CSE 31
Computer Organization

Lecture 4 – C Programming (4):
Structures & Memory

Announcement

- Lab #2 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
  - Chapter 6, 8.7 of K&R (C book) to review on C/C++ programming

C structures : Overview

- A **`struct`** is a data structure composed from simpler data types.
  - Like a class in Java/C++ but without methods or inheritance.

```
struct point {   /* type definition */
     int x;
     int y;
};

void PrintPoint(struct point p){

     printf("(%d,%d)", p.x, p.y);
}

struct point p1 = {0,10}; /* x=0, y=10 */

PrintPoint(p1);
```

As always in C, the argument is passed by "value" – a copy is made.

C structures: Pointers to them

- Usually, more efficient to pass a pointer to the struct.
- The C arrow operator (->) dereferences and extracts a structure field (member) with a single operator.
- The following are equivalent:

```
struct point *p;
/* code to assign to pointer */
printf("x is %d\n", (*p).x);
printf("x is %d\n", p->x);
```

How big are structs?

- Recall C operator `sizeof()` which gives size in bytes (of type or variable)
- How big is `sizeof(p)`?

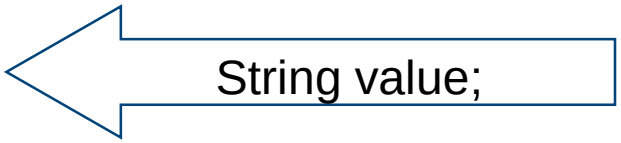
```
struct p {
    char x;
    int y;
};
```

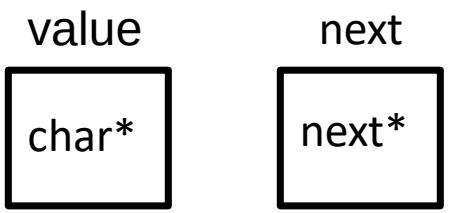
  - 5 bytes? 8 bytes?
  - Compiler may word align integer `y`
  - More on this later lectures

Linked List Example

- Let's look at an example of using structures, pointers, `malloc()`, and `free()` to implement a linked list of strings.

```
/* node structure for linked list */
struct Node {
    char *value;
    struct Node *next;
};
```





typedef simplifies the code

```
struct Node {
    char *value;
    struct Node *next;
  };

/* "typedef" means define a new type */
typedef struct Node NodeStruct;
```

... OR ...

```
typedef struct Node {
    char *value;
    struct Node *next;
  } NodeStruct;
```

... THEN

```
 typedef NodeStruct *List;
 typedef char *String;
```

```
/* Note similarity! */
/* C++ */
/* To define 2 nodes */

struct Node {
    char *value;
    struct Node *next;
} node1, node2;
```

Linked List Example

```
/* Add a string to an existing list */
List cons(String s, List list)
{
  List node = (List) malloc(sizeof(NodeStruct));

  node->value = (String) malloc (strlen(s) + 1);
  strcpy(node->value, s);
  node->next = list;
  return node;
}
```

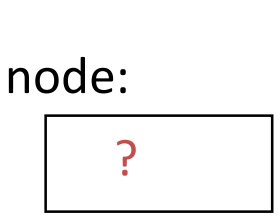
List is a NodeStruct pointer type

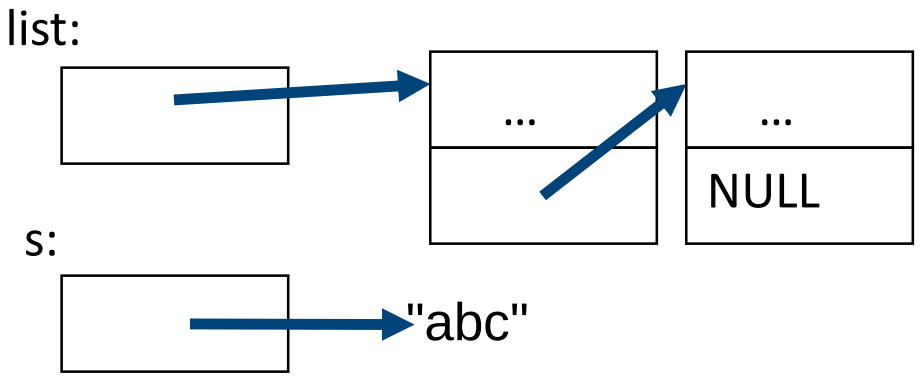
String is a char pointer type

```
String s1 = "abc", s2 = "cde";
List theList = NULL;
theList = cons(s2, theList);
theList = cons(s1, theList);
    /* or embedded */
theList = cons(s1, cons(s2, NULL));
```

Linked List Example

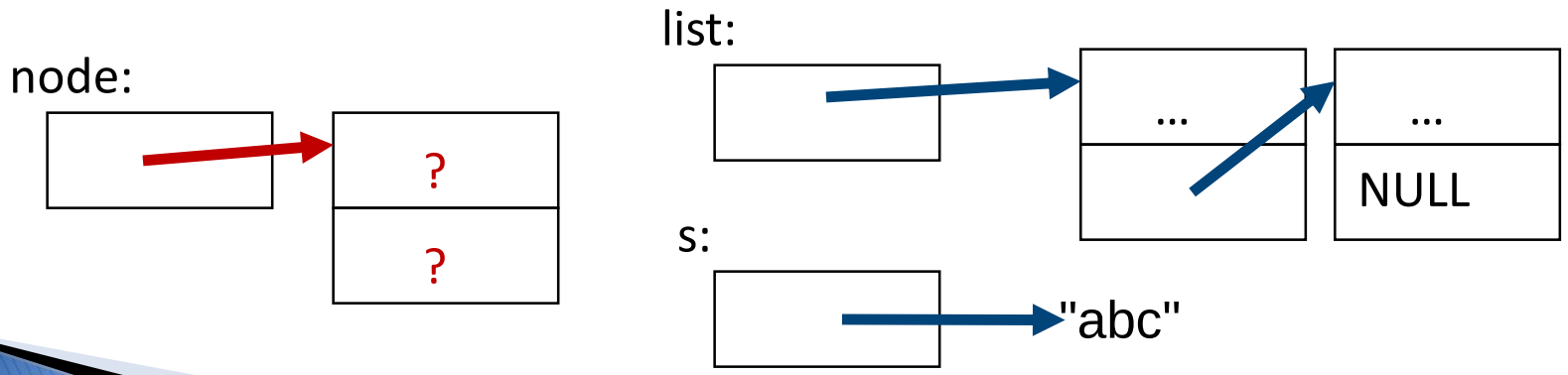
```
/* Add a string to an existing list */
List cons(String s, List list)
{
  List node = (List) malloc(sizeof(NodeStruct));

  node->value = (String) malloc (strlen(s) + 1);
  strcpy(node->value, s);
  node->next = list;
  return node;
}
```





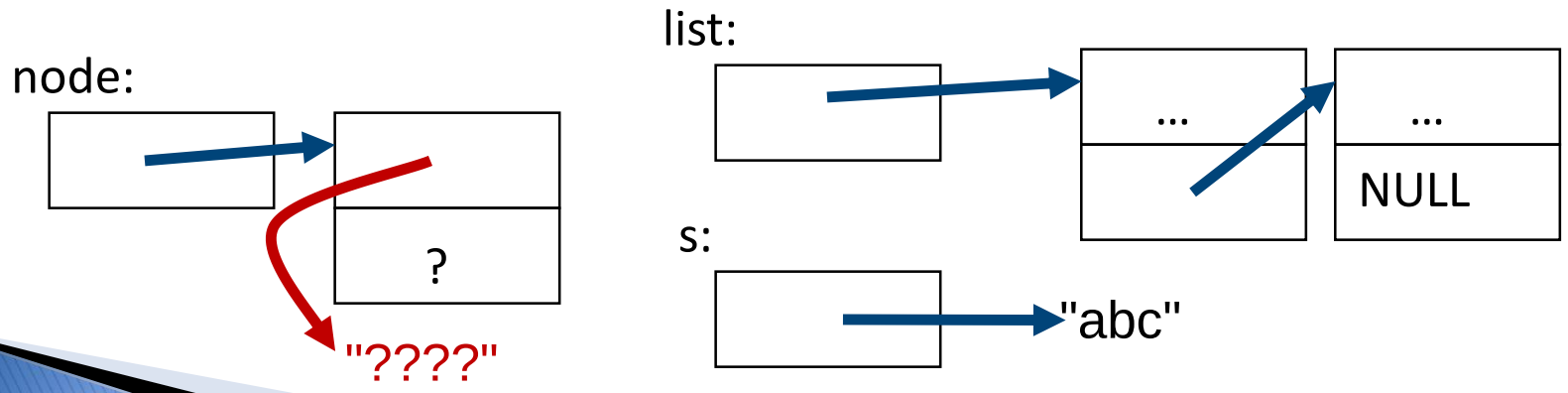
Linked List Example

```
/* Add a string to an existing list */
List cons(String s, List list)
{
  List node = (List) malloc(sizeof(NodeStruct));

  node->value = (String) malloc (strlen(s) + 1);
  strcpy(node->value, s);
  node->next = list;
  return node;
}
```



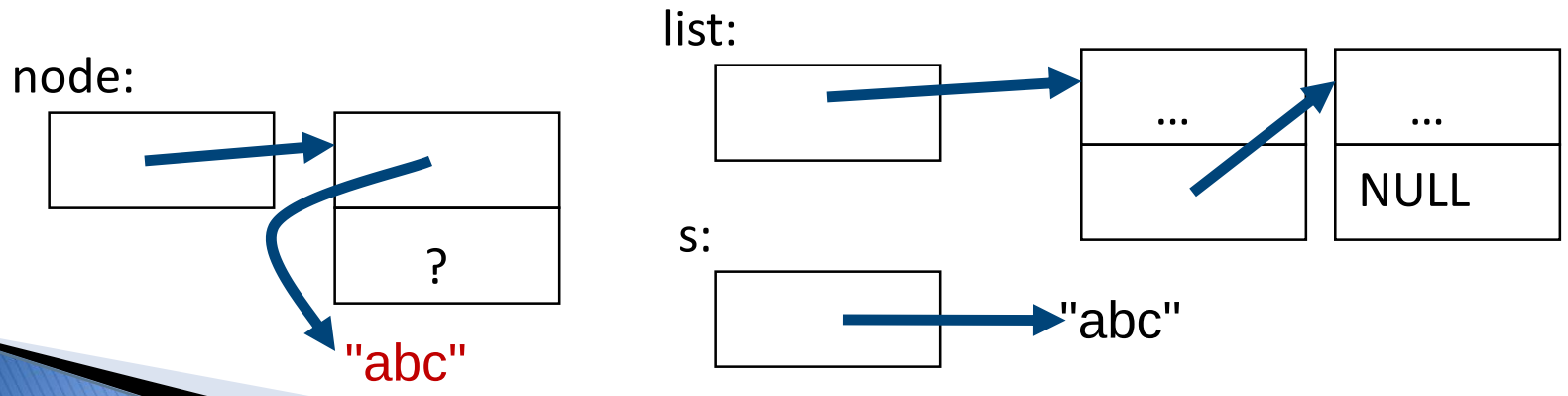
Linked List Example

```
/* Add a string to an existing list */
List cons(String s, List list)
{
  List node = (List) malloc(sizeof(NodeStruct));

  node->value = (String) malloc (strlen(s) + 1);
  strcpy(node->value, s);
  node->next = list;
  return node;
}
```



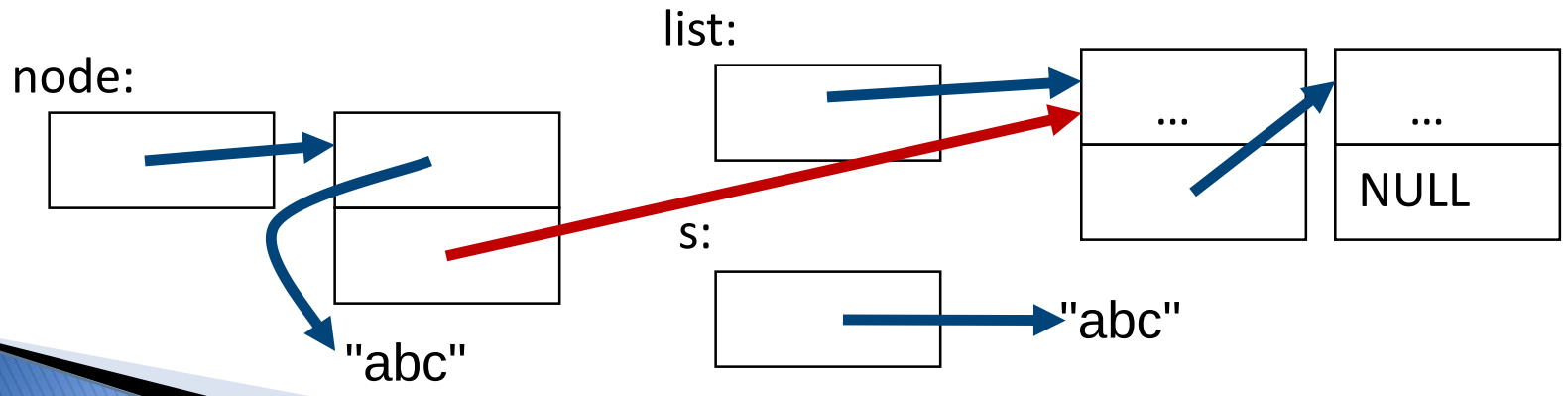
Linked List Example

```
/* Add a string to an existing list */
List cons(String s, List list)
{
  List node = (List) malloc(sizeof(NodeStruct));

  node->value = (String) malloc (strlen(s) + 1);
  strcpy(node->value, s);
  node->next = list;
  return node;
}
```



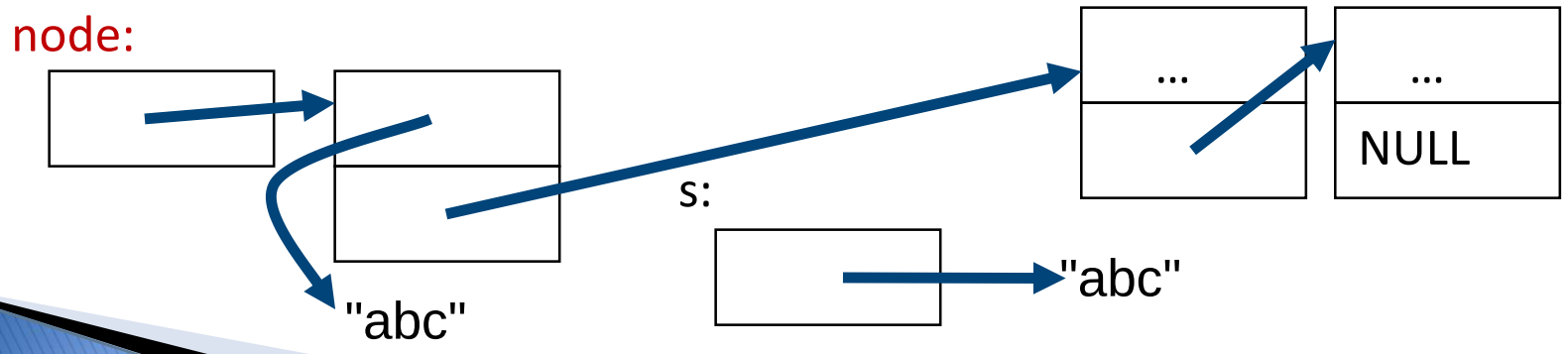
Linked List Example

```
/* Add a string to an existing list */
List cons(String s, List list)
{
  List node = (List) malloc(sizeof(NodeStruct));

  node->value = (String) malloc (strlen(s) + 1);
  strcpy(node->value, s);
  node->next = list;
  return node;
}
```



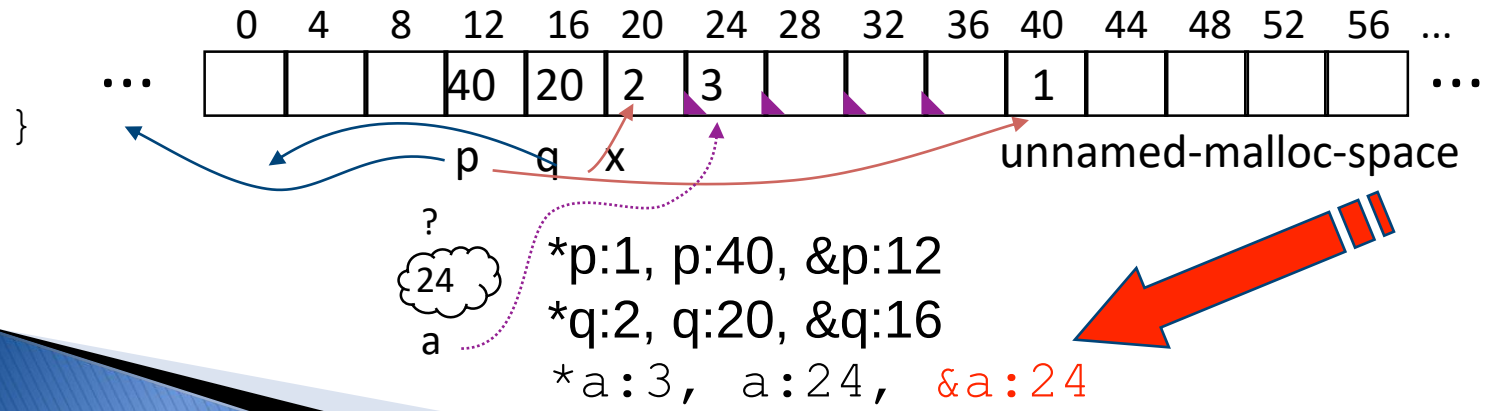
Linked List Example

```
/* Add a string to an existing list */
List cons(String s, List list)
{
  List node = (List) malloc(sizeof(NodeStruct));

  node->value = (String) malloc (strlen(s) + 1);
  strcpy(node->value, s);
  node->next = list;
  return node;
}
```



Arrays not implemented as you'd think

```
void foo() {
  int *p, *q, x;
  int a[4];
  p = (int *) malloc (sizeof(int));
  q = &x;

  *p = 1; // p[0] would also work here
  printf("*p:%u, p:%u, &p:%u\n", *p, p, &p);
  *q = 2; // q[0] would also work here
  printf("*q:%u, q:%u, &q:%u\n", *q, q, &q);
  *a = 3; // a[0] would also work here
  printf("*a:%u, a:%u, &a:%u\n", *a, a, &a);
}
```



K&R: "An array name is not a variable"

Don't forget the globals!

- Remember:
  - Structure declaration does not allocate memory
    - Only when you instantiate it.
  - Variable declaration does allocate memory
- So far we have talked about several different ways to allocate memory for data:
  1. Declaration of a local variable in a function

```
int i; struct Node list; char *string;
int ar[n];
```

  2. "Dynamic" allocation at runtime by calling allocation function (malloc).

```
ptr = (struct Node *) malloc(sizeof(struct Node)*n);
```

- One more possibility exists…
  3. Data declared outside of any procedure/function (i.e., before `main`).
  - Similar to #1 above, but has "global" scope.

Useful in C, but not in Java/C++

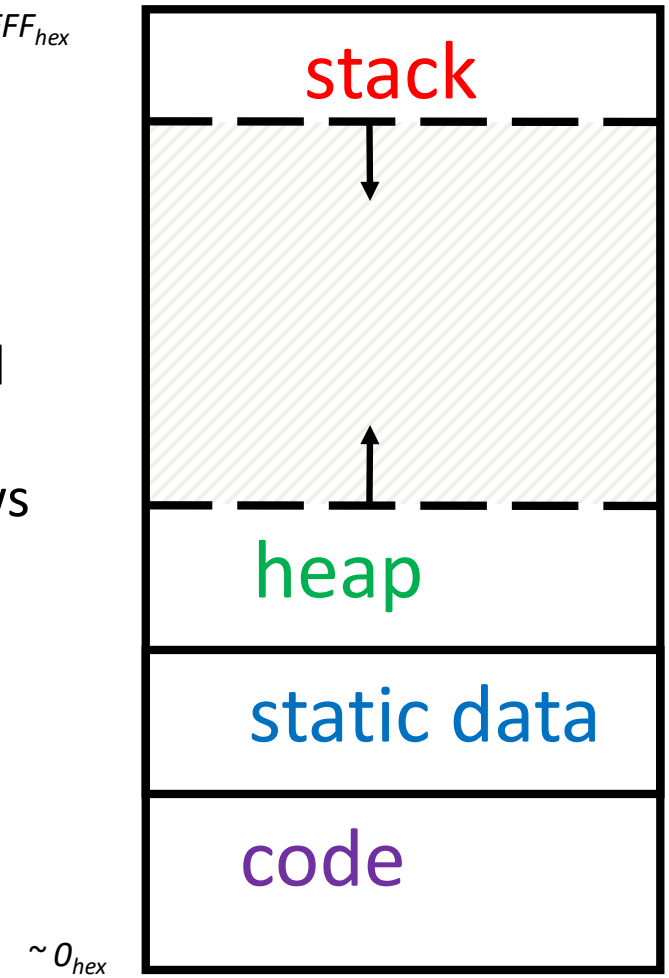
```
int myGlobal;
main() {
        …
}
```

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

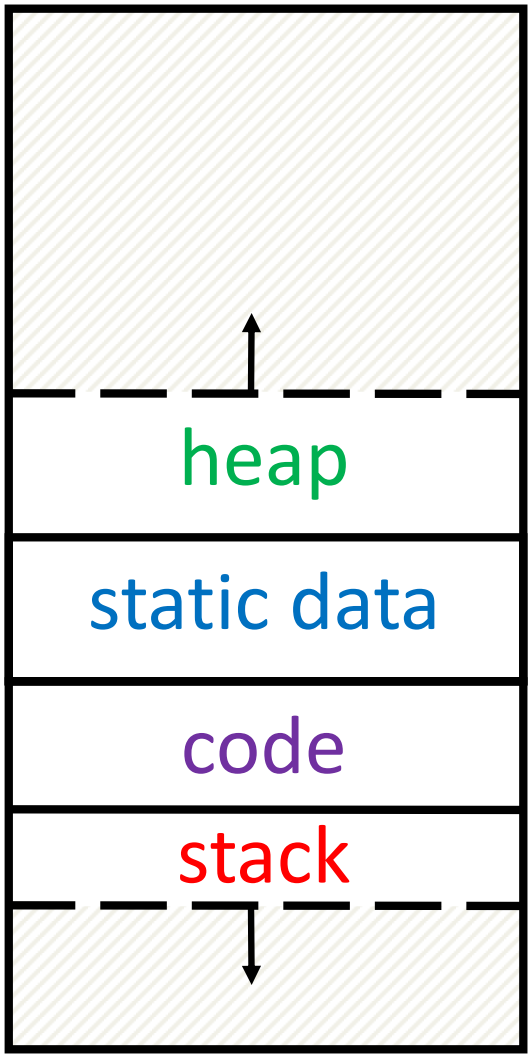


For now, OS somehow prevents accesses between stack and heap (gray hash lines). Wait for virtual memory

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

$\sim 08000000_{hex}$



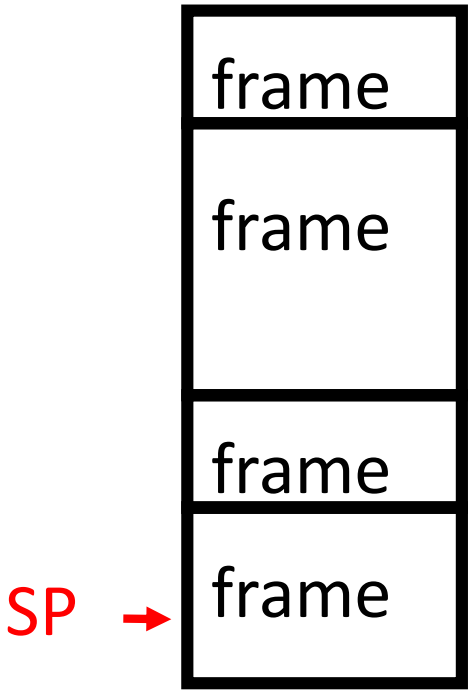
Where are variables allocated?

- If declared outside of a 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

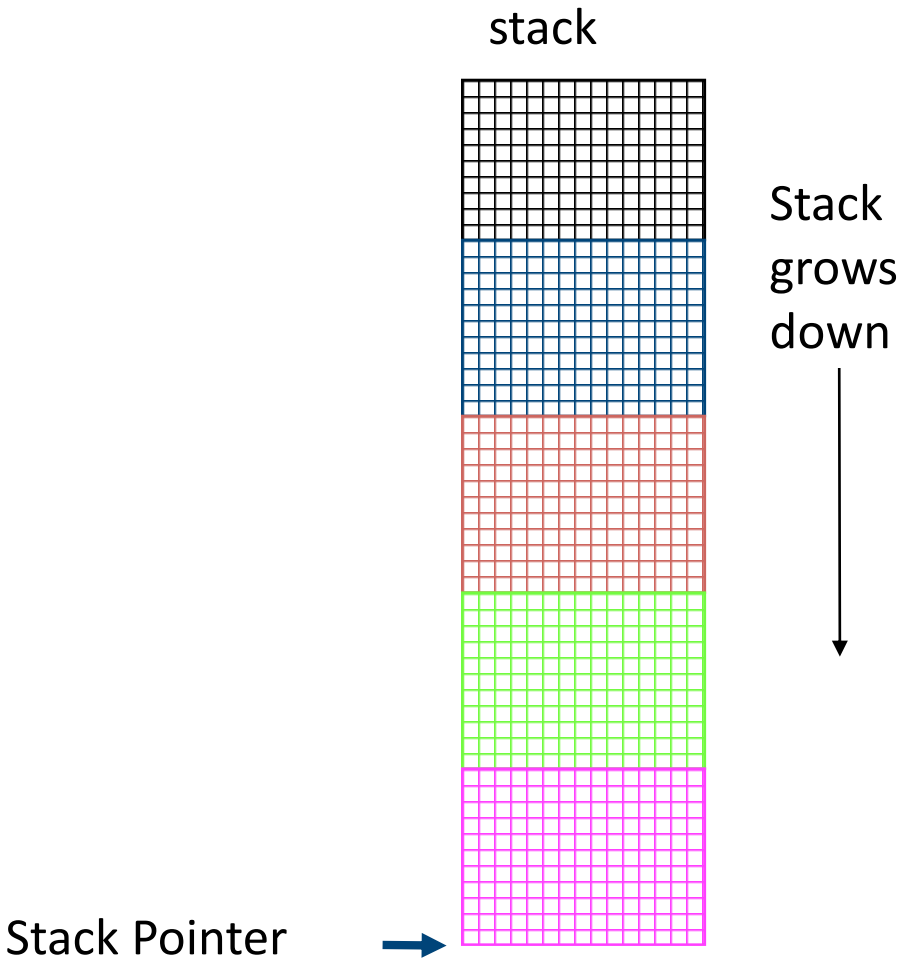
- Stack frame includes storage for:
  - Return "instruction" address
  - Parameters (input arguments)
  - Space for other local variables
- Stack frames:
  - contiguous blocks of memory for a function
  - stack pointer tells where top stack frame is
- When a function ends, stack frame is "popped off"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



Stack

- Last In, First Out (LIFO) data structure

```
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) {
}
```