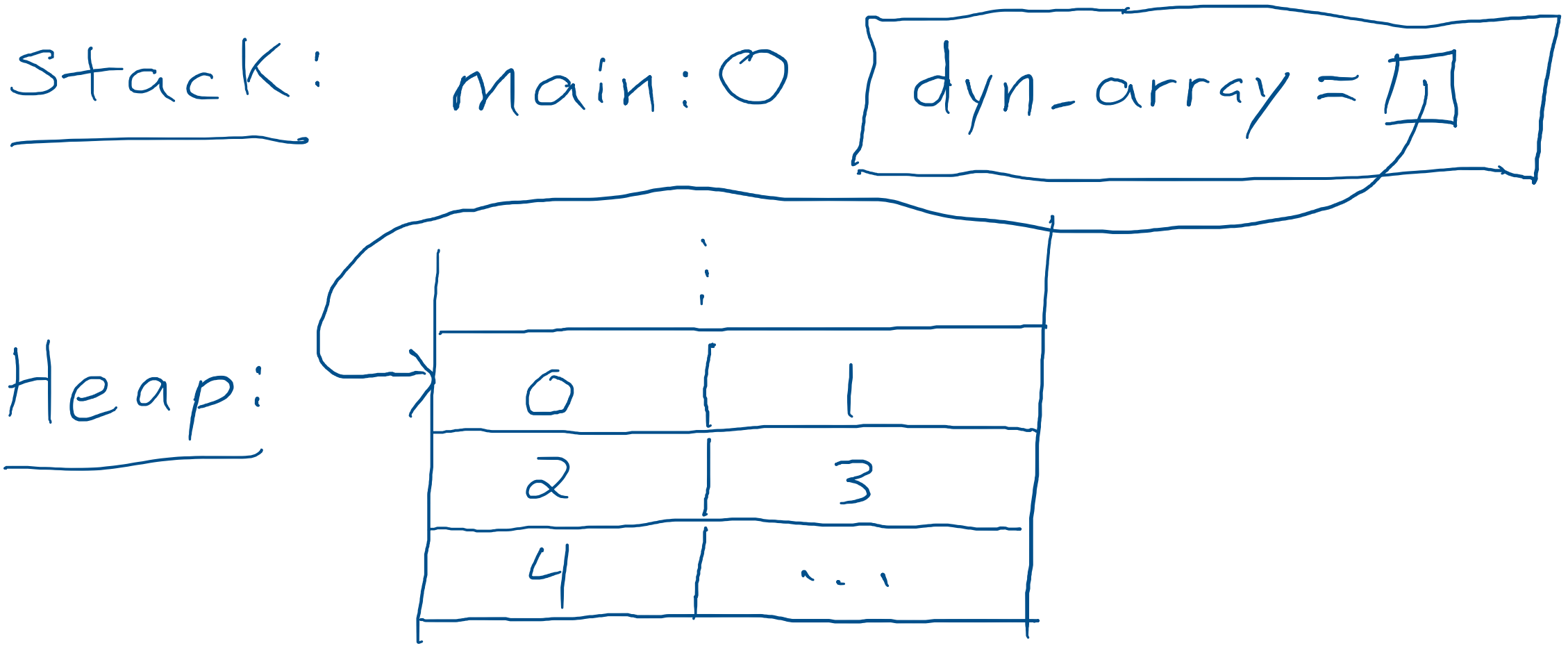
Rules to follow for malloc

- You have 3 TYPEs to fill in:
  - TYPE1 variable_name = (TYPE2)malloc(sizeof(TYPE3)*number);
  - int *pi = (int*)malloc(sizeof(int) * 1);
- RULE1: TYPE1 matches TYPE2, both are pointer types:
  - The point of casting is to tell C how we plan to interpret the heap memory allocated for us by malloc. Malloc returns void* so that it can handle any type. Since this is initially empty memory, casting is always safe.
- RULE2: TYPE3 is the de-referenced version of TYPE2
  - “One less star”
  - When we de-reference the variable with “*variable_name”, C will assume the memory is of the pointer’s underlying type

Malloc

Malloc visual example

```
int *dyn_array = (int*)malloc(5*sizeof(int));
```



Using malloc correctly

- Allocate a single integer on heap:
  - int *pi = (int*)malloc(sizeof(int));
- Allocate an array of 10 integers on heap:
  - int *my_numbers = (int*)malloc(10*sizeof(int));
- Allocate a single integer pointer on heap:
  - int **ppi = (int**)malloc(sizeof(int*));
  - This is our first time seeing a double pointer. It will be explained in due order, but note it’s nothing scarier than single pointers. Just follow the arrow twice!

Common malloc errors (these are all considered “wrong” for 206)

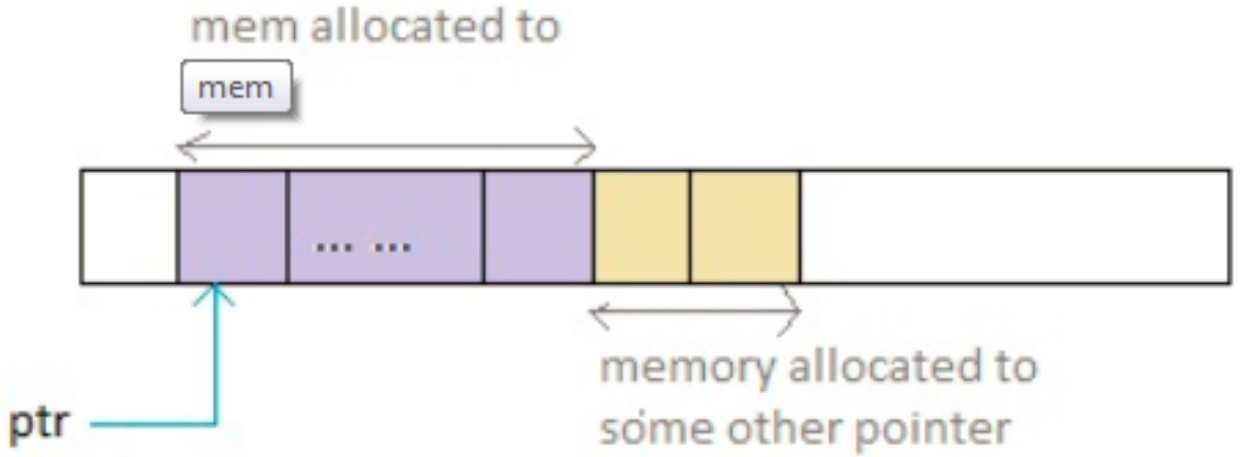
- Mismatch between sizes:
  - int *pi = (int*)malloc(10*sizeof(char));
- Not casting to pointer:
  - int i = (int)malloc(sizeof(int));
- Forgetting sizeof the datatype:
  - int *my_array = (int*)malloc(10);

What if you change your mind?

- realloc changes the size of memory (size is the new total size, not added to the old request)

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

- Note that the heap might not contain enough space to simply expand at this address. So, be aware the data can be copied to a new location, changing the pointer:



Dangerous example

```
int * ptr1 = (int*) malloc(5 * sizeof(int));
int * ptr2 = ptr1;
ptr2 = (int*) realloc(ptr2, 10 * sizeof(int));
// ptr1 may become a dangling pointer
```

This is another thing C leaves up to you as a programmer. If you plan to keep around multiple pointers to dynamic memory, it is your job to keep them all consistent across, allocation and re-allocation. Two common approaches:

1) Never re-allocate (just plan ahead and malloc enough)
2) Be very careful and use a debugger (coming up soon!)

Last step: always!

- For every dynamically allocated section in memory, ensure to run free when finished using:

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

- The pointer must be to heap memory -> that allocated by malloc, calloc or realloc
- Signals the operating system this space can now be used again
- Good practice, ptr =NULL; immediately after free always. Ensures you don’t forget and mess with memory of another variable.

Back to our original goal, fill up the whole memory with ’2’s (ultimate midterm studying procrastination!)

- Recall, stack memory was very limited and global variables needed a size at compile time.

- The solution is to make calls to malloc.

- Let’s see how much memory we can ask for!

Dynamically Sized Arrays

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

int main(){

        long int dynamic_array_size = 2;
        char *dynamic_array;

        while(1){

                dynamic_array = (char*)malloc( dynamic_array_size*sizeof(char) );

                if( dynamic_array == NULL ){
                        printf( "I failed to allocate string of size %ld.\n", dynamic_array_size );
                        break;
                }

                printf( "I managed to allocate a string of size %ld.\n", dynamic_array_size );

                dynamic_array[dynamic_array_size-1] = '2';
                printf( "The last element holds %c.\n", dynamic_array[dynamic_array_size-1] );

                free(dynamic_array);
                dynamic_array_size *= 2;
        }

        dynamic_array_size /= 2; // The last one that worked
        dynamic_array = (char*)malloc( dynamic_array_size*sizeof(char) );

        if( dynamic_array != NULL ){
                printf( "OK, populating an array of size %ld... how long will this take?\n",
dynamic_array_size );
                for( long int pos=0; pos<dynamic_array_size; pos++ ){
                        dynamic_array[pos] = '2';
                }
        }

        return 0;
}
```

Wrap-up

- We have now experienced a few different ways that we can control the memory of our program. We will continue to expand this through the term!

Exercises and Reading

- K&R text chapter 5
- Practice will come with A3