CSE 31
Computer Organization

Lecture 5 – C Memory Management

Announcement

- Lab #2 this week
  - Due in one week
- HW #1
  - From zyBooks
  - Due today
- Reading assignment
  - Chapter 7 and 8.7 of K&R (C book) to review on C/C++ programming
- Tutoring from PALS

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;
};
```

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

```
void PrintPoint(struct point p){

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

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

PrintPoint(p1);
```

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`

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;
};
```

Recursive definition!

typedef simplifies the code

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

String value;

```
/* "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!  */
/* To define 2 nodes */

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

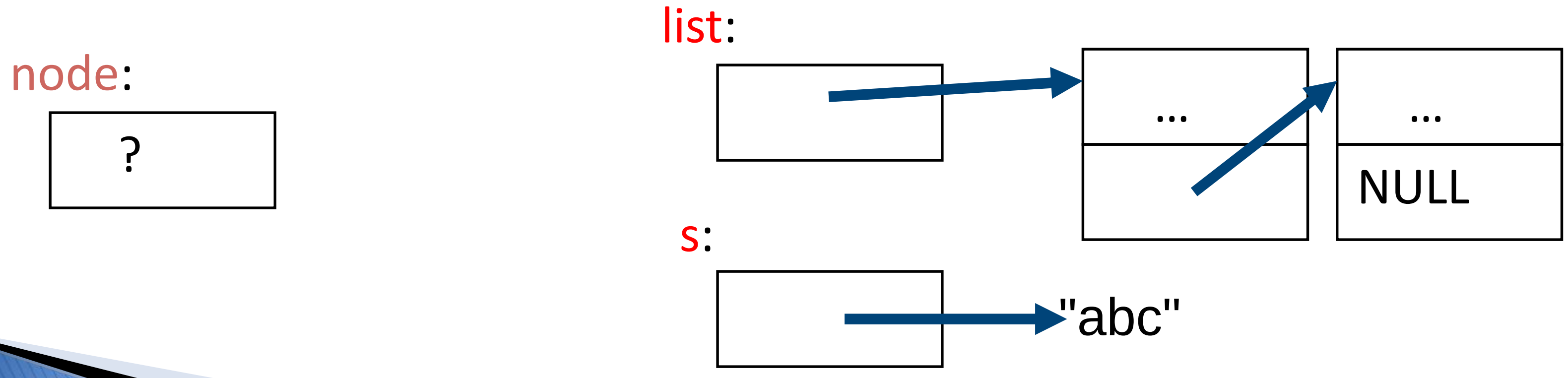
Linked List Example

```
/* 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;
}
```

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

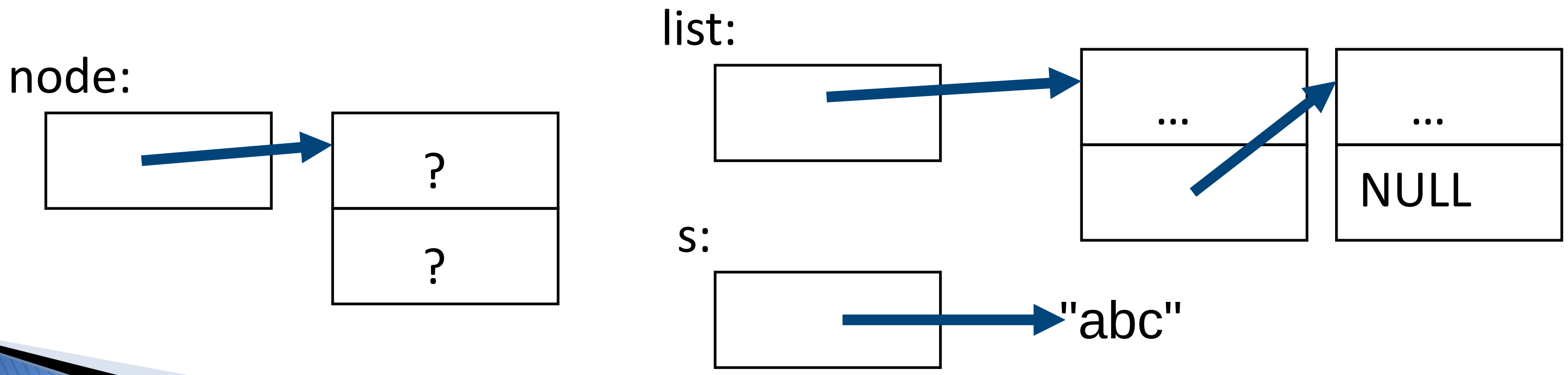
Linked List Example

```
/* Add a string to an existing list, 2nd call */
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;
}
```



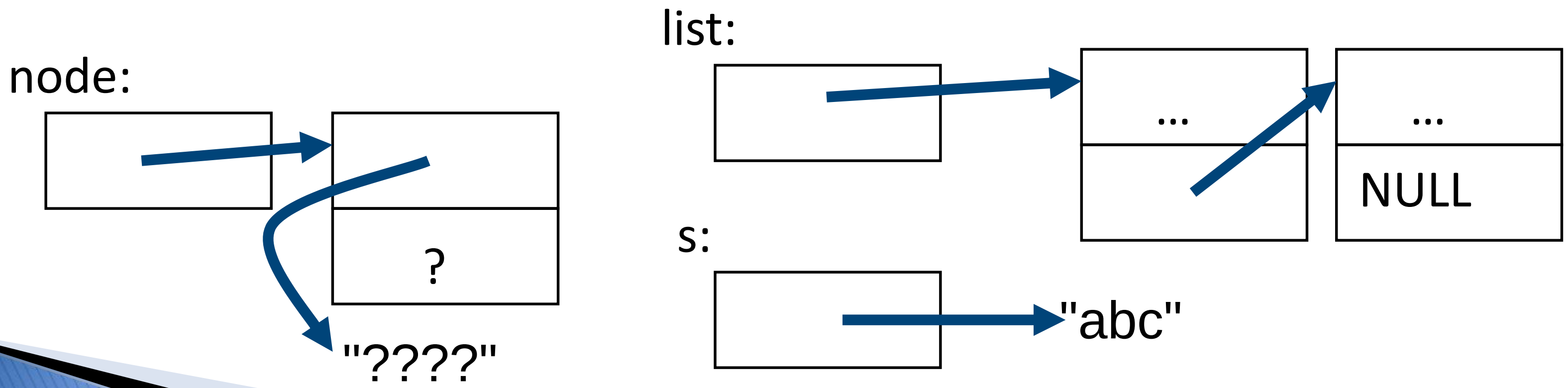
Linked List Example

```
/* Add a string to an existing list, 2nd call */
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;
}
```



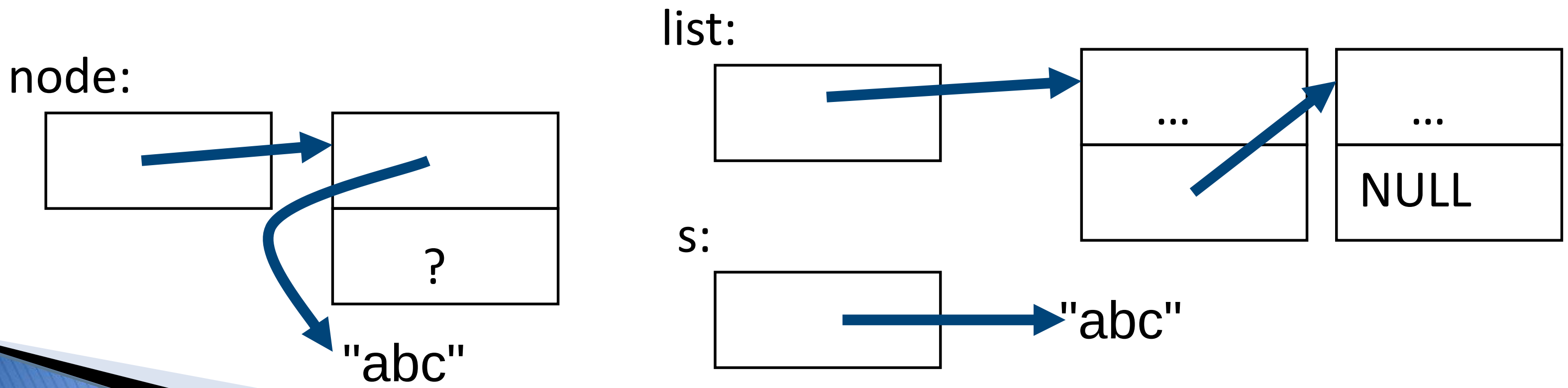
Linked List Example

```c
/* Add a string to an existing list, 2nd call */
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;
}
```



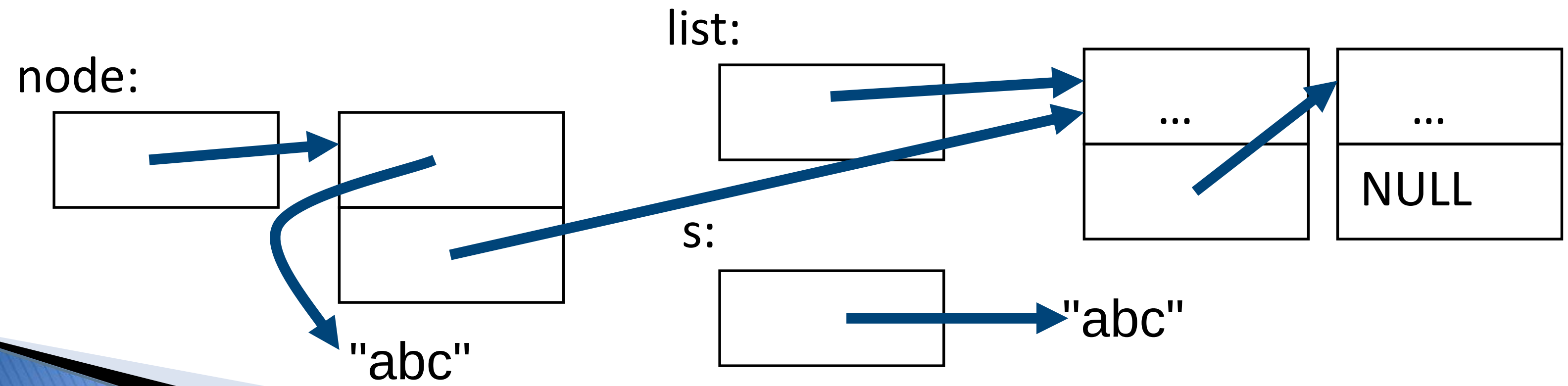
Linked List Example

```
/* Add a string to an existing list, 2nd call */
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;
}
```



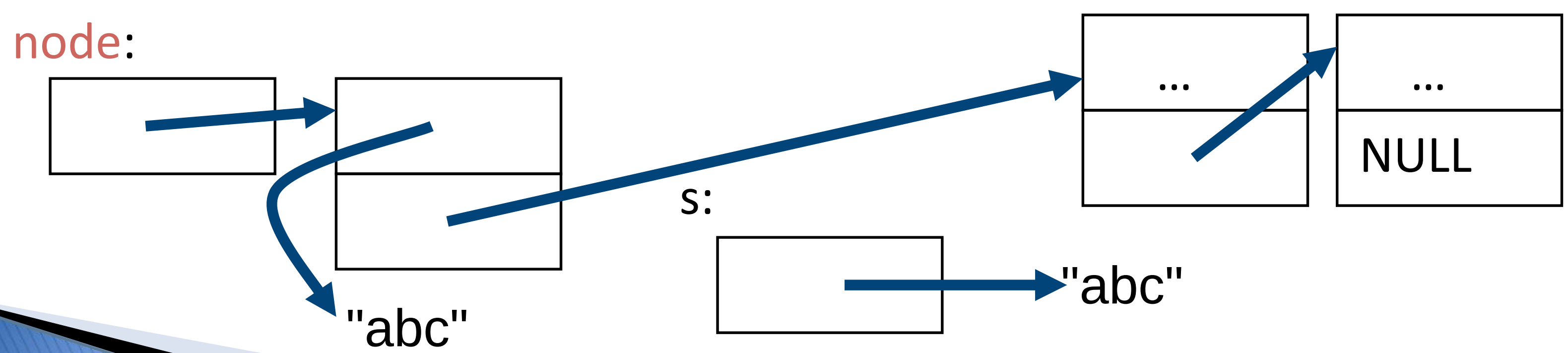
Linked List Example

```
/* Add a string to an existing list, 2nd call */
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;
}
```



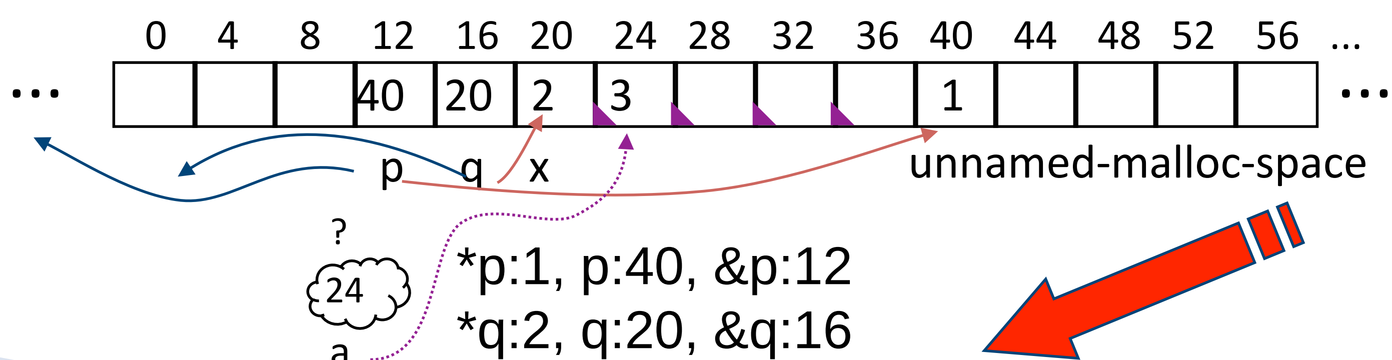
Linked List Example

```
/* Add a string to an existing list, 2nd call */
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);
}
```



*p:1, p:40, &p:12
*q:2, q:20, &q:16
`*a:3, a:24, &a:24`

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
     ```
     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.

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

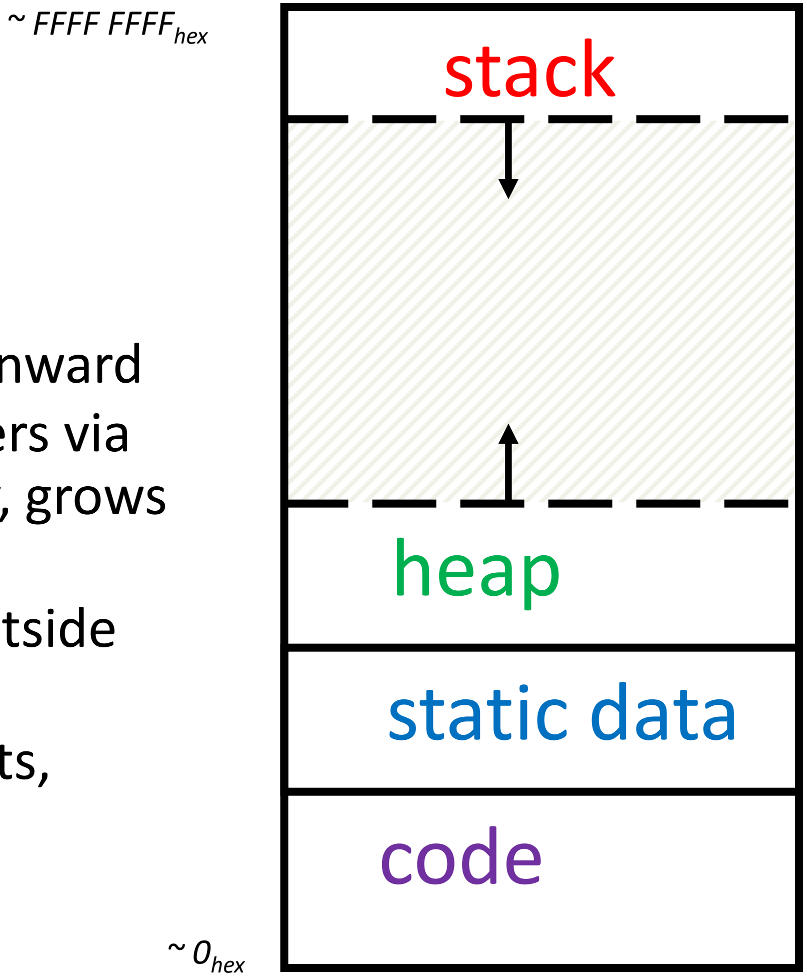
Useful in C, but not in Java/C++

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 objects 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

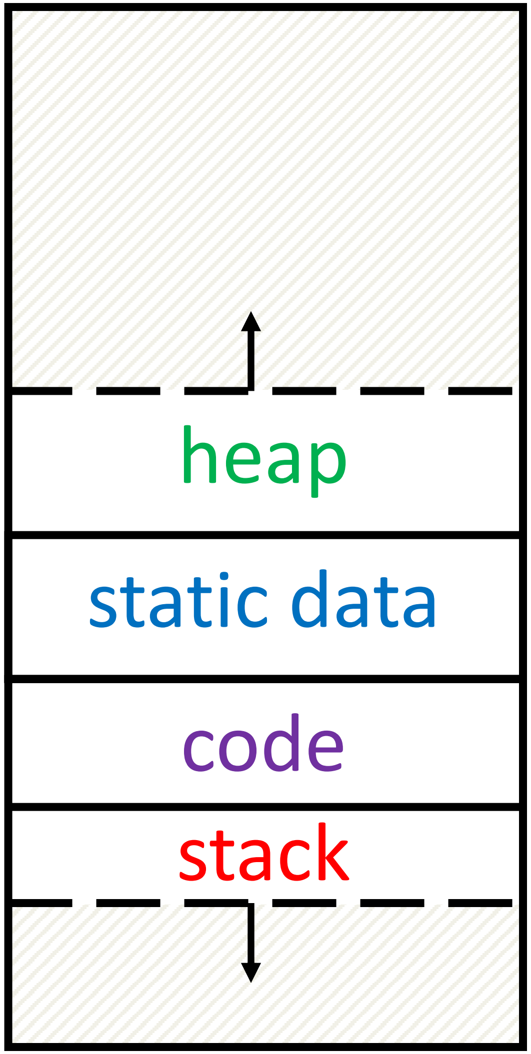


For now, OS somehow prevents accesses between stack and heap (gray hash lines). Wait for virtual memory

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

~ 08000000_{hex}



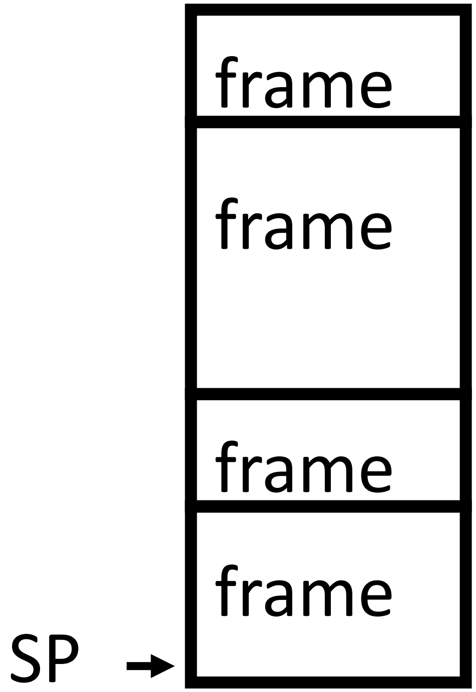
Where are variables allocated?

- 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

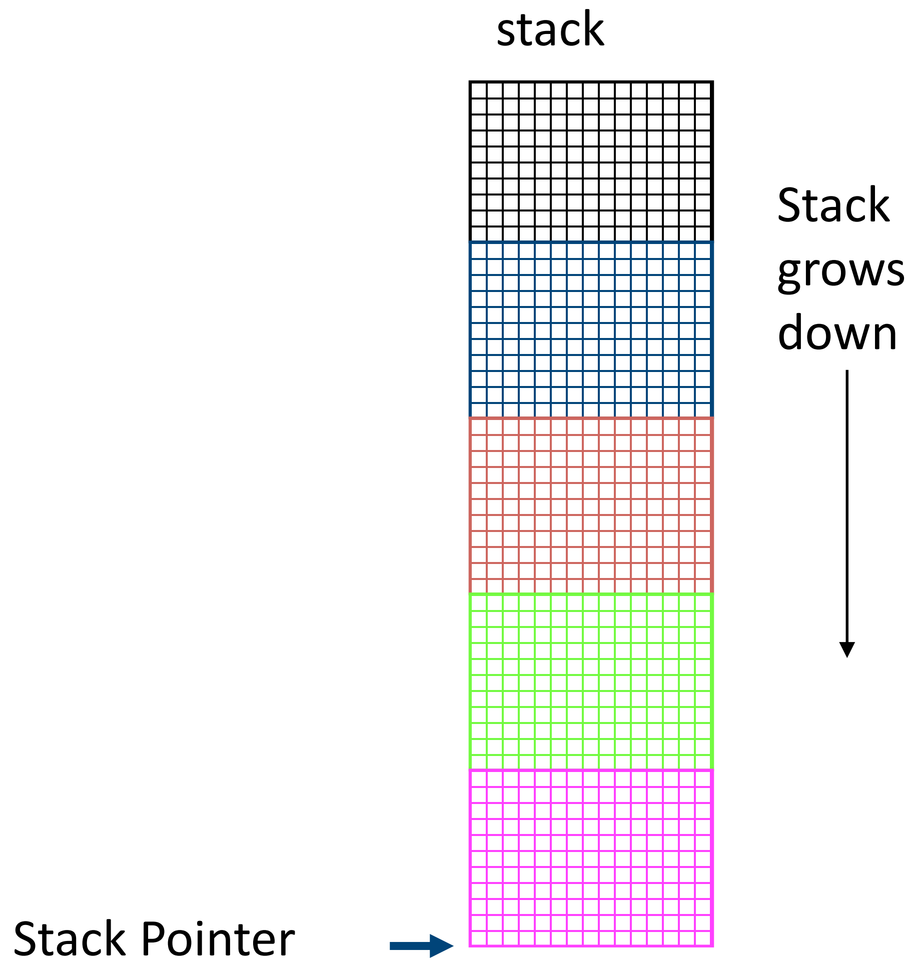
- Stack frame includes storage for:
  - Return "instruction" address
  - Parameters (input arguments)
  - Space for other local variables
- Stack frames:
  - contiguous blocks of memory
  - stack pointer tells where top stack frame is
- When a function ends, stack frame is "popped off" the stack; frees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



Stack

- Last In, First Out (LIFO) data structure

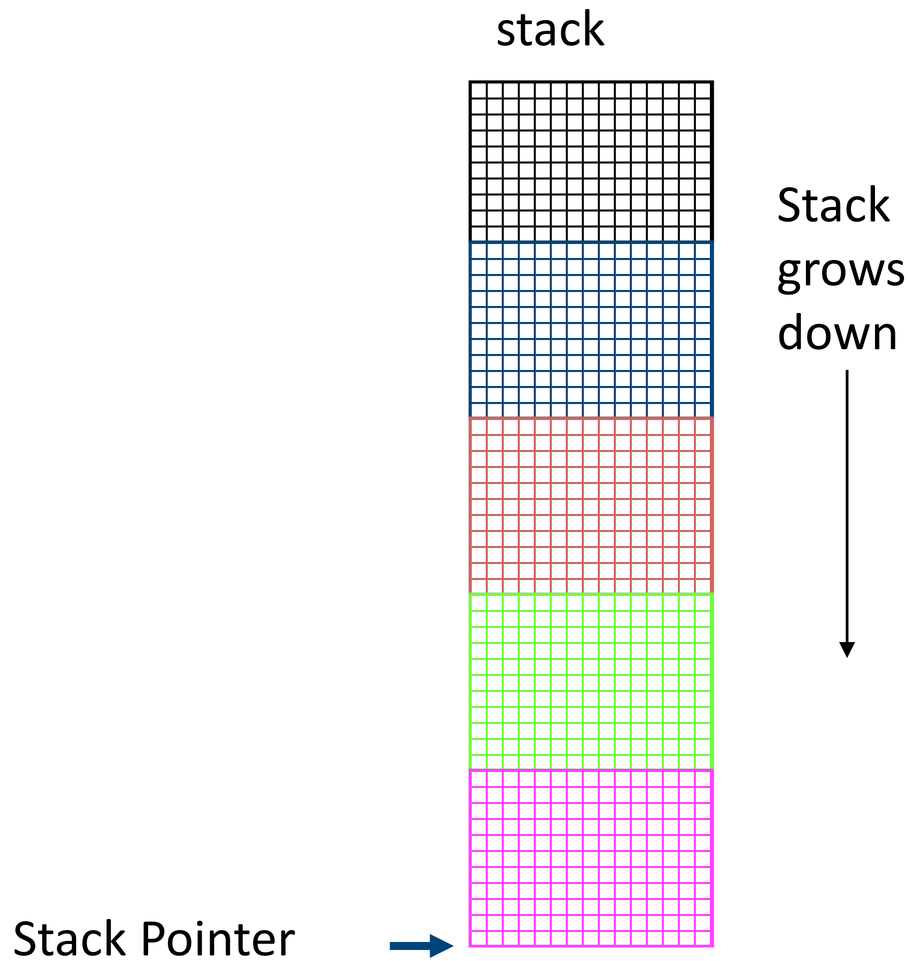
```
main (){
   a(0);
}
void a (int m) {
   b(1);
}
void b (int n){
   c(2);
}
void c (int o){
   d(3);
}
void d (int p) {
}
```



Stack

- Last In, First Out (LIFO) data structure

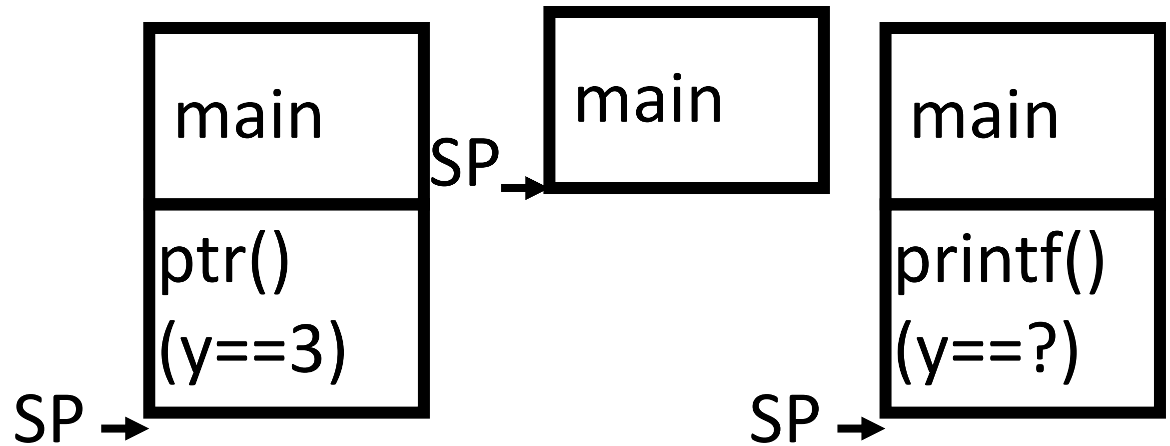
```
main (){
   a(0);
}
void a (int m) {
   b(1);
}
void b (int n){
   c(2);
}
void c (int o){
   d(3);
}
void d (int p) {
}
```



Who cares about stack management?

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

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




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

The Heap (Dynamic memory)

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

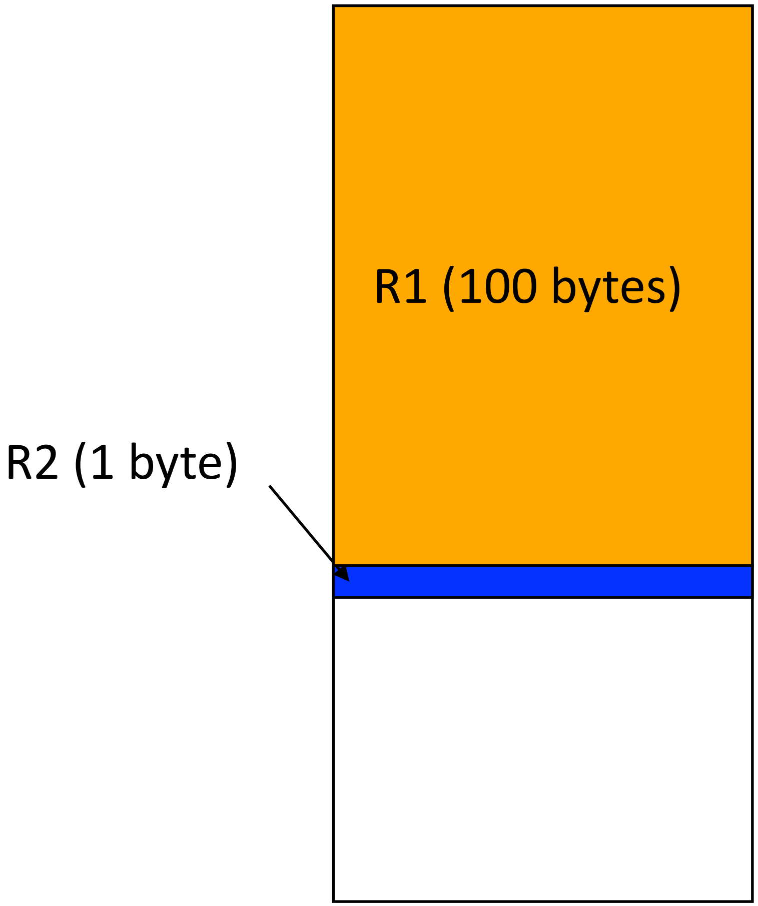
Heap Management Requirements

- 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 fragmention*

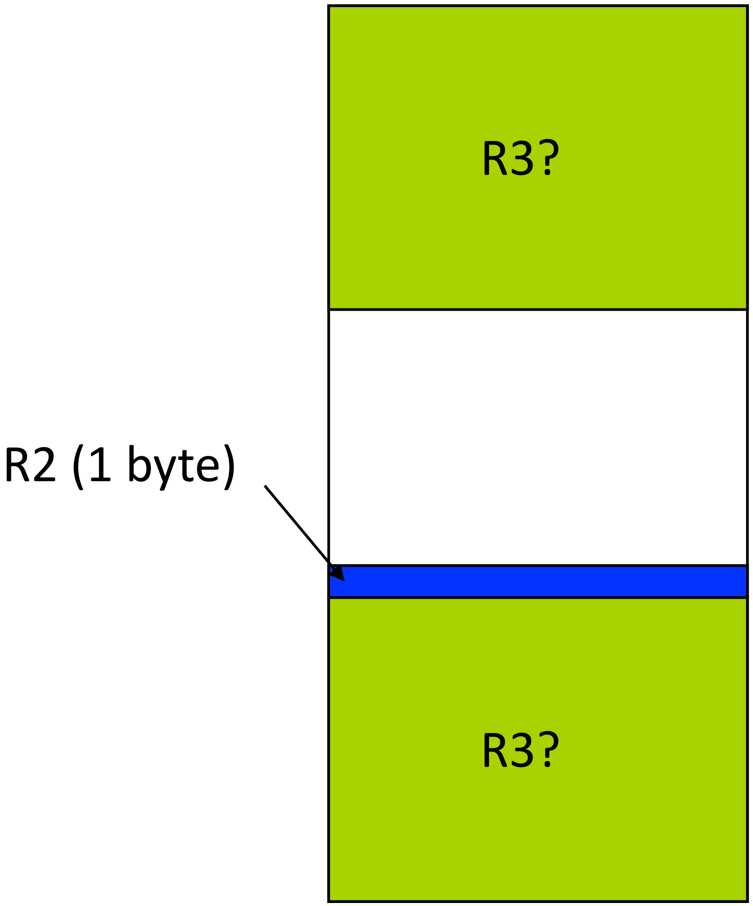
Heap Management

- 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

Choosing a block in `malloc()`

- If there are multiple free blocks of memory that are big enough for some request, how do we choose which one to use?
  - best-fit: choose the smallest block that is big enough for the request
  - first-fit: choose the first block we see that is big enough
  - next-fit: like first-fit but remember where we finished searching and resume searching from there

Tradeoffs of allocation policies

- Best-fit: Tries to limit fragmentation but at the cost of time (must examine all free blocks for each malloc).
  - Leaves lots of small blocks (why?)
- First-fit: Quicker than best-fit (why?) but potentially more fragmentation.
  - Tends to concentrate small blocks at the beginning of the free list (why?)
- Next-fit: Does not concentrate small blocks at front like first-fit, should be faster as a result.

Quiz – Pros and Cons of fits

1) first-fit results in many small blocks at the beginning of the free list
2) next-fit is slower than first-fit, since it takes longer in steady state to find a match
3) best-fit leaves lots of tiny blocks

	123
a)	FFT
b)	FTT
c)	TFF
d)	TFT
e)	TTT

Quiz – Pros and Cons of fits

1) first-fit results in many small blocks at the beginning of the free list
2) next-fit is slower than first-fit, since it takes longer in steady state to find a match
3) best-fit leaves lots of tiny blocks

	123
a)	FFT
b)	FTT
c)	TFF
d)	**TFT**
e)	TTT

Summary

- C has 3 pools of memory
  - Static storage: global variable storage, basically permanent, entire program run
  - The Stack: local variable storage, parameters, return address
  - The Heap (dynamic storage): `malloc()` grabs space from here, `free()` returns it.

- `malloc()` handles free space with freelist. Three different ways to find free space when given a request:
  - First fit (find first one that's free)
  - Next fit (same as first, but remembers where left off)
  - Best fit (finds most "snug" free space)