CSE 31
Computer Organization

Lecture 8 – C Memory Management

Announcement

- Labs
  - Lab 3 due this week (with 7 days grace period after due date)
    - Demo is REQUIRED to receive full credit
  - Lab 4 out this week
    - Due at 11:59pm on the same day of your next lab
    - You must demo your submission to your TA within 14 days
- Reading assignment
  - Reading 02 (zyBooks 2.1 – 2.9) due **tonight**, 27-SEP and Reading 03 (zyBooks 3.1 - 3.7, 3.9) due 11-OCT
    - Complete Participation Activities in each section to receive grade towards Participation
    - IMPORTANT: Make sure to submit score to CatCourses by using the link provided on CatCourses

Announcement

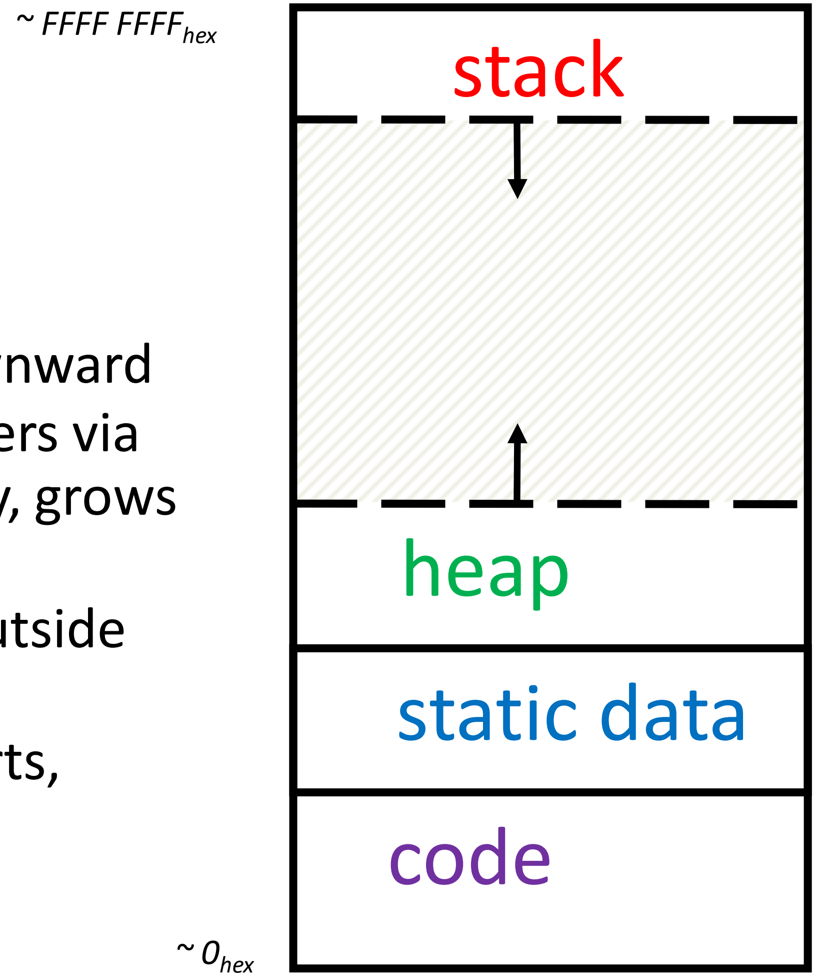
- Homework assignment
  - Homework 01 (zyBooks 1.1 – 1.5) due **tonight**, 27-SEP and Homework 02 (zyBooks 2.1 - 2.9) due 04-OCT
    - Complete *Challenge Activities* in each section to receive grade towards Homework
    - IMPORTANT: Make sure to submit score to CatCourses by using the link provided on CatCourses

C Memory Management

- C has 3 pools of memory (based on the nature of usage)
  - Static storage: global variable storage, basically permanent, entire program run
  - The Stack: local variable storage, parameters, return address (location of “activation records” in Java or “stack frame” in C)
  - The Heap (dynamic malloc storage): data lives until deallocated by programmer
- C requires knowing where things are in memory, otherwise things don’t work as expected
  - Java hides location of objects

Normal C Memory Management

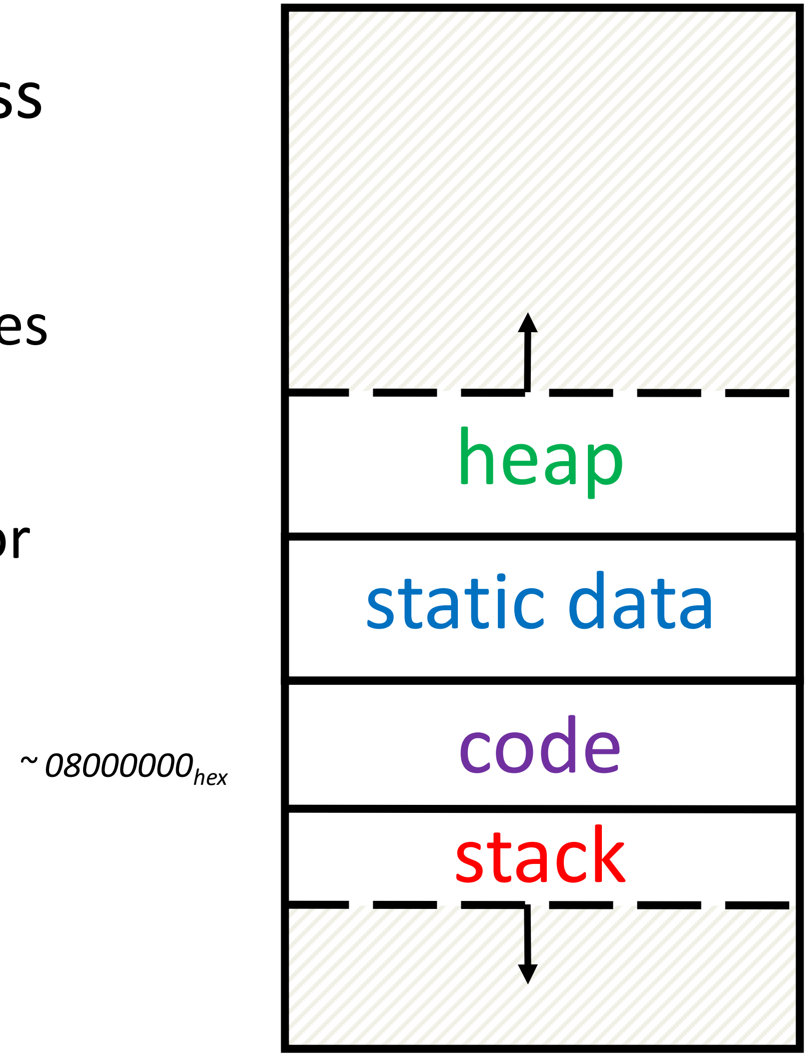
- A program's address space contains 4 regions:
  - stack: local variables, grows downward
  - heap: space requested for pointers via `malloc()` ; resizes dynamically, grows upward
  - static data: variables declared outside main, does not grow or shrink
  - code: loaded when program starts, does not change



For now, OS somehow prevents accesses between stack and heap (gray hash lines). Wait for virtual memory

Intel 80x86 C Memory Management

- A C program's 80x86 address space :
  - heap: space requested for pointers via `malloc()`; resizes dynamically, grows upward
  - static data: variables declared outside main, does not grow or shrink
  - code: loaded when program starts, does not change
  - stack: local variables, grows downward



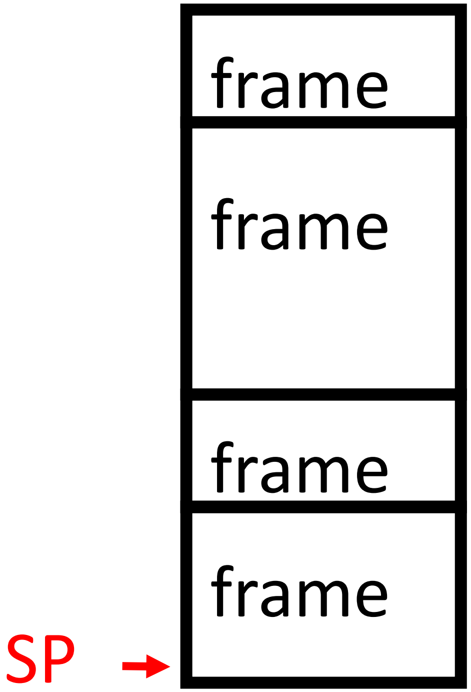
Where are variables allocated?

- If declared outside of any function
  - allocated in “static” storage
- If declared inside of a function
  - allocated in the “stack”
  - freed when a function returns.
    - That’s why the scope is within the function
- Note: `main()` is a function!

```
int myGlobal;
main() {
  int myTemp;
}
```

Stack frames

- Stack frame includes storage for:
  - Return “instruction” address
  - Parameters (input arguments)
  - Space for other local variables
- Stack frames:
  - contiguous blocks of memory for a function
  - stack pointer tells where top stack frame is
- When a function ends, stack frame is “popped off” the stack; frees memory for future stack frames



Stack

- Last In, First Out (LIFO) data structure

stack

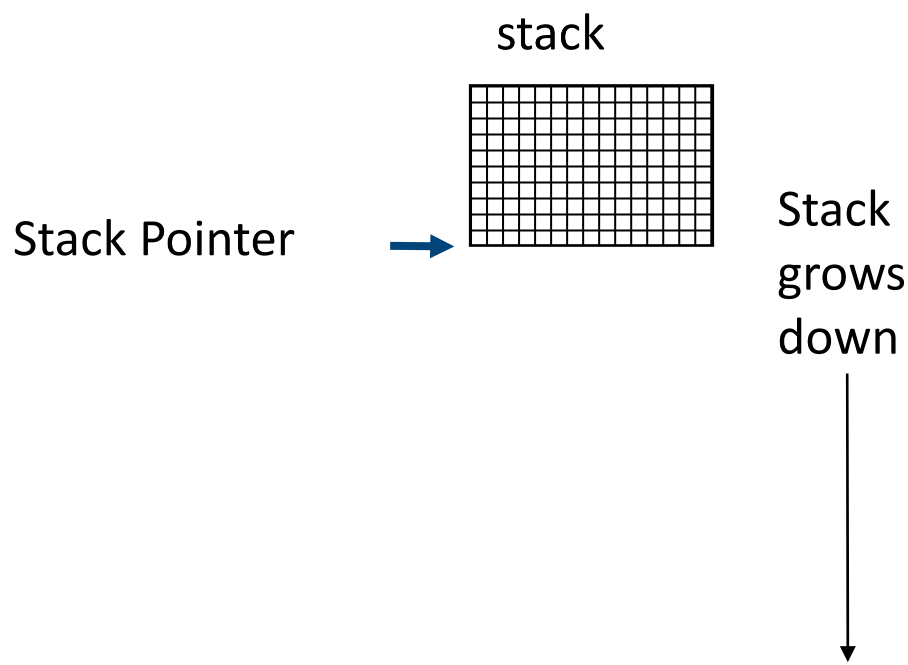
```
main (){
 a(0);
}
void a (int m) {
  b(1);
}
void b (int n){
 c(2);
}
void c (int o){
 d(3);
}
void d (int p) {
}
```

Stack grows down

Stack

- Last In, First Out (LIFO) data structure

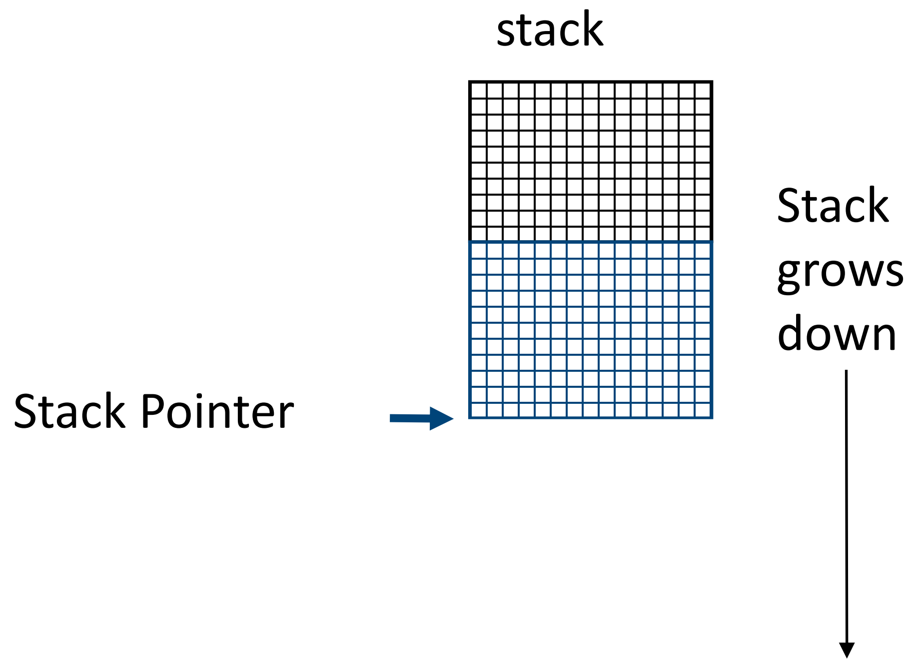
```
main (){
 a(0);
}
void a (int m) {
  b(1);
}
void b (int n){
  c(2);
 }
void c (int o){
  d(3);
 }
void d (int p) {
}
```



Stack

- Last In, First Out (LIFO) data structure

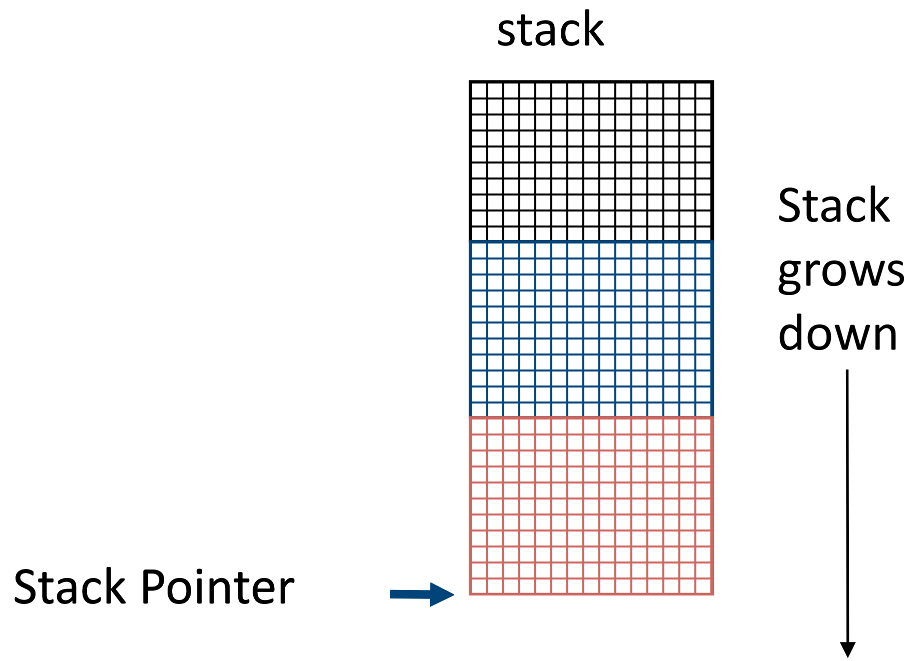
```
main (){
 a(0);
}
void a (int m) {
  b(1);
}
void b (int n){
 c(2);
}
void c (int o){
 d(3);
}
void d (int p) {
}
```



Stack

- Last In, First Out (LIFO) data structure

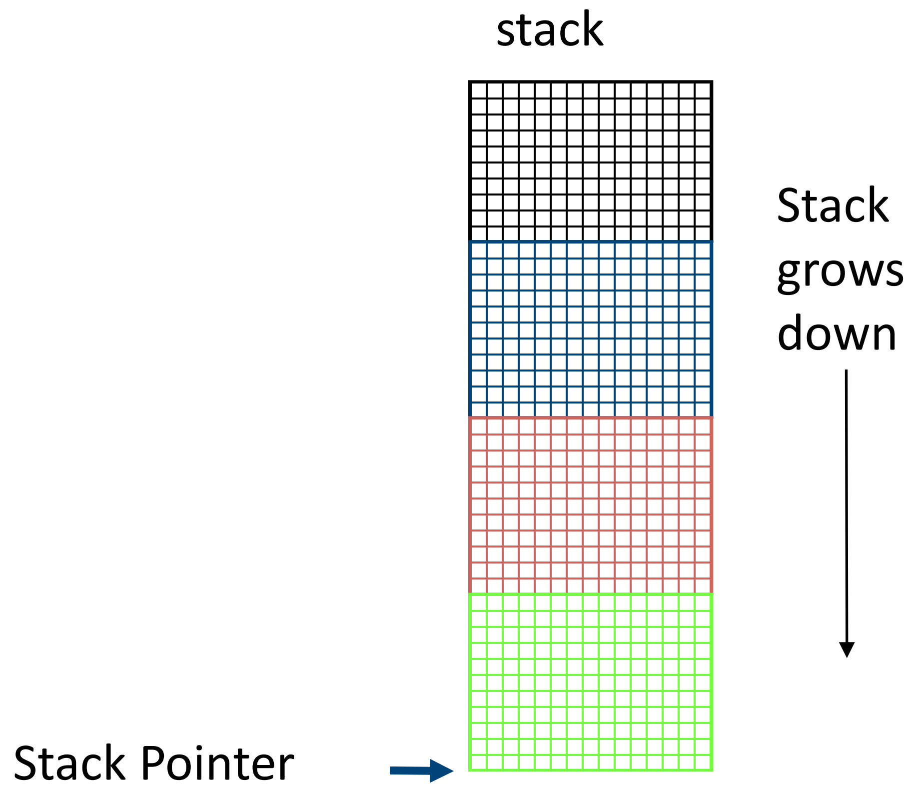
```
main (){
 a(0);
}
void a (int m) {
   b(1);
}
void b (int n){
  c(2);
}
void c (int o){
  d(3);
}
void d (int p) {
}
```



Stack

- Last In, First Out (LIFO) data structure

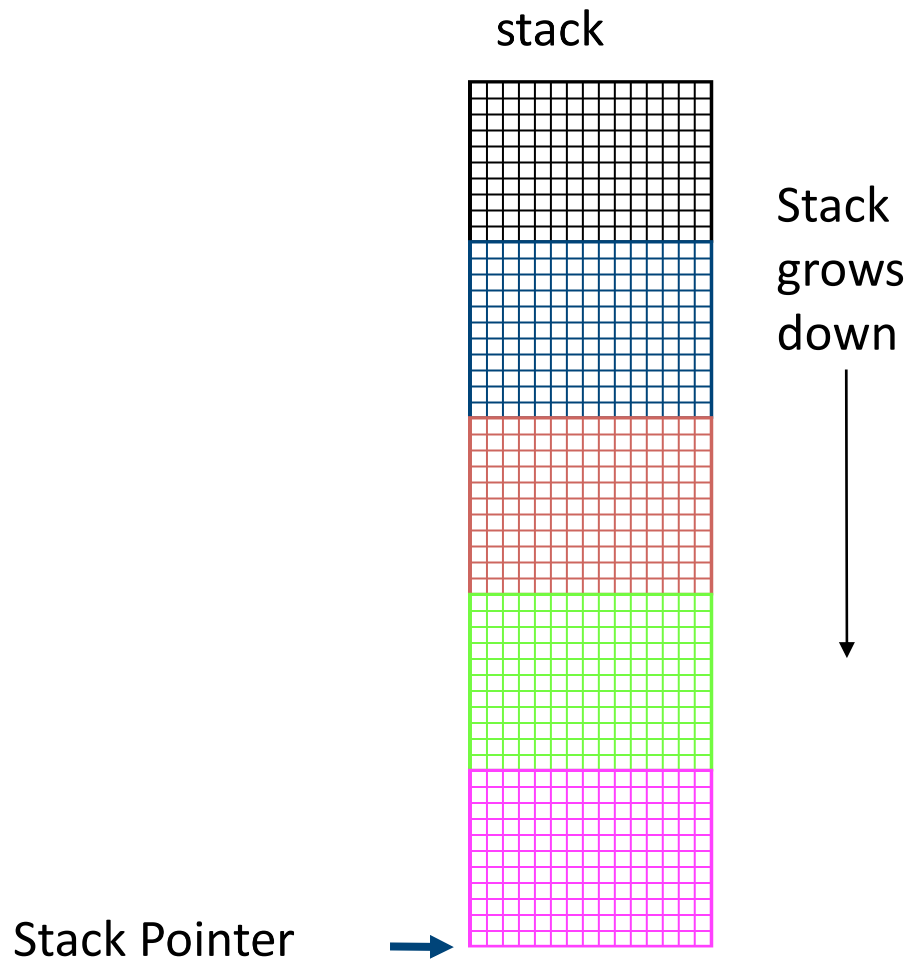
```
main (){
 a(0);
}
void a (int m) {
   b(1);
}
void b (int n){
  c(2);
 }
void c (int o){
  d(3);
 }
void d (int p) {
}
```



Stack

- Last In, First Out (LIFO) data structure

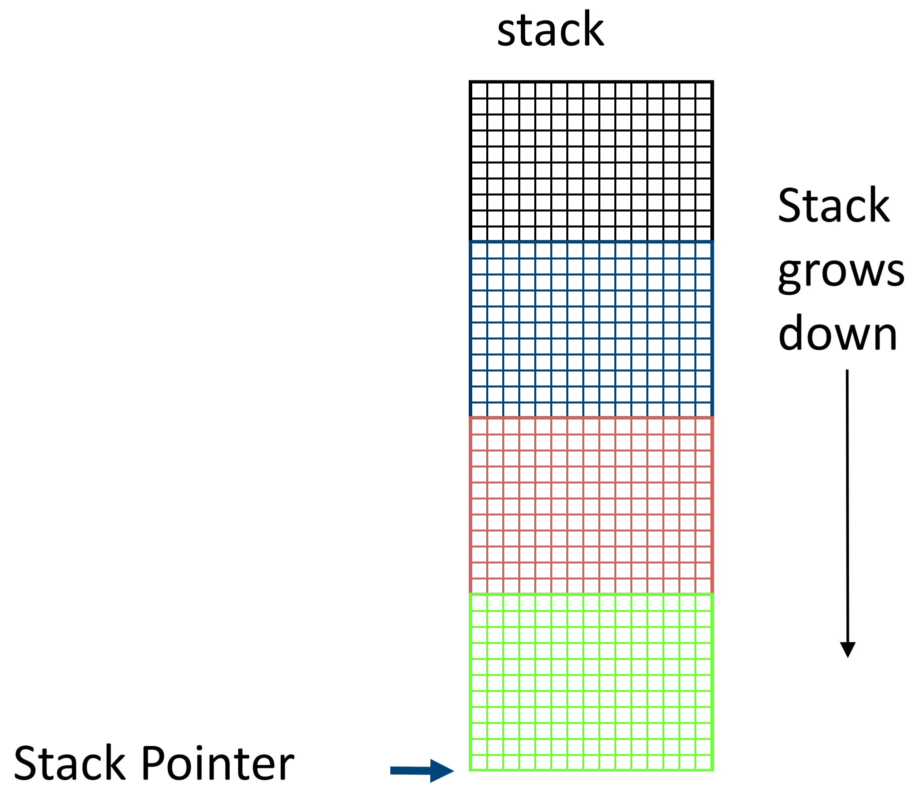
```
main (){
 a(0);
}
void a (int m) {
   b(1);
}
void b (int n){
  c(2);
 }
void c (int o){
  d(3);
 }
void d (int p) {
}
```



Stack

- Last In, First Out (LIFO) data structure

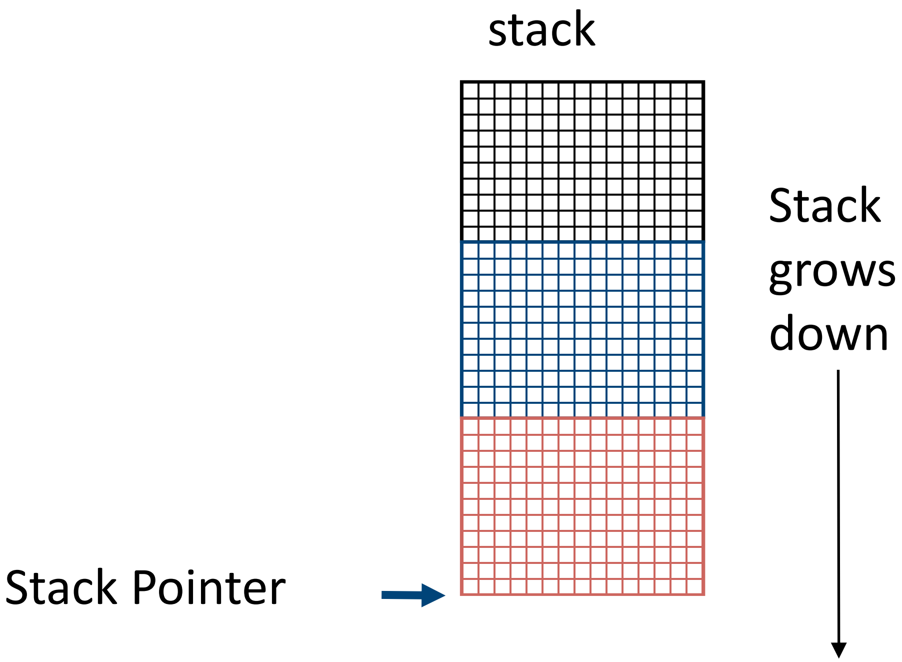
```
main (){
 a(0);
}
void a (int m) {
  b(1);
}
void b (int n){
 c(2);
}
void c (int o){
 d(3);
}
void d (int p) {
}
```



Stack

- Last In, First Out (LIFO) data structure

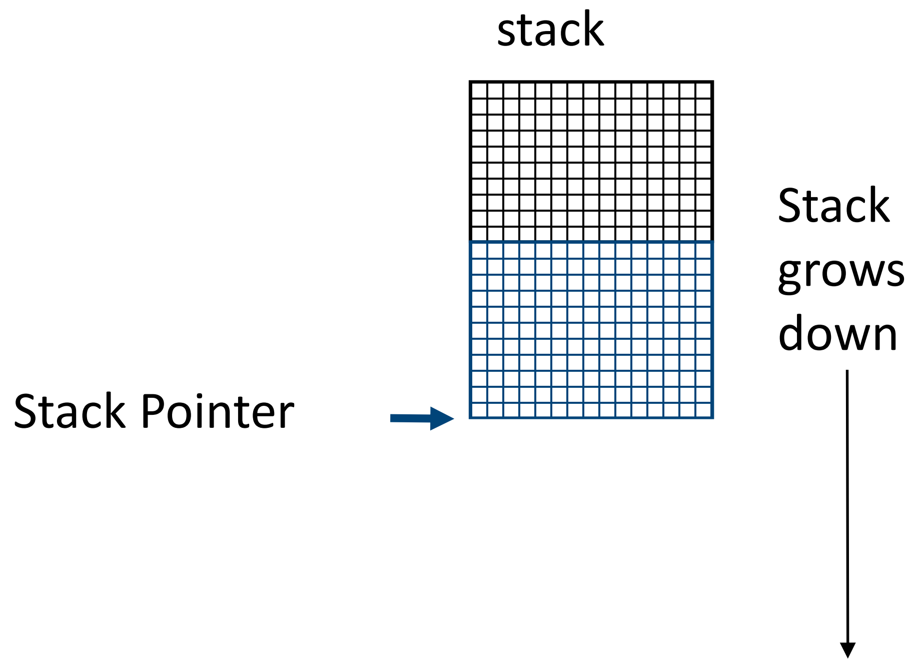
```
main (){
 a(0);
}
void a (int m) {
   b(1);
}
void b (int n){
  c(2);
}
void c (int o){
  d(3);
}
void d (int p) {
}
```



Stack

- Last In, First Out (LIFO) data structure

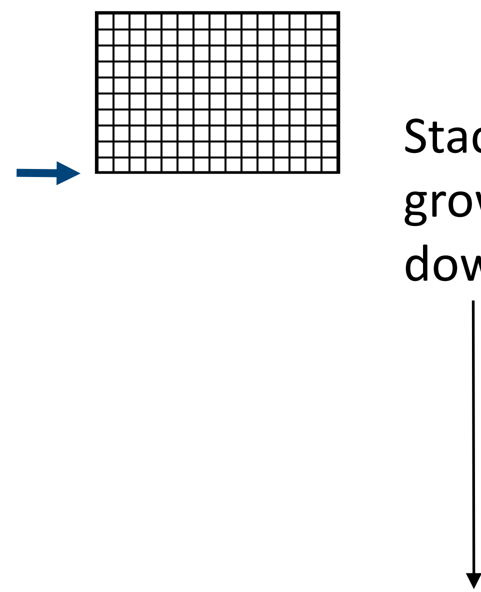
```
main (){
 a(0);
}
void a (int m) {
  b(1);
}
void b (int n){
  c(2);
}
void c (int o){
  d(3);
}
void d (int p) {
}
```



Stack

- Last In, First Out (LIFO) data structure

```
main (){
 a(0);
}
void a (int m) {
  b(1);
}
void b (int n){
 c(2);
}
void c (int o){
 d(3);
}
void d (int p) {
}
```



Stack

- Last In, First Out (LIFO) data structure

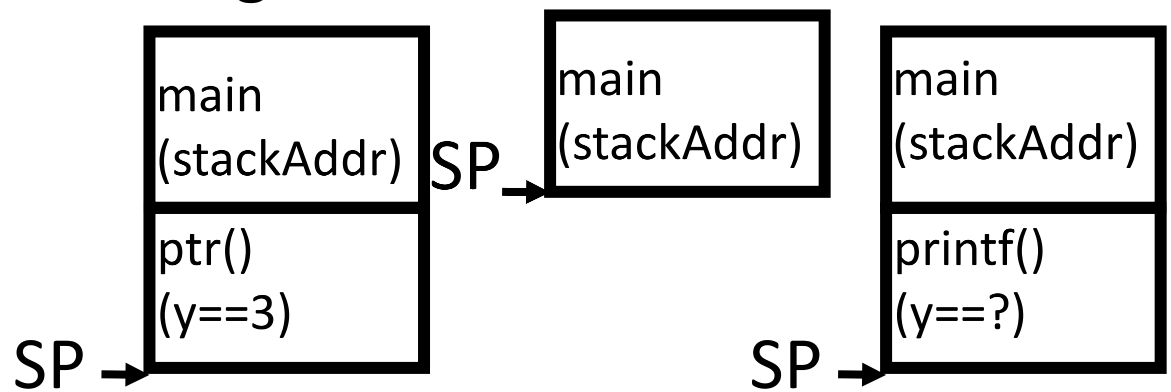
stack

```
main (){
 a(0);
}
void a (int m) {
  b(1);
}
void b (int n){
 c(2);
}
void c (int o){
 d(3);
}
void d (int p) {
}
```

Stack grows down

Who cares about stack management?

- Pointers in C allow access to deallocated memory, leading to hard-to-find bugs !




```c
int *ptr () {
    int y;
    y = 3;
    return &y;
}
```

```c
int main () {
        int *stackAddr, content;
        stackAddr = ptr();
        content = *stackAddr;
        printf("%d", content); /* 3 */
        content = *stackAddr;
        printf("%d", content); /*-2*/
        return 0;
}
```

The Heap (Dynamic memory)

- Large pool of memory, <u>not</u> allocated in contiguous order
  - back-to-back requests for heap memory could result in blocks very far apart
  - where Java/C++ ***new*** command allocates memory
- In C, specify number of <u>bytes</u> of memory explicitly to allocate item

```
int *ptr;
ptr = (int *) malloc(sizeof(int));
/* malloc returns type (void *),
so need to cast to right type */
```

  - `malloc()`: Allocates raw, uninitialized memory from heap

Memory Management

- How do we manage memory?
  - Code, Static
    - Simple
    - They never grow or shrink
  - Stack
    - Simple
    - Stack frames are created and destroyed in last-in, first-out (LIFO) order
  - Heap
    - Tricky
    - Memory can be allocated / deallocated at any time

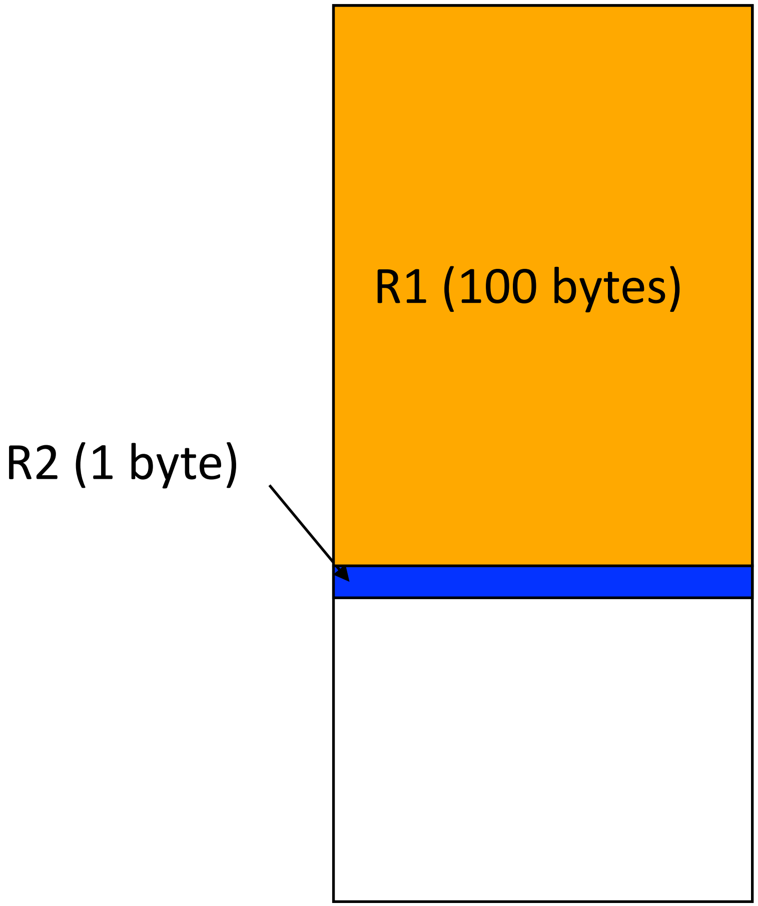
Heap Management Requirements

- Want `malloc()` and `free()` to run quickly.
- Want minimal memory overhead
- Want to avoid fragmentation*
  - When most of our free memory is in many small chunks
  - In this case, we might have many free bytes but not be able to satisfy a large request since the free bytes are not contiguous in memory.

* This is technically called *external fragmentation*

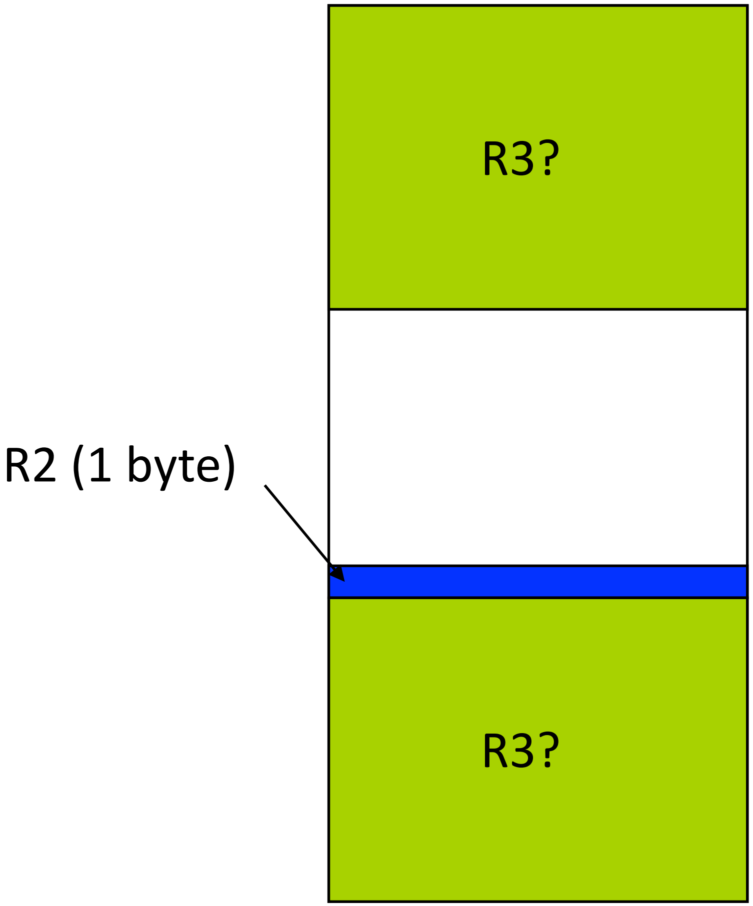
Heap Management

- An example
  - Request R1 for 100 bytes
  - Request R2 for 1 byte
  - Memory from R1 is freed



Heap Management

- An example
  - Request R1 for 100 bytes
  - Request R2 for 1 byte
  - Memory from R1 is freed
  - Request R3 for 50 bytes



K&R Malloc/Free Implementation

- From Section 8.7 of K&R
  - Code in the book uses some C language features we haven't discussed and is written in a very terse style, don't worry if you can't decipher the code
- Each block of memory is preceded by a header that has two fields:
  - size of the block
  - a pointer to the next block
- All free blocks are kept in a circular linked list, the pointer field is unused in an allocated block

K&R Implementation

- `malloc()` searches the free list for a block that is big enough. If none is found, more memory is requested from the operating system. If what it gets can't satisfy the request, it fails.
- `free()` checks if the blocks adjacent to the freed block are also free
  - If so, adjacent free blocks are merged (coalesced) into a single, larger free block
  - Otherwise, the freed block is just added to the free list