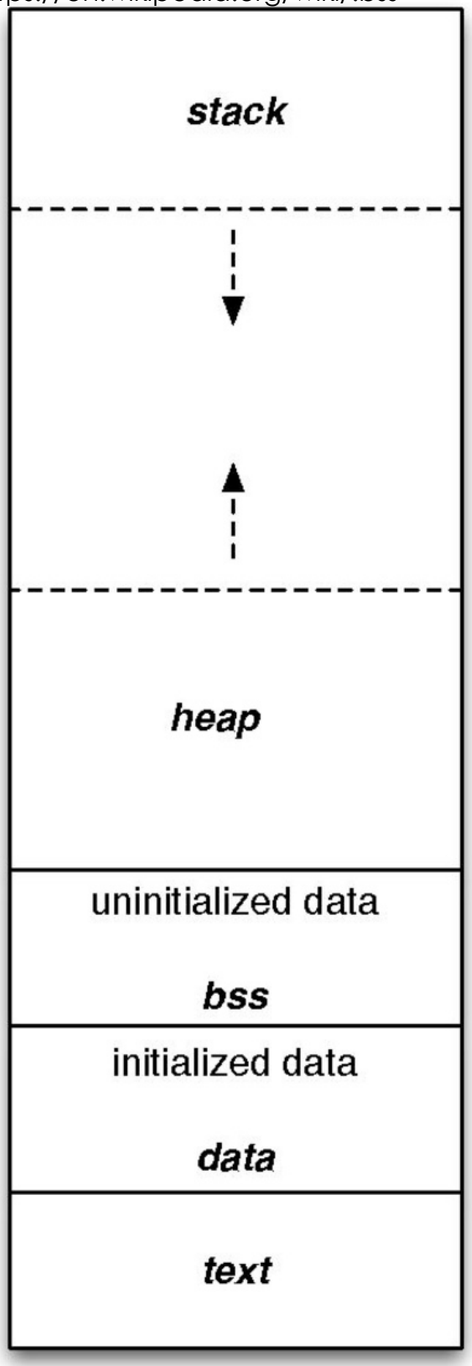
STATIC KEYWORD

```
static int instances = 0;
```

https://en.wikipedia.org/wiki/.bss

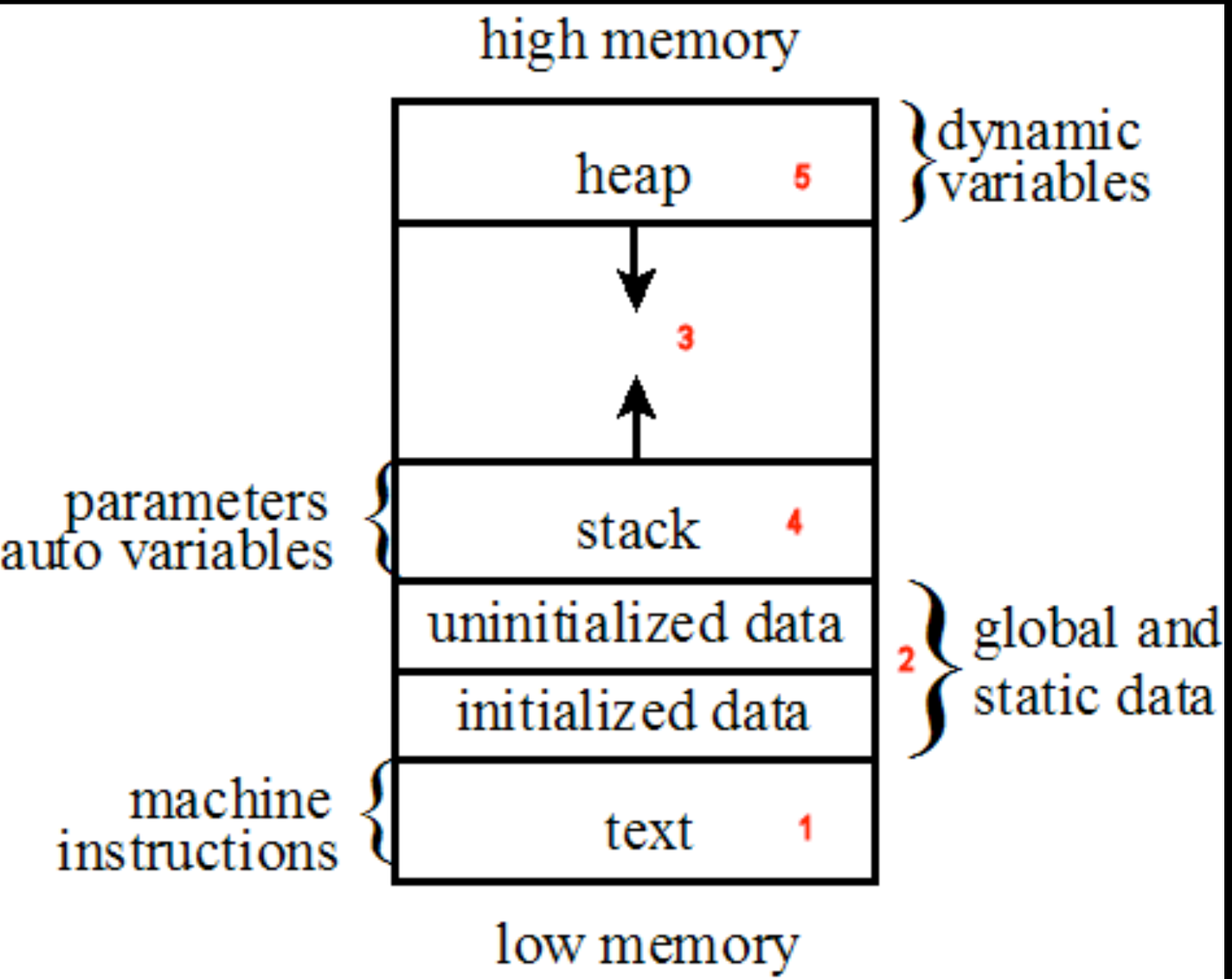


WHAT IS A STATIC VARIABLE?

- A static variable is one that is in memory from program launch to tear down (the program's lifetime)
- It is stored in the BSS segment of your program when uninitialized and DATA when initialized (see left)
- It will persistently stay in memory whether you:
  - Use the function it is defined in
  - or Use the variable
- The scope is unaffected by declaring a variable as static
- *Note on diagram: At WSU, we will show that the heap and stack at swapped (see diagram on next page)*

WHERE ARE STATIC VARIABLES STORED?

https://icarus.cs.weber.edu/~dab/cs1410/textbook/4.Pointers/images/layout.png



- Dynamic variables are stored in the heap (malloc, calloc)
- Stack is where parameters, local vars, and call records are
- Uninitialized & initialized are where static and global variables are stored
  - As we can see, this will not change so we ideally want minimal (if any) global and static data
  - If you need a global variable, rethink your approach – there is possibly a better way

WHEN SHOULD I USE A STATIC VARIABLE?

- When we need to
  - Maintain a variable among multiple instances of a class (C++)
  - Maintain a value from function call to function call (i.e., strtok)

PROS & CONS

- Pros
  - Allows us to maintain a value in between function calls
  - Allows us to maintain a count of instances of a specific class
- Cons
  - Becomes non-thread safe
    - We either need to write a new function or use locks
  - Uses memory even if we do not touch the function or class (stored in BSS or DATA)
    - Non-static will grow/shrink based on the usage of functions and classes
  - Can become more difficult to debug when there are many static vars

RESOURCES TO LOOK AT FOR FURTHER DETAIL

- https://en.wikipedia.org/wiki/Data_segment
- https://en.wikipedia.org/wiki/.bss
- https://www.gnu.org/software/c-intro-and-ref/manual/html_node/Static-Functions.html

STRTOK

#include <string.h> (C)

#include <cstring> (C++)

HOW TO CALL STRTOK

```
char * input = ”everything is a string”;
char * token;

token = strtok(input, “ “);
printf(“%s\n”, token);

while ((token = strtok(NULL, “ “)) != NULL) {
      printf(“%s\n”, token);
}
```

HOW TO CALL STRTOK (COMMENTED)

```
char * input = ”everything is a string”; // input
char * token; // where we will store the token

token = strtok(input, “ “); // start strtok (static pointer in strtok is set to input)
printf(“%s\n”, token); // process token

while ((token = strtok(NULL, “ “)) != NULL) { // set to the next token, and run if non-null
      printf(“%s\n”, token); // process token
}
```

STRUCT KEYWORD

Look at https://github.com/swiftlydesigner/121-Practice-Review/blob/main/Labs/Fall%20'23/Lab-11/Lab%2011.pdf

121-Practice-Review > Labs > Fa ‘23 > Lab-11

MALLOC

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

HOW TO CALL

```
int * randomNumbers = (int *)malloc(sizeof(int) * 100);
if (!randomNumbers) {
      // handle error and stop execution. (not allocated)
}
memset(randomNumbers, 0, sizeof(int) * 100);
// populate with rand nums

// use as needed

free(randomNumbers);
```

HOW TO CALL (ENGLISH DESCRIPTION)

1. Assign a variable to the call to malloc, casting to the pointer type (i.e., int*)
    1. If we want to allocate a node, then the size will be `sizeof(NAME_OF_STRUCT)`
2. **Check if it was allocated!**
3. Use memset to zero out all numbers or a for loop to set a default value
    1. We do this to insure we do not read garbage values
    2. If you are going to write data to all addresses, 3 can be skipped
4. Process it as you would any other variable
5. Once you no longer need the allocation, call free
    1. Malloc and free are likely to be in different functions
        1. Still, they must have a 1:1 call ratio

FUNCTIONS

- malloc `ptr = malloc(SIZE_OF_TYPE * NUM_OF_ELES);`
  - Use when we want to allocate anything (although calloc is sometimes preferred)
- calloc `ptr = calloc(NUMBER_OF_ELES, SIZE_OF_TYPE);`
  - Use when we have an array to allocate. Format is easy to understand
- realloc `oldPtr = realloc(oldPtr, NEW_SIZE);`
  - Use when we need to resize a previous allocation
- free `free(ptr);`
  - Must be called once for every time malloc is called.
  - Any allocated memory must be freed.
  - If it is not freed, it is called a memory leak. Inaccessable allocated memory