Review

- Pointers and arrays are very similar
- Strings are just char pointers/arrays with a null terminator at the end
- Pointer arithmetic moves the pointer by the size of the thing it's pointing to
- Pointers are the source of many C bugs!

Multiple Ways to Store Program Data

- Static global data
  - *Fixed size* at compile-time
  - Entire *lifetime of the program* (loaded from executable)
  - Portion is read-only (e.g. string literals)
- Stack-allocated data
  - Local/temporary variables
    - *Can* be dynamically sized (in some versions of C)
  - *Known lifetime* (deallocated on `return`)
- **Dynamic (heap) data**
  - Size known only at runtime (i.e. based on user-input)
  - Lifetime known only at runtime (long-lived data structures)

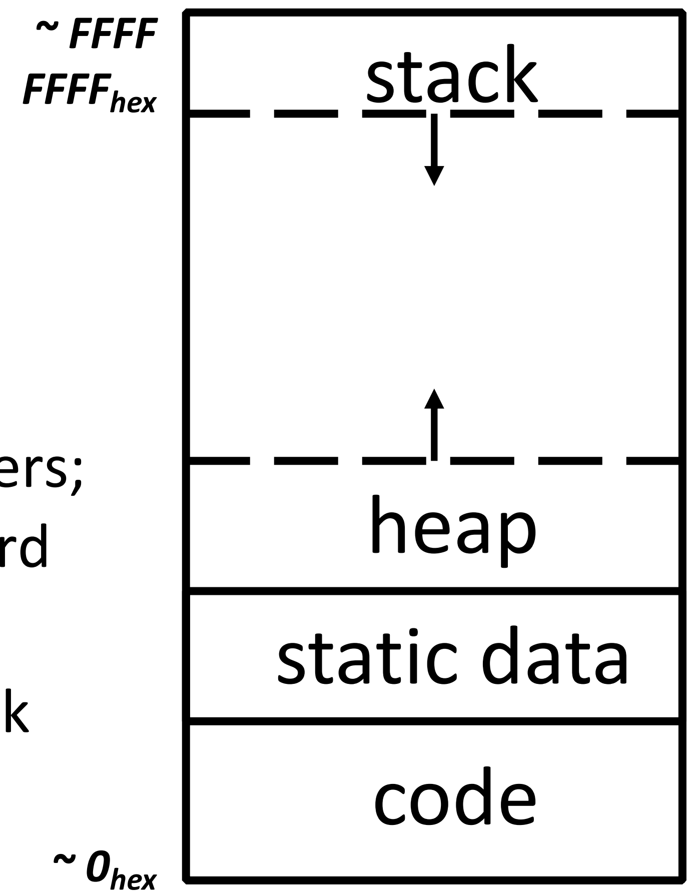
```
int array[1024];

void foo(int n) {
  int tmp;
  int local_array[n];

  int* dyn =
     (int*)malloc(n*sizeof(int));
}
```

Agenda

- C Memory Layout
  - Stack, Static Data, and Code
- Dynamic Memory Allocation
  - Heap
- Common Memory Problems
- C Wrap-up: Linked List Example

C Memory Layout

- Program's *address space* contains 4 regions:
  - Stack: local variables, grows downward
  - Heap: space requested via `malloc()` and used with pointers; resizes dynamically, grows upward
  - Static Data: global and static variables, does not grow or shrink
  - Code: loaded when program starts, does not change



OS prevents accesses between stack and heap (via virtual memory)

Where Do the Variables Go?

- Declared outside a function:

  Static Data

- Declared inside a function:

  Stack
  - `main()` is a function
  - Freed when function returns

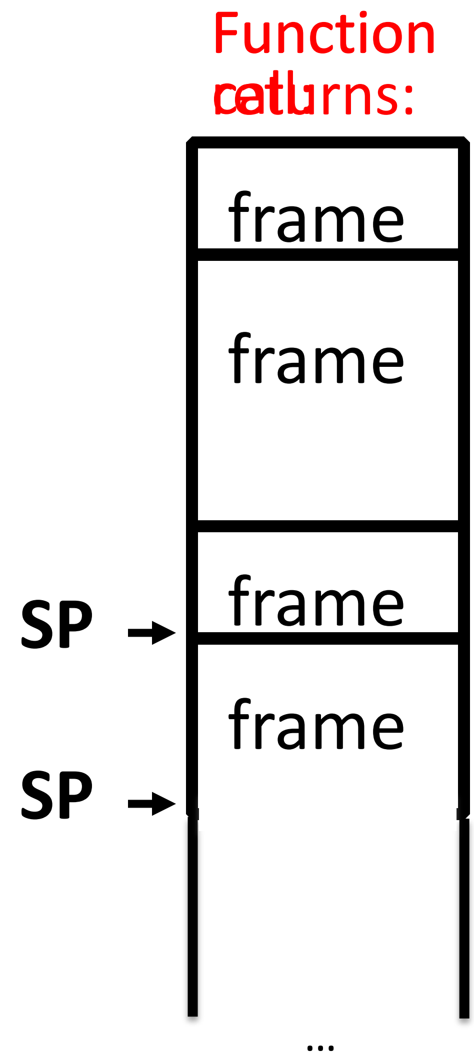
- Dynamically allocated:

  Heap
  - i.e. malloc (we will cover this shortly)

```
#include <stdio.h>

int varGlobal;

int main() {
  int varLocal;
  int *varDyn =
    malloc(sizeof(int));
}
```

The Stack

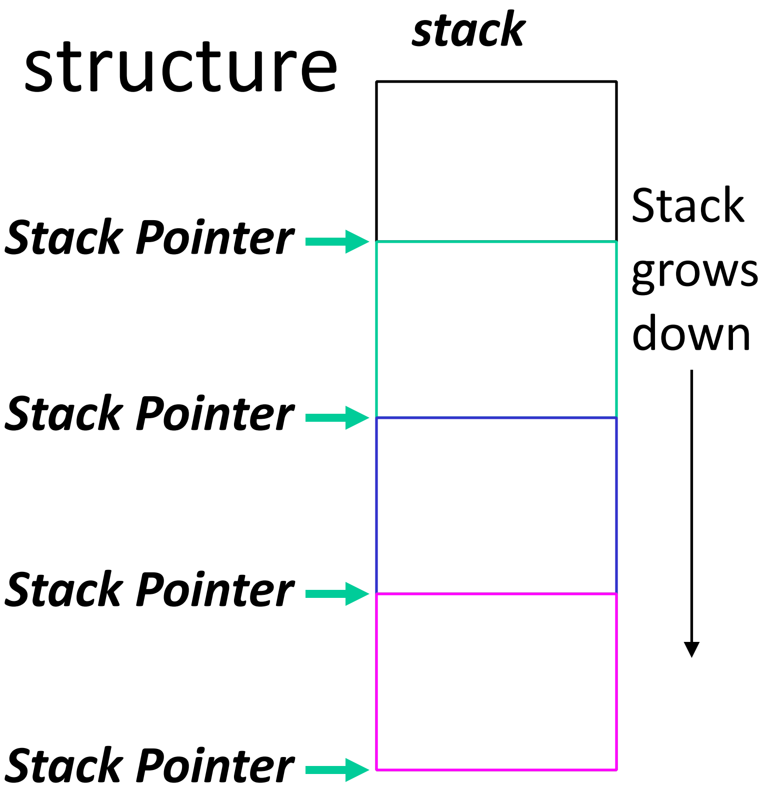
- Each stack frame is a contiguous block of memory holding the local variables of a single procedure
- A stack frame includes:
  - Location of caller function
  - Function arguments
  - Space for local variables
- Stack pointer (SP) tells where lowest (current) stack frame is
- When procedure ends, stack pointer is moved back (but data remains (garbage!)); frees memory for future stack frames;



The Stack

- Last In, First Out (LIFO) data structure

```
int main() {
   a(0);
   return 1; }
void a(int m) {
   b(1); }
void b(int n) {
   c(2);
   d(4); }
void c(int o) {
   printf("c"); }
void d(int p) {
   printf("d"); }
```



Stack Misuse Example

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

int main () {
   int *stackAddr,content;
   stackAddr = getPtr();
   content = *stackAddr;
   printf("%d", content); /* 3 */
   content = *stackAddr;
   printf("%d", content); /* ? */
};
```

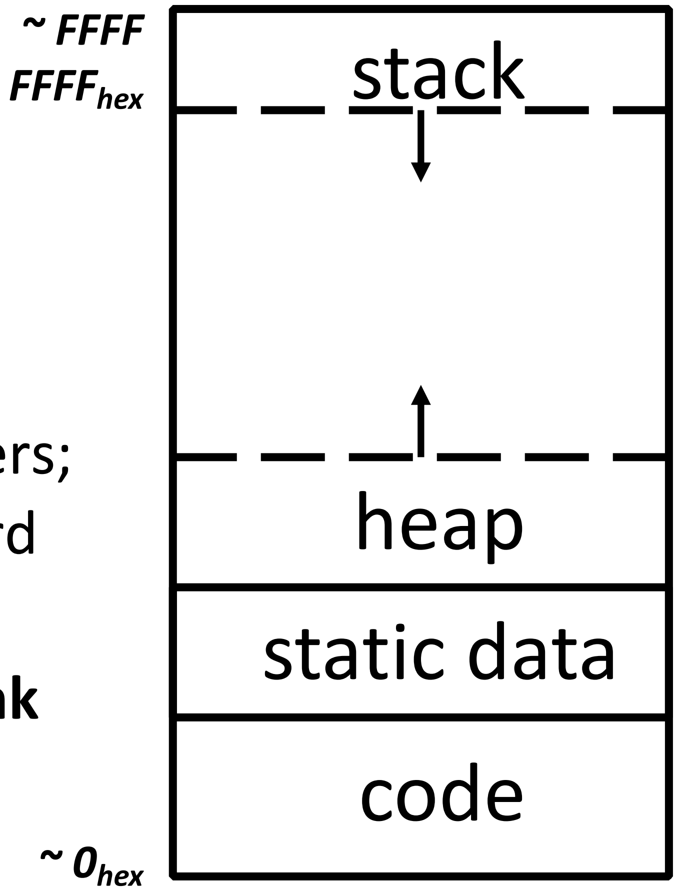
Never return pointers to local variable from functions

Your compiler will warn you about this
– don't ignore such warnings!

printf overwrites stack frame

C Memory Layout

- Program's *address space* contains 4 regions:
  - Stack: local variables, grows downward
  - Heap: space requested via `malloc()` and used with pointers; resizes dynamically, grows upward
  - **Static Data: global and static variables, does not grow or shrink**
  - **Code: loaded when program starts, does not change**



OS prevents accesses between stack and heap (via virtual memory)

Static Data

- Place for variables that persist
  - Data not subject to comings and goings like function calls
  - Examples: **String literals, global variables**
  - String literal example: char * str = “hi”;
- Size does not change, but sometimes data can
  - Notably string literals cannot

Code

- Copy of your code goes here
  - C code becomes data too!
- Does not change

Question: Which statement below is FALSE?
All statements assume each variable exists.

```
void funcA() {int x; printf("A");}
void funcB() {
  int y;
  printf("B");
  funcA();
}
void main() {char *s = "s"; funcB();}
```

(A) `&x` < `&y`

(B) `x` and `y` are in adjacent frames

(C) `&x` < `s`

(D) **`y` is in the 2nd frame from the top of the Stack**

Question: Which statement below is FALSE?
All statements assume each variable exists.

```
void funcA() {int x; printf("A");}
void funcB() {
  int y;
  printf("B");
  funcA();
}
void main() {char *s = "s"; funcB();}
```

This is a string literal, and thus stored in STATIC DATA.

(A) &x < &y

(B) x and y are in adjacent frames

(C) &x < s

(D) y is in the 2nd frame from the top of the Stack

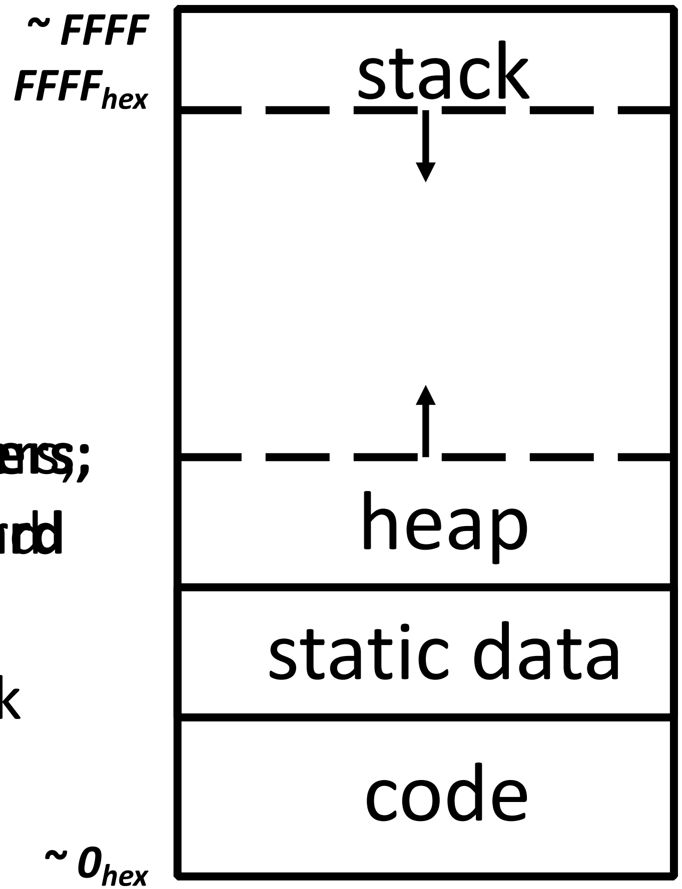
Note: We're talking about *s, not s, i.e. the location where s points!

Agenda

- C Memory Layout
  - Stack, Static Data, and Code
- Administrivia
- Dynamic Memory Allocation
  - Heap
- Common Memory Problems
- C Wrap-up: Linked List Example

C Memory Layout

- Program's *address space* contains 4 regions:
  - Stack: local variables, grows downward
  - **Heap: space requested via `malloc()` and used with pointers; resizes dynamically, grows upward**
  - Static Data: global and static variables, does not grow or shrink
  - Code: loaded when program starts, does not change



OS prevents accesses between stack and heap (via virtual memory)

Dynamic Memory Allocation

- Want persisting memory (like static) even when we don't know size at compile time?
  - e.g. input files, user input
  - Stack won't work because stack frames aren't persistent
- Dynamically allocated memory goes on the Heap – more permanent than Stack
- Need as much space as possible without interfering with Stack
  - Start at opposite end and grow towards Stack

sizeof()

- If integer sizes are machine dependent, how do we tell?
- Use `sizeof()` function
  - Returns size in bytes of variable or data type name

    Examples: `int x; sizeof(x); sizeof(int);`
- Acts differently with arrays and structs, which we will cover later
  - Arrays: returns size of whole array
  - Structs: returns size of one instance of struct (sum of sizes of all struct variables + padding)

Allocating Memory in C

- Need to `#include <stdlib.h>`
- `void* malloc(size_t size)`
  - Allocates a continuous block of `size` bytes of uninitialized memory
  - Returns a pointer to the beginning of the allocated block; NULL indicates failed request
    - Typically aligned to an 8-byte (x86) or 16-byte (x86-64) boundary
    - Returns `NULL` if allocation failed (also sets `errno`) or `size==0`
  - Different blocks not necessarily adjacent
- Related functions:
  - `void* calloc(size_t nitems, size_t size)`
    - "Zeros out" allocated block
  - `void* realloc(void* ptr, size_t size)`
    - Changes the size of a previously allocated block (if possible)
  - `void* sbrk(intptr_t increment)`
    - Used internally by allocators to grow or shrink the heap

Using malloc()

- Almost always used for arrays or structs
- Good practice to use `sizeof()` and typecasting

```
int *p = (int *) malloc(n*sizeof(int));
```

  - `sizeof()` makes code more portable
  - `malloc()` returns `void *`; typecast will help you catch coding errors when pointer types don't match
- Can use array or pointer syntax to access

Releasing Memory

- Release memory on the Heap using `free()`
  - Memory is limited, release when done
- **`free(p)`**
  - Pass it pointer `p` to beginning of allocated block; releases the whole block
  - `p` must be the address *originally* returned by `m/c/realloc()`, otherwise throws system exception
  - Don't call `free()` on a block that has already been released or on NULL
  - Make sure you don't lose the original address
    - `eg: p++` is a **BAD IDEA**; use a separate pointer

End-to-End Example

```
void foo(int n, int m) {
  int i, *p;
  p = (int*) malloc(n*sizeof(int));  /* allocate block of n ints */
  if (p == NULL) {                   /* check for allocation error */
    perror("malloc");
    exit(0);
  }
  for (i=0; i<n; i++)                /* initialize int array */
    p[i] = i;
                                     /* add space for m ints to end of p block */
  p = (int*) realloc(p,(n+m)*sizeof(int));
  if (p == NULL) {                   /* check for allocation error */
    perror("realloc");
    exit(0);
  }
  for (i=n; i < n+m; i++)            /* initialize new spaces */
    p[i] = i;
  for (i=0; i<n+m; i++)              /* print new array */
    printf("%d\n", p[i]);
  free(p);                           /* free p */
}
```

Dynamic Memory Example

- Need `#include <stdlib.h>`

```
typedef struct {
        int x;
        int y;
} point;

point *rect; /* opposite corners = rectangle
*/

...
if( !(rect=(point *) malloc(2*sizeof(point)))
)
   {
        printf("\nOut of memory!\n");
        exit(1);
}
...
free(rect);
```

Check for returned NULL

Do NOT change `rect` during this time!!!

Question: Want output: a[] = {0,1,2} with no errors. Which lines do we need to change?

```
1   #define N 3
2   int *makeArray(int n) {
3     int *ar;
4     ar = (int *) malloc(n * sizeof(int));
5     return ar;
6   }
7   void main() {
8     int i,*a = makeArray(N);
9     for(i=0; i<N; i++)
10      *(a+i) = i;
11    printf("a[] =
        {%i,%i,%i}",a[0],a[1],a[2]);
12    free(a);
13  }
```

(A) 4, 12
(B) 5, 12
(C) 4, 10
(D) 5, 10

Agenda

- C Memory Layout
  - Stack, Static Data, and Code
- Administrivia
- Dynamic Memory Allocation
  - Heap
- Common Memory Problems
- C Wrap-up: Linked List Example

Question: Want output: a[] = {0,1,2} with no errors. Which lines do we need to change?

```
1   #define N 3
2   int *makeArray(int n) {
3     int *ar;
4     ar = (int *) malloc(n);
5     return ar;
6   }
7   void main() {
8     int i,*a = makeArray(N);
9     for(i=0; i<N; i++)
10      *a++ = i;
11    printf("a[] =
        {%i,%i,%i}",a[0],a[1],a[2]);
12    free(a);
13  }
```

(A) 4, 12
(B) 5, 12
(C) 4, 10
(D) 5, 10

Know Your Memory Errors

(Definitions taken from `http://www.hyperdictionary.com`)

- Segmentation Fault ⟵ More common

  "An error in which a running Unix program attempts to access memory not allocated to it and terminates with a segmentation violation error and usually a core dump."

- Bus Error ⟵ Less common in 295

  "A fatal failure in the execution of a machine language instruction resulting from the processor detecting an anomalous condition on its bus. Such conditions include invalid address alignment (accessing a multi-byte number at an odd address), accessing a physical address that does not correspond to any device, or some other device-specific hardware error."

Common Memory Problems

1) Using uninitialized values
2) Using memory that you don’t own
   – Using NULL or garbage data as a pointer
   – De-allocated stack or heap variable
   – Out of bounds reference to stack or heap array
3) Freeing invalid memory
4) Memory leaks

Using Uninitialized Values

- What is wrong with this code?

```
void foo(int *p) {
   int j;
   *p = j;
}

void bar() {
   int i=10;
   foo(&i);
   printf("i = %d\n", i);
}
```

`j` is uninitialized (garbage), copied into `*p`

Using `i` which now contains garbage

Using Memory You Don't Own (1)

- What is wrong with this code?

```
typedef struct node {
   struct node* next;
   int val;
} Node;

int findLastNodeValue(Node* head) {
   while (head->next != NULL)
      head = head->next;
   return head->val;
}
```

What if `head` is NULL?

No warnings! Just Seg Fault that needs finding!

Using Memory You Don't Own (2)

- What is wrong with this code?

```
char *append(const char* s1, const char *s2) {
   const int MAXSIZE = 128;
   char result[MAXSIZE];
   int i=0, j=0;
   for (; i<MAXSIZE-1 && j<strlen(s1); i++,j++)
      result[i] = s1[j];
   for (j=0; i<MAXSIZE-1 && j<strlen(s2); i++,j++)
      result[i] = s2[j];
   result[++i] = '\0';
   return result;
}
```

Local array appears on Stack (arrow pointing to `char result[MAXSIZE];`)

Pointer to Stack (array) no longer valid once function returns (arrow pointing to closing `}`)

Using Memory You Don't Own (3)

- What is wrong with this code?

```
typedef struct {
   char *name;
   int age;
} Profile;

Profile *person =(Profile *)malloc(sizeof(Profile));
char *name = getName();
person->name = malloc(sizeof(char)*strlen(name));
strcpy(person->name,name);
...   // Do stuff (that isn't buggy)
free(person);
free(person->name);
```

Did not allocate space for the null terminator! Want `(strlen(name)+1)` here.

Accessing memory after you've freed it. These statements should be switched.

Using Memory You Haven't Allocated

- What is wrong with this code?

```
void StringManipulate() {
  const char *name = "Safety Critical";
  char *str = malloc(sizeof (char) * 10);
  strncpy(str, name, 10);
  str[10] = '\0';          ← Write beyond array bounds
  printf("%s\n", str);     ← Read beyond array bounds
}
```

Using Memory You Haven't Allocated

- What is wrong with this code?

```
char buffer[1024]; /* global */
int foo(char *str) {
   strcpy(buffer,str);
   ...
}
```

What if more than a kibi characters?

This is called BUFFER OVERRUN or BUFFER OVERFLOW and is a security flaw!!!

Freeing Invalid Memory

- What is wrong with this code?

```
void FreeMemX() {
   int fnh = 0;
   free(&fnh);        ←— 1) Free of a Stack variable
}
```

```
void FreeMemY() {
   int *fum = malloc(4*sizeof(int));
   free(fum+1);       ←— 2) Free of middle of block
   free(fum);
   free(fum);         ←— 3) Free of already freed block
}
```

Memory Leaks

- What is wrong with this code?

```
int *pi;

void foo() {
    pi = (int*)malloc(8*sizeof(int));
    ...
    free(pi);
}

void main() {
    pi = (int*)malloc(4*sizeof(int));
    foo();
}
```

Overrode old pointer! No way to free those 4*sizeof(int) bytes now

`foo()` leaks memory

Memory Leaks

- Remember that Java has garbage collection but C doesn't
- Memory Leak: when you allocate memory but lose the pointer necessary to free it
- **Rule of Thumb:** More `malloc`s than `free`s probably indicates a memory leak
- Potential memory leak: Changing pointer – do you still have copy to use with `free` later?

```
plk = (int *)malloc(2*sizeof(int));
...
plk++;
```

← Mem Leak! Typically happens through incrementation or reassignment

Agenda

- C Memory Layout
  - Stack, Static Data, and Code
- Administrivia
- Dynamic Memory Allocation
  - Heap
- Common Memory Problems
- C Wrap-up: Linked List Example

Linked List Example

- We want to generate a linked list of strings
  - This example uses structs, pointers, `malloc()`, and `free()`
- Create a structure for nodes of the list:

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

The link of the linked list

Adding a Node to the List

- Want to write addNode to support functionality as shown:

In what part of memory are these stored?

```
char *s1 = "start", *s2 = "middle",
*s3 = "end";
struct node *theList = NULL;
theList = addNode(s3, theList);
theList = addNode(s2, theList);
theList = addNode(s1, theList);
```

Must be able to handle a NULL input

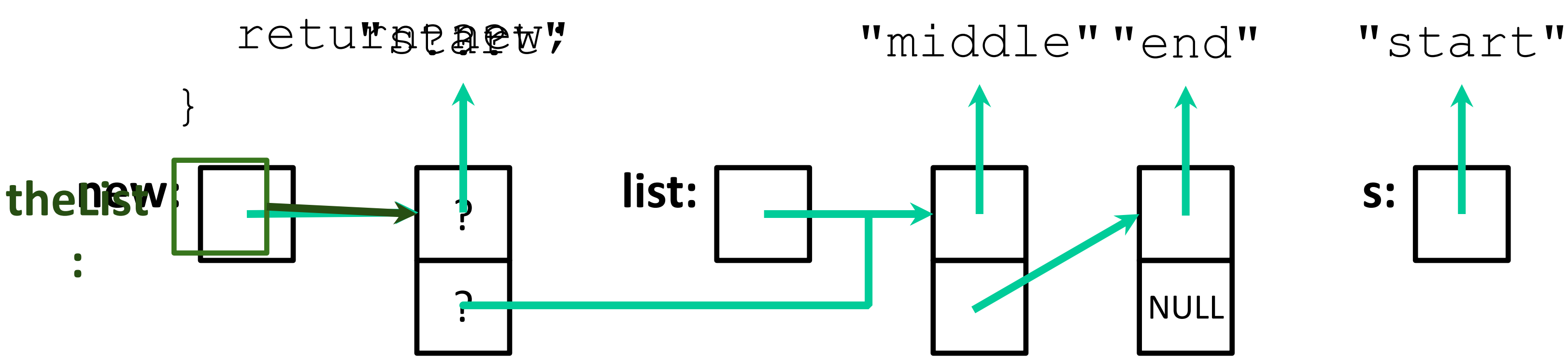
If you're more familiar with Lisp/Scheme, you could name this function `cons` instead.

Adding a Node to the List

- Let's examine the 3rd call ("`start`"):

```
node *addNode(char *s, node *list) {
    node *new =(node *)
malloc(sizeof(NodeStruct));
    new->value = (char *) malloc (strlen(s) +
1);
    strcpy(new->value, s);
    new->next = list;
    return new;
}
```

Don't forget this for the null terminator!

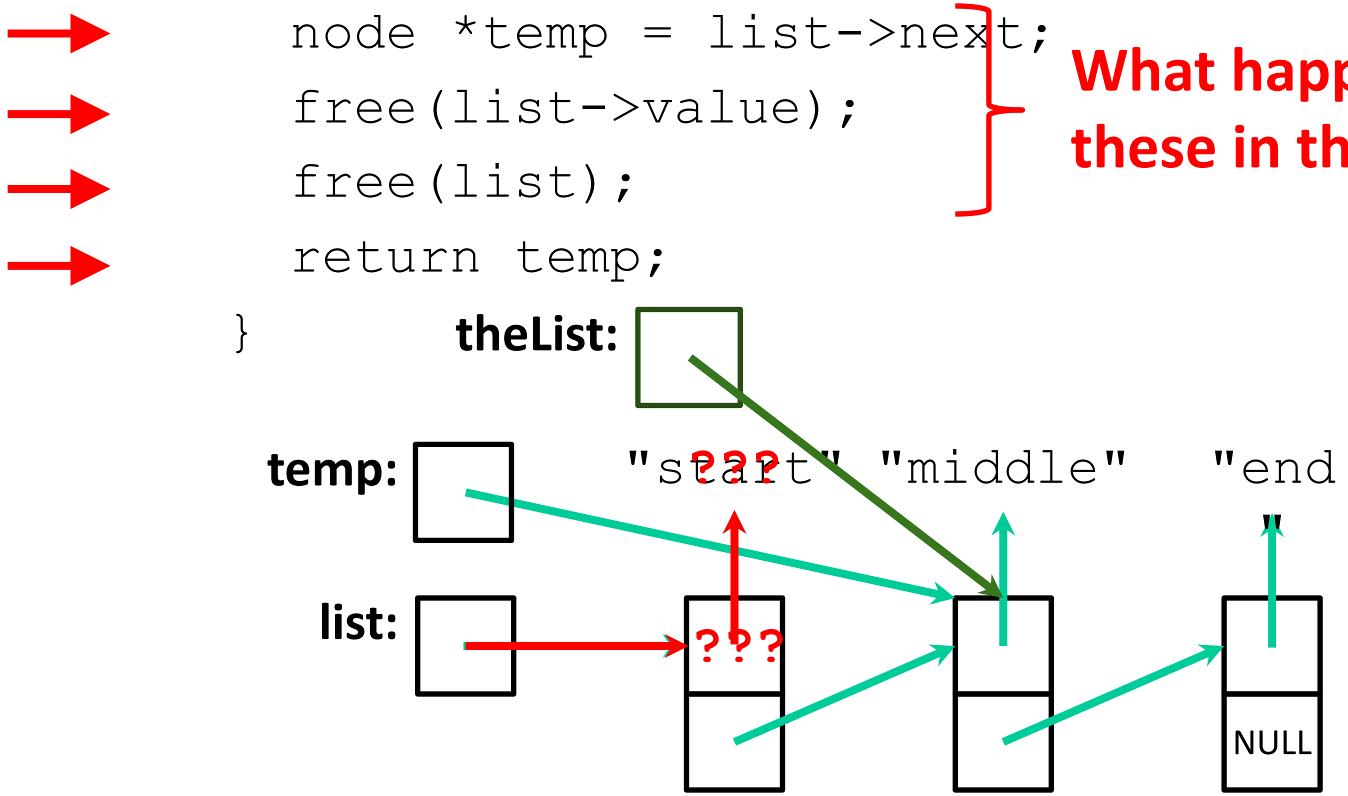


Removing a Node from the List

- Delete/free the first node (`"start"`):

```
node *deleteNode(node *list) {
   node *temp = list->next;
   free(list->value);
   free(list);
   return temp;
}
```

What happens if you do these in the wrong order?

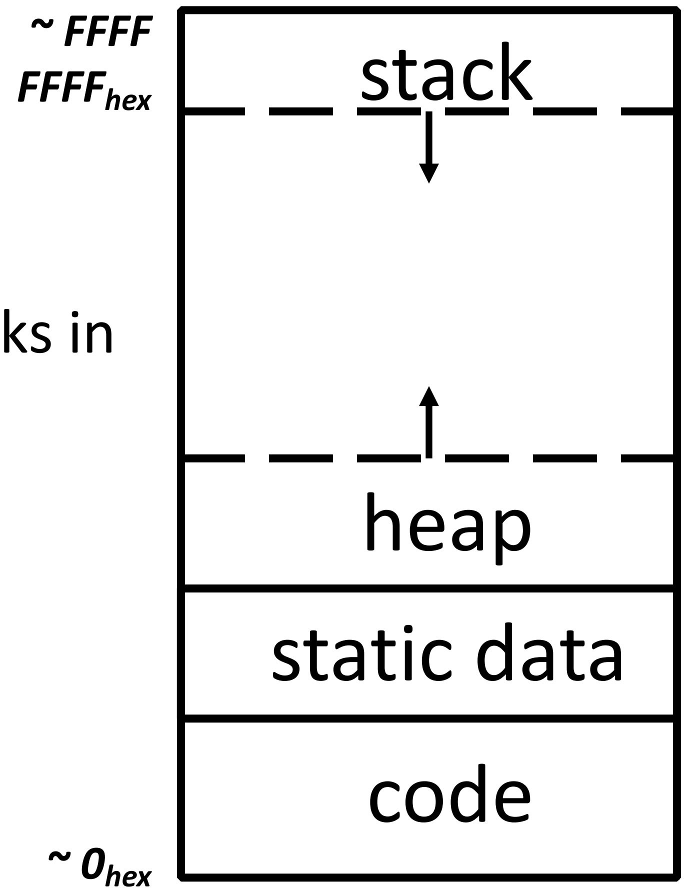


Additional Functionality

- How might you implement the following?
  - Append node to end of a list
  - Delete/free an entire list
  - Join two lists together
  - Reorder a list alphabetically (sort)

Summary

- C Memory Layout
  - **Stack:** local variables (grows & shrinks in LIFO manner)
  - **Static Data:** globals and string literals
  - **Code:** copy of machine code
  - **Heap:** dynamic storage using `malloc` and `free`
    The source of most memory bugs!
- Common Memory Problems
- Last C Lecture!



OS prevents accesses between stack and heap (via virtual memory)