

CS214-system programming

- Section 03/08 recitation 02

Yunhe Gao

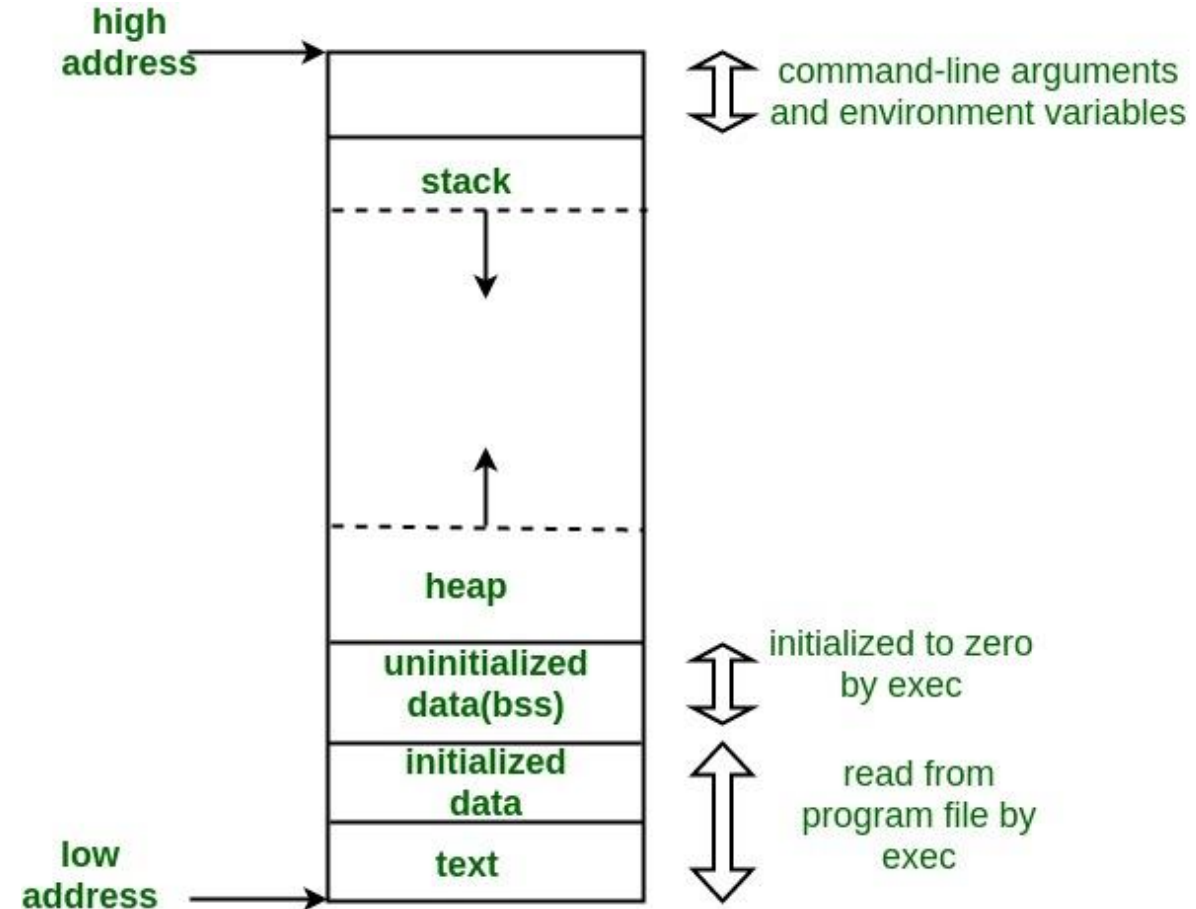
yg397@scarletmail.rutgers.edu

Content

- Memory Layout
- Dynamic Memory management
- Compile Process
- Header file and Macro

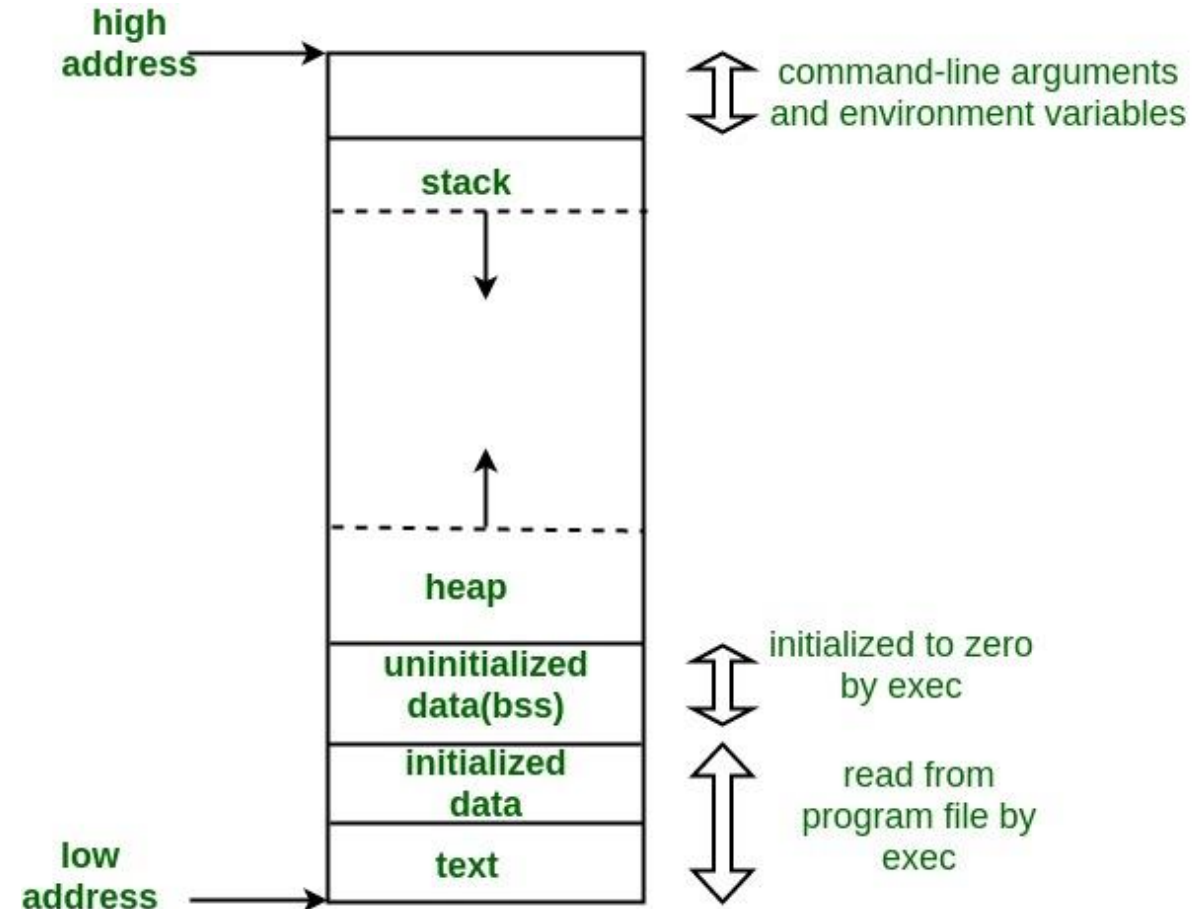
Memory Layout of C

- Text segment
- Initialized data segment
- Uninitialized data segment
- Stack
- Heap



Text Segment

- Also known as a **code segment**, is one of the sections of a program in an object file or in memory, which contains **executable instructions**.
- As a memory region, a text segment may be placed below the heap or stack in order to prevent heaps and stack overflows from overwriting it.
- Usually, the text segment is sharable so that only a single copy needs to be in memory for frequently executed programs, such as text editors, the C compiler, the shells, and so on. Also, the text segment is often read-only, to prevent a program from accidentally modifying its instructions.

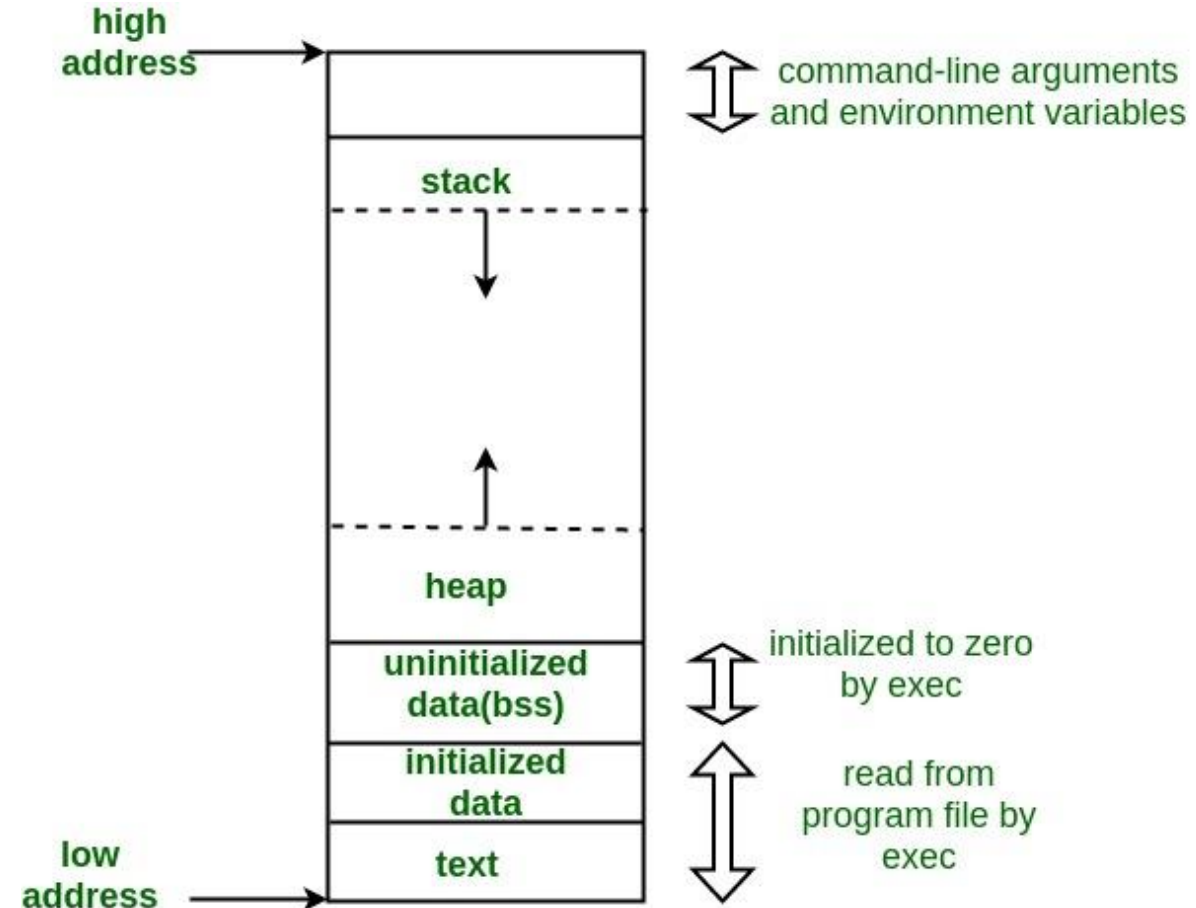


Initialized Data Segment:

- Also named Data Segment, contains the **global variables** and **static variables** that are initialized by the programmer.
- This segment can be further classified into initialized read-only area and initialized read-write area.
- Global and static variables are stored in read-write area, since the values of the variables can be altered at run time. Const variables are stored in read-only area.

example:

```
static int i = 10;  
global int i = 10;
```

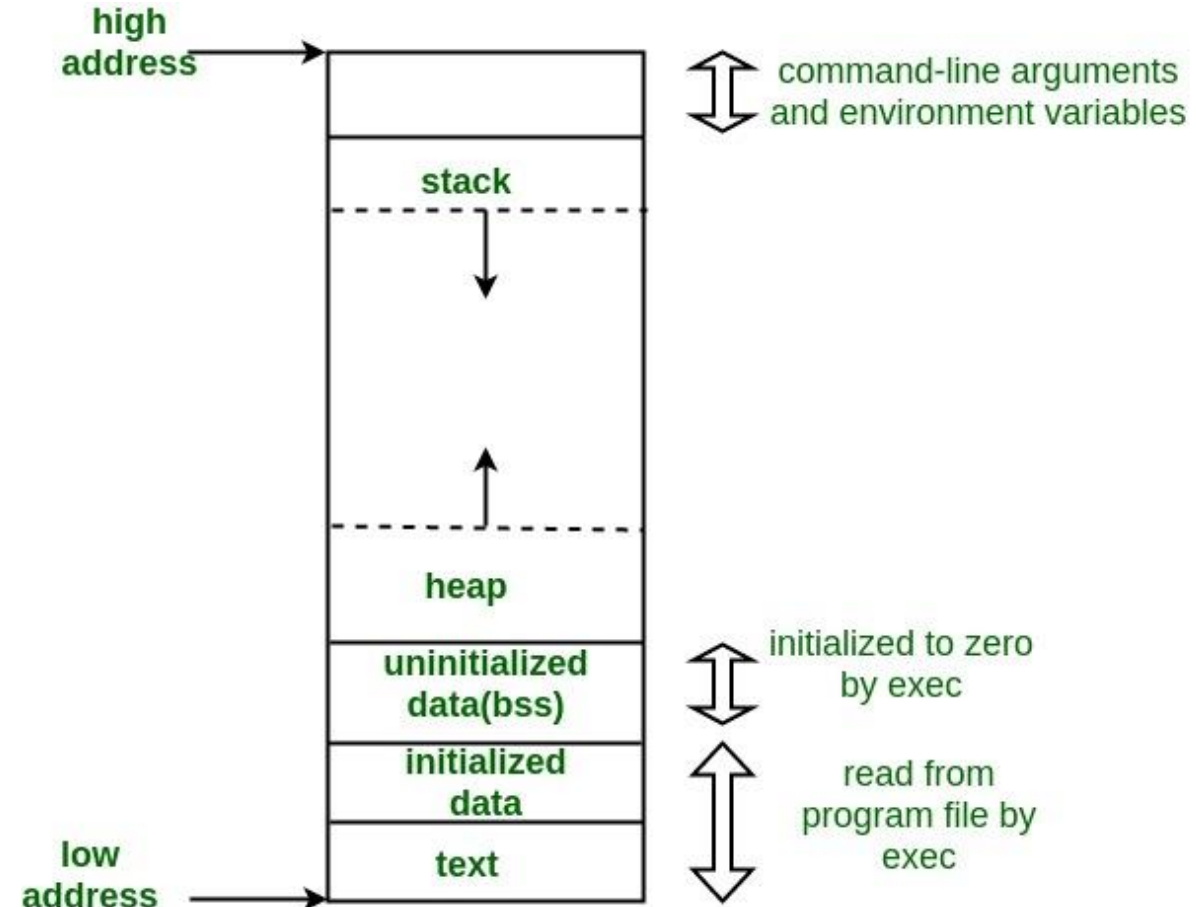


Uninitialized Data Segment

- Also called the “bss” segment, named after an ancient assembler operator that stood for “block started by symbol.” Data in this segment is initialized by the kernel to arithmetic 0 before the program starts executing
- uninitialized data starts at the end of the data segment and contains all global variables and static variables that are initialized to zero or do not have explicit initialization in source code.

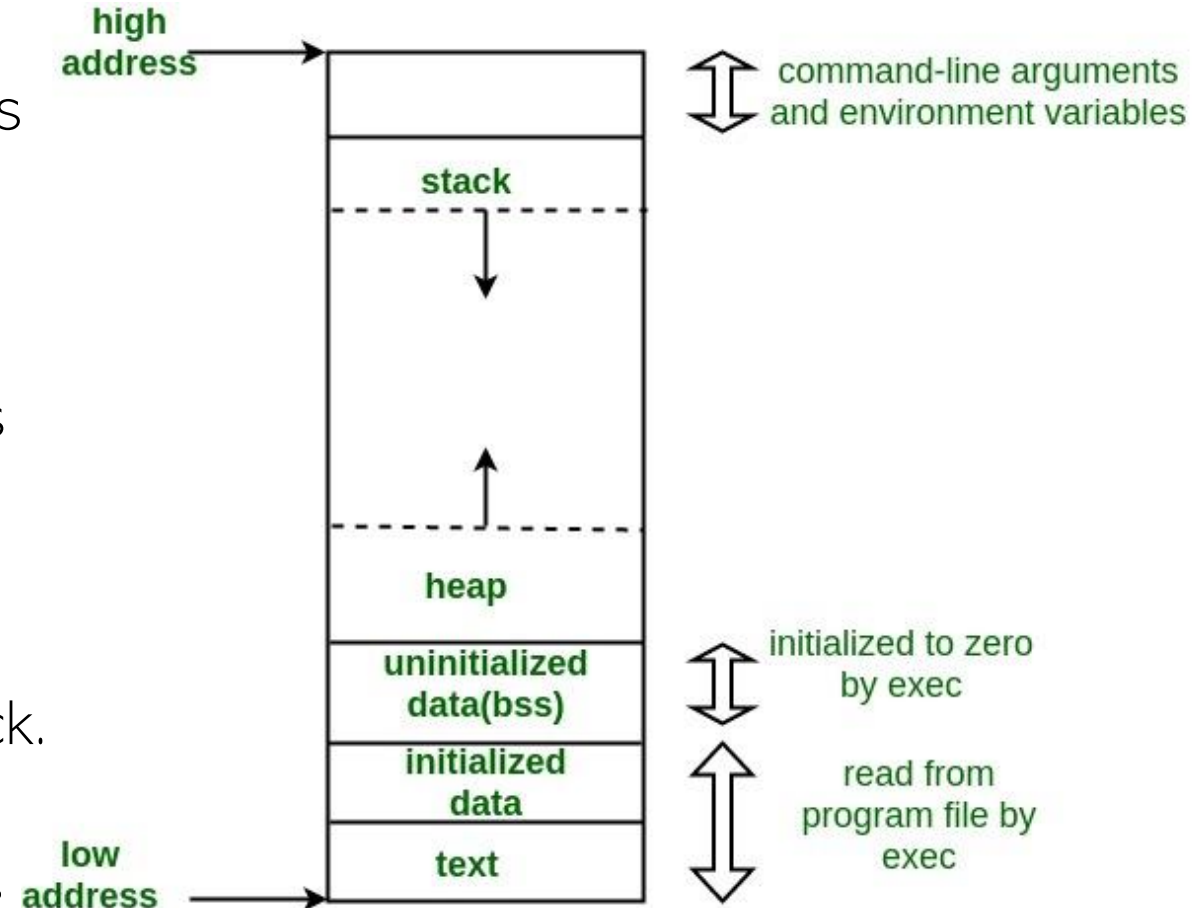
example:

```
static int i;  
global int i;
```



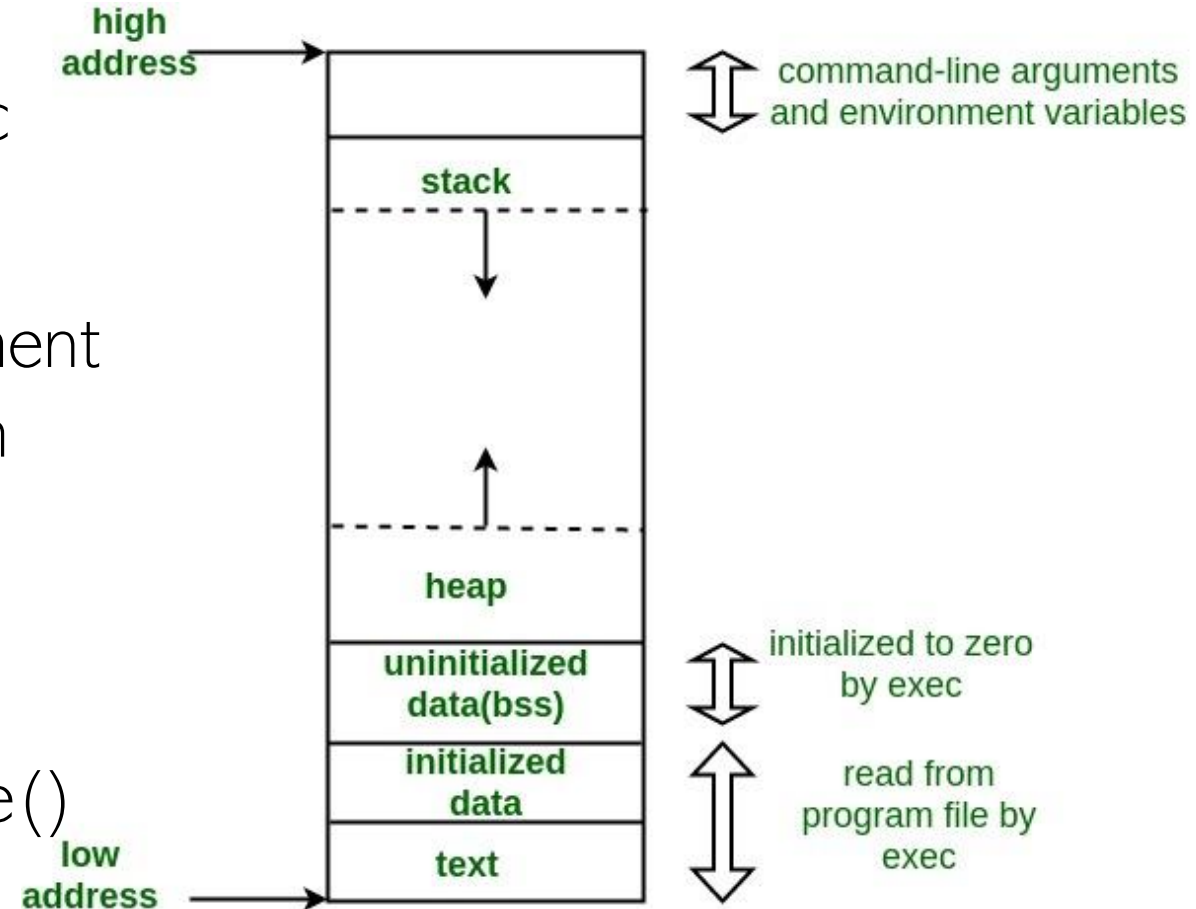
Stack

- Region of memory that temporarily stores **local variables** created by functions (e.g. `main()`, `foo()`)
- The stack is a LIFO data structure. It shrinks and grows throughout a process's execution.
- When a function starts, its variables are pushed in to the stack. When a function exits, its variables are popped off the stack. This is why local variables created in functions are lost outside of that function.
- Managed by the OS



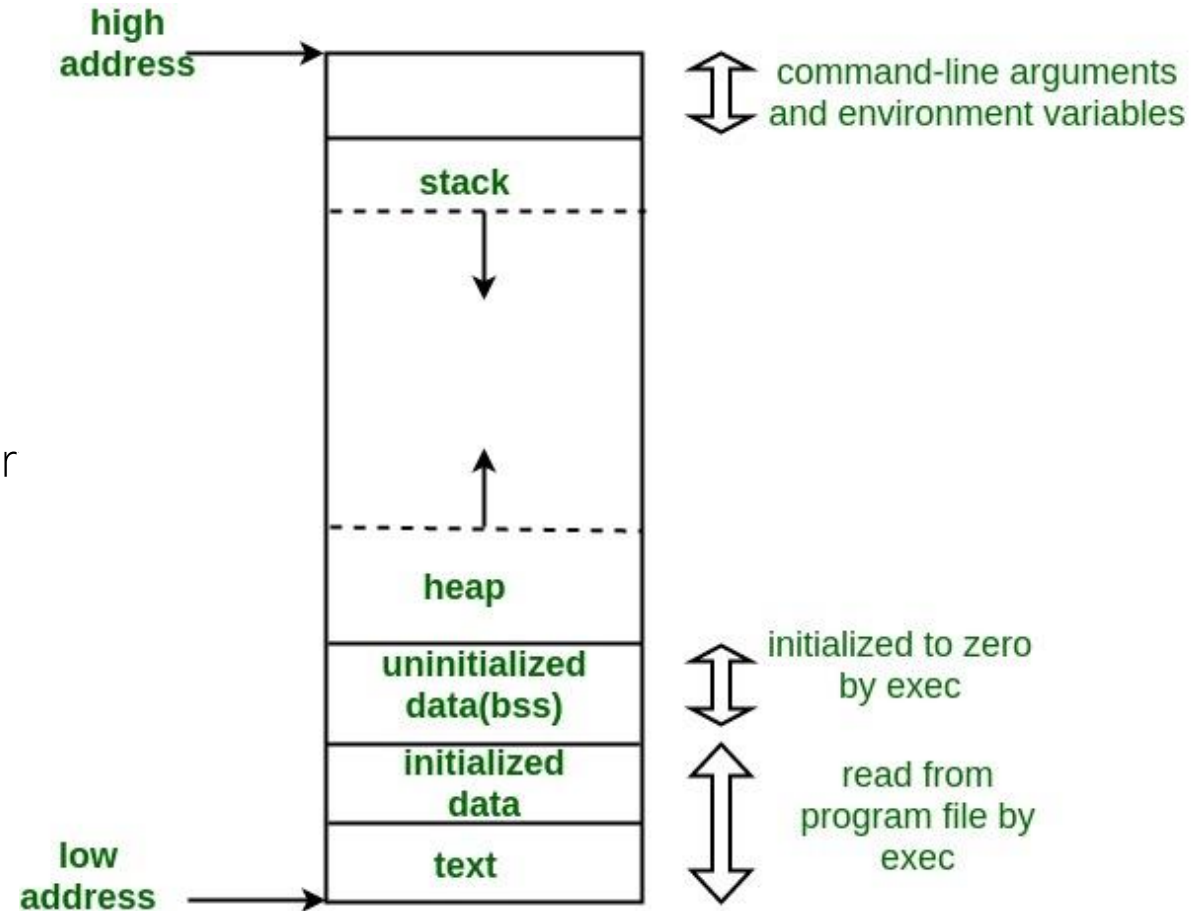
Heap

- Region of free memory for dynamic allocation.
- begins at the end of the BSS segment and grows to larger addresses from there.
- Somewhat user-managed
 - `malloc()`, `calloc()`, `realloc()`, `free()`



Dynamic Memory Management

- Why do we need it?
 - We might not know how big an object needs to be until run-time.
 - E.g. an array of data read from a file.
 - We can declare a very large array, but it's ver inefficient.




malloc

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

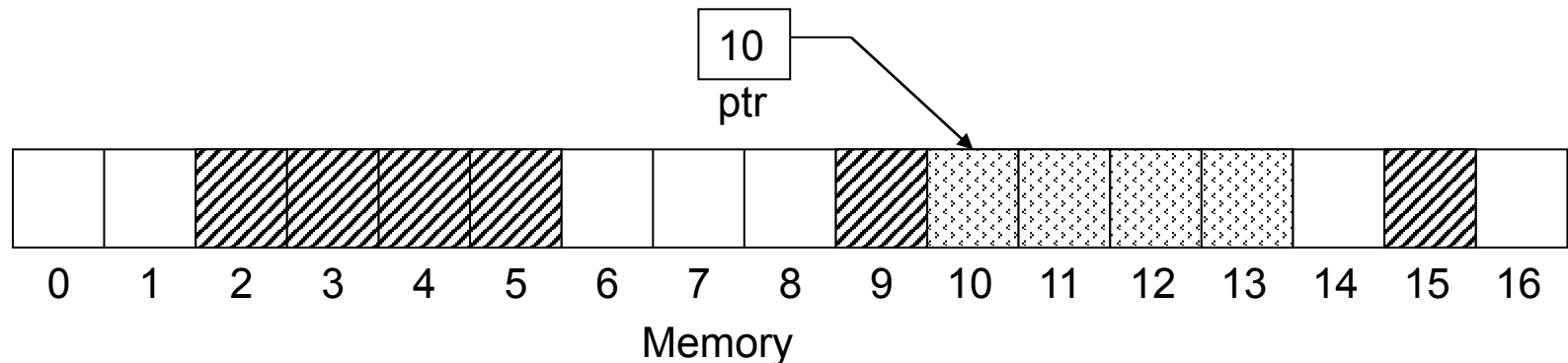
- dynamically allocates memory from the heap
- returns a void pointer, need to specify the type of the pointer before using.
- function searches heap for size contiguous free bytes
- function returns the address of the first byte
- programmer's responsibility to manage the pointer (don't lose it), and not write into area beyond the last byte allocated

```
char *ptr;  
ptr = (char*) malloc(4*sizeof(char));
```

Key

 previously allocated

 new allocation



Void Pointer

- How can we fetch a data item from memory?
 - Memory address
 - Data item size (determined by its data type)
- **Void*** is the generic pointer type
 - The size of `int*`, `double*`, `char*` and etc. are the same
 - The only difference is let the compiler know how to interpret the data in that address
 - A **void*** is a pointer where we don't specify the type of what it points to.
 - We need to explicitly cast to some pointer type before using a **void*** pointer.
- What we can't do with a `void*`
 - Dereference, because we don't know the size of the data it points to
 - Pointer arithmetic, `p++` etc. because we don't know the size
- What we can do with a `void*`
 - Pass it as an argument to a function
 - Receive from a function
 - Cast to some other pointer type

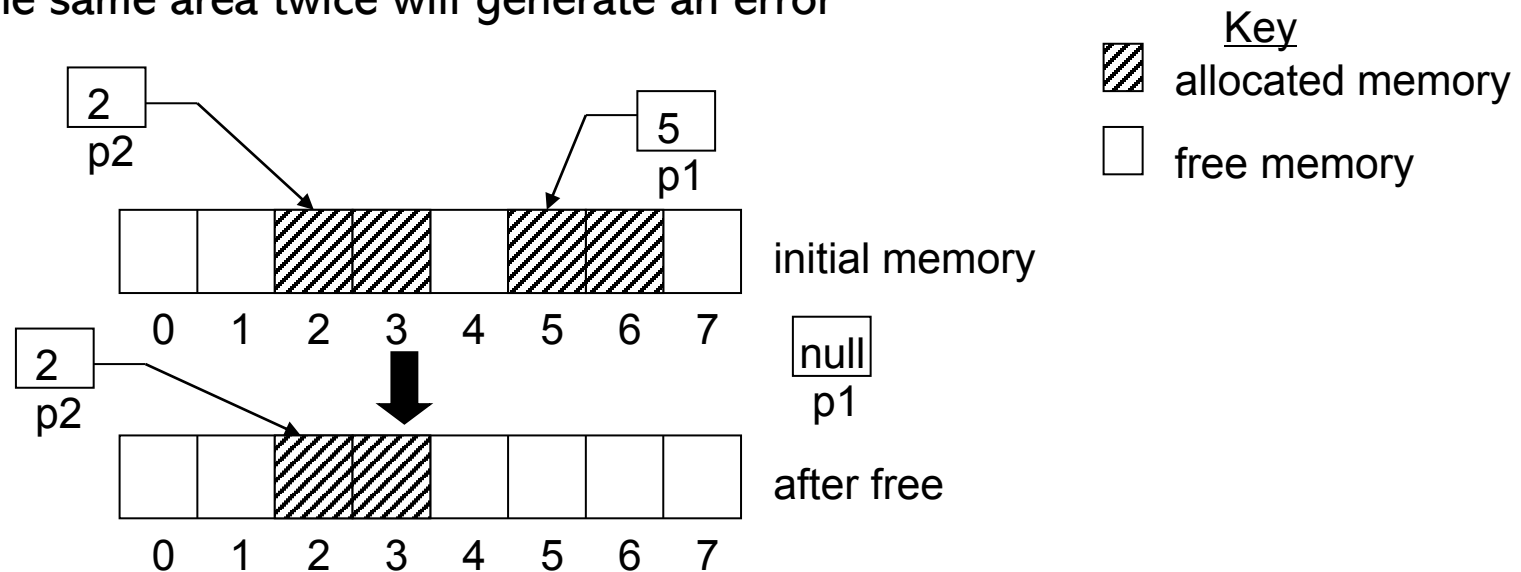
Void Pointer

- Void pointer can “contain any pointer of a specific data type
 - Void pointer cannot be dereferenced directly
-
- `char a = 'a';`
 - `int b = 2;`
 - `void *p = &b; // Allowed`
 - `p = &a; // Still allowed`
 - `printf(“%c\n”, *p); // Cannot compile!`
 - `printf(“%c\n”, *(char*)p); // Good!`

free

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

- release the area pointed by ptr
- must manually free for garbage collection
- ptr must come from malloc
- ptr must not be null
 - trying to free the same area twice will generate an error



Dynamic Memory

- `void *calloc(size_t nmemb, size_t size)`
 - Allocates memory for an array of `nmemb` elements of `size`. The memory is set to zero.
 - Calloc is safer as it has initialization. Malloc is faster as it doesn't initialize
- `void *realloc(void *ptr, size_t size)`
 - Changes the size of the memory block pointed to *ptr* to *size* bytes.

Dynamic Memory

More about realloc function

- If new size < old size
 - the ptr value will not change
- If new size \geq old size
 - the ptr value may change or not
- If new size == 0:
 - Allocated memory will be deallocated and ptr becomes NULL

Common Mistakes

- Using a pointer before allocating heap space

```
int *ptr;  
*ptr = 5;
```

- Changing the pointer, not the value it references

```
int *ptr = (int*) malloc(sizeof(int));  
ptr = 10;    // sets value of pointer to 10, not value on the heap
```

- Forgetting to free space on the heap (memory leak)

```
int *p1 = (int*) malloc(sizeof(int));  
int *p2 = (int*) malloc(sizeof(int));  
p1 = p2;    // making p1 point to p2 is fine, but now you can't free  
             // the space originally allocated to p1
```


Compilation

- Compiler converts a C program into an executable.
- There are four phases for a C program to become an executable:
 1. Pre-processing
 2. Compilation
 3. Assembly
 4. Linking

Compilation

Pre-processing

- The first phase through which source code is passed. This phase include:
 - Removal of Comments
 - Expansion of Macros
 - Expansion of the included files. (< > for standard folder, “ ” for current folder)
 - Conditional compilation

```
gcc -E xxx.c -o xxx.i
```

Compilation

Compiling

The next step is to compile filename.i and produce an intermediate compiled output file **filename.s**. This file is in assembly level instructions.

```
gcc -S xxx.i -o xxx.s
```

Compilation

Assembly

In this phase the filename.s is taken as input and turned into filename.o by assembler. This file contain machine level instructions. At this phase, only existing code is converted into machine language, the function calls like `printf()` are not resolved.

Linking

This is the final phase in which all the linking of function calls with their definitions are done. Linker knows where all these functions are implemented. Linker does some extra work also, it adds some extra code to our program which is required when the program starts and ends.

Macros

- When we use **define** for a constant, the preprocessor produces a C program where the defined constant is searched and matching tokens are replaced with the given expression.

```
#include <stdio.h>
#define max 100
int main(){
    printf("max is %d", max);
    return 0;
}
```

Macros

- The macros can take function like arguments, the arguments are not checked for data type.

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

```
#define INCREMENT(x) ++x
```

\\ INCREMENT(x) can be used for x of any data type.

```
int main(){  
    int x = 10;  
    char *ptr = "IlikeC"  
    printf("%s\n", INCREMENT(ptr));  
    printf("%d\n", INCREMENT(x));  
    return 0;  
}
```

Output:

likeC
11

Macros

- The macro arguments are not evaluated before macro expansion.

```
#include <stdio.h>
#define MULTIPLY(a, b) a*b
int main(){
    printf("%d\n", MULTIPLY(2+3, 3+5));
    return 0;
}
```

// The macro is expended as $2+3*3+5 = 16$

```
#define MULTIPLY(a, b) (a)*(b)
// (2+3)*(3+5) = 40
```

Thanks

- `yg397@scarletmail.rutgers.edu`