C: Memory Management and Usage

Review

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

Great Idea #1: Levels of Representation/Interpretation

Higher-Level Language Program (e.g. C)

```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

This week

Compiler

Assembly Language Program (e.g. MIPS)

```
lw   $t0, 0($2)
lw   $t1, 4($2)
sw   $t1, 0($2)
sw   $t0, 4($2)
```

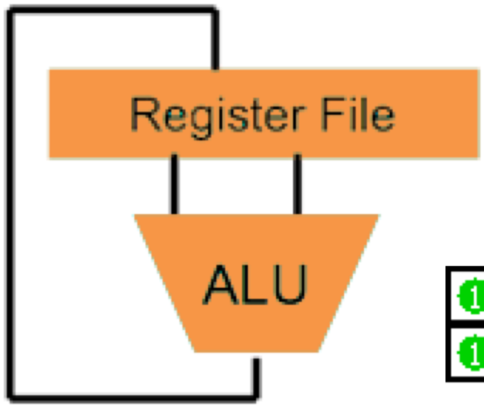
Assembler

Machine Language Program (MIPS)

0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111

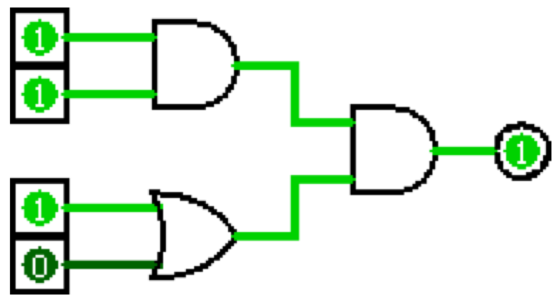
Machine Interpretation

Hardware Architecture Description (e.g. block diagrams)



Architecture Implementation

Logic Circuit Description (Circuit Schematic Diagrams)

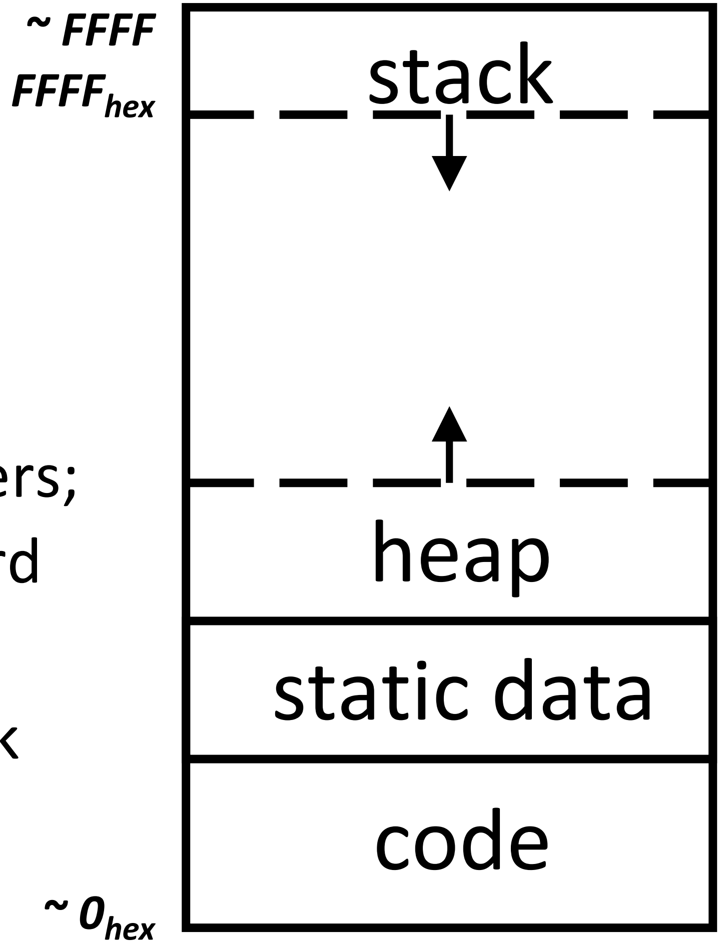


Agenda

- 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)

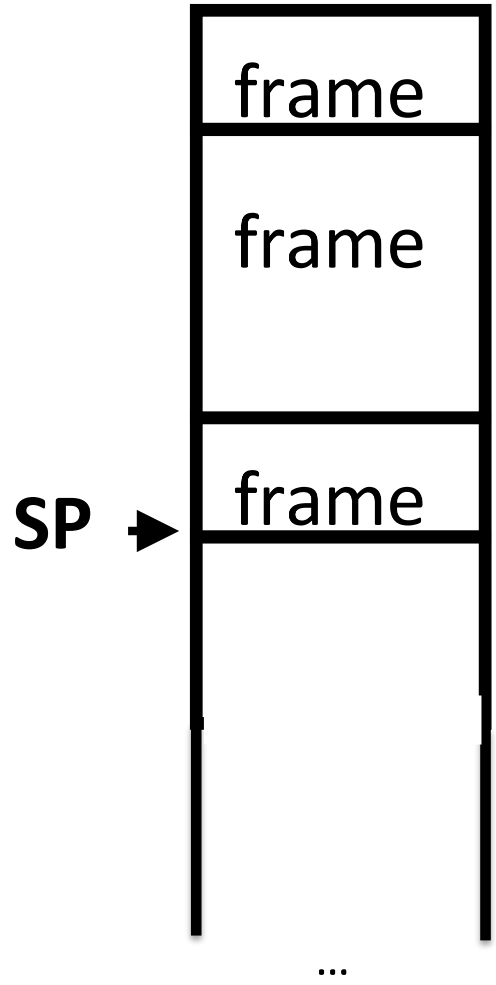
Where Do the Variables Go?

- 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));
}
```

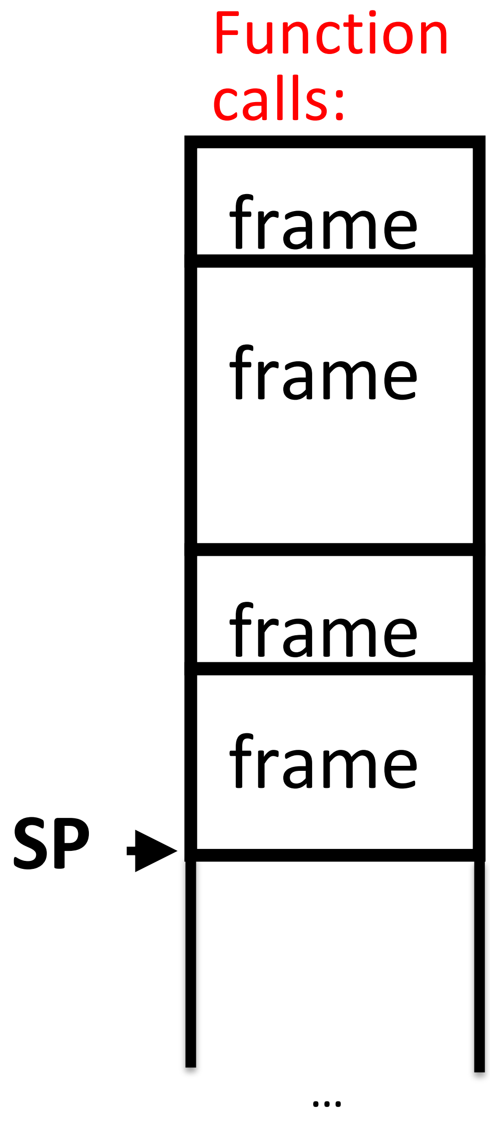
The Stack

- 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



The Stack

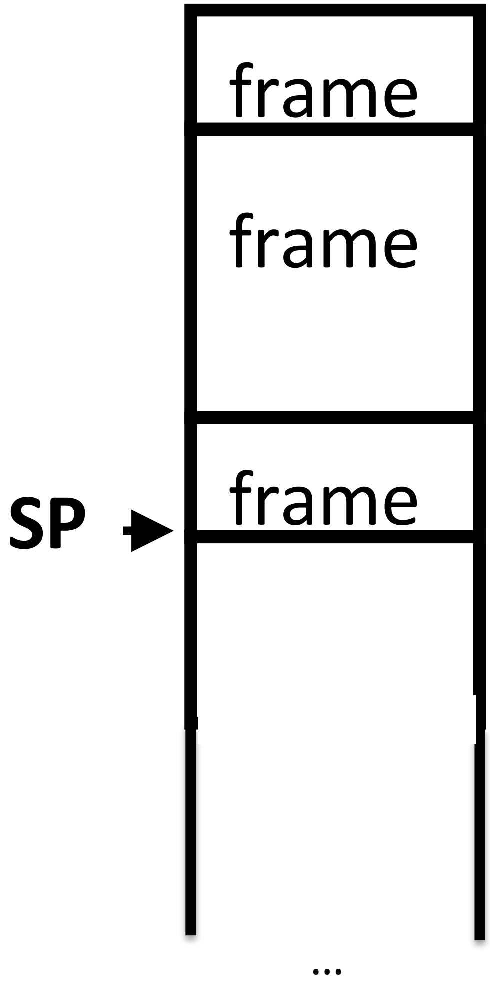
- Each stack frame is a contiguous block of memory holding the local variables of a single procedure
- A stack frame includes:
  - Location of caller function
  - Function arguments
  - Space for local variables
- Stack pointer (SP) tells where lowest (current) stack frame is



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;

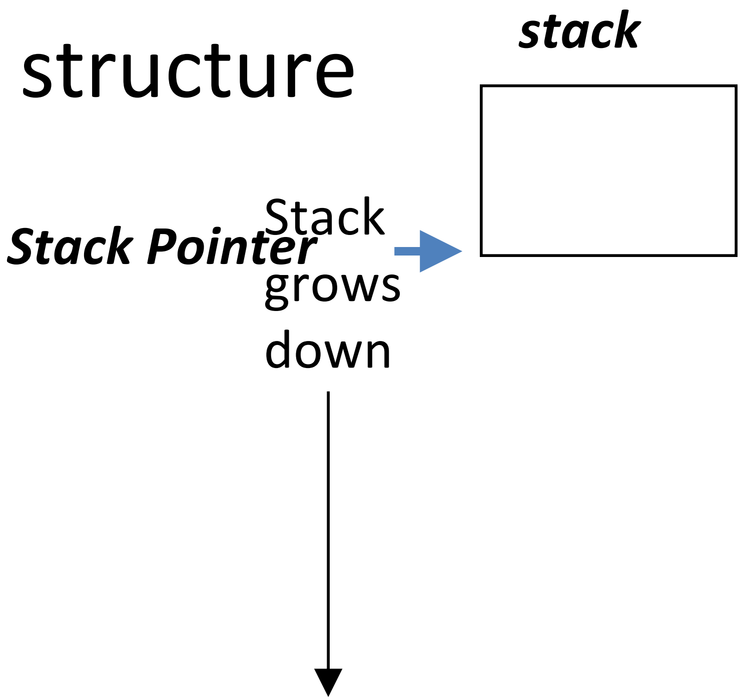
Function returns:



The Stack

- Last In, First Out (LIFO) data structure

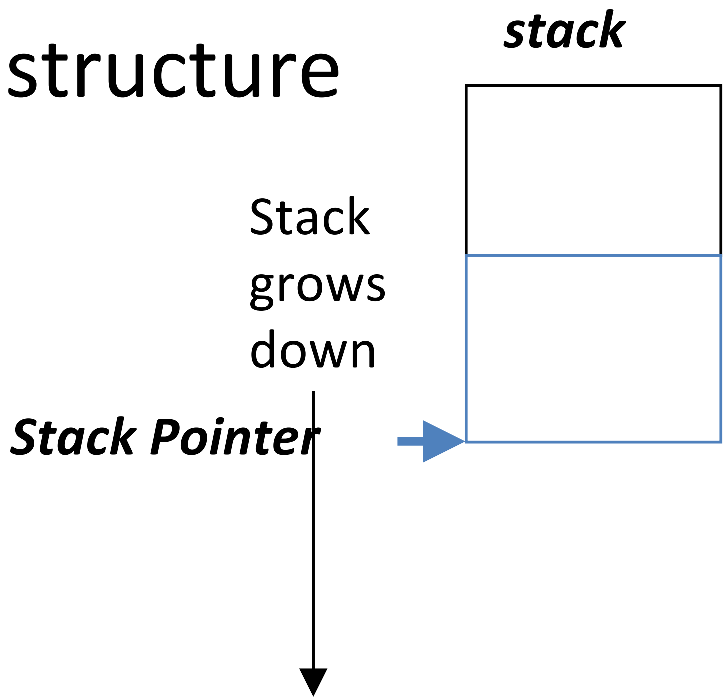
```
int main() {
   a(0);
   return 1; }
void a(int m) {
   b(1); }
void b(int n) {
   c(2);
   d(4); }
void c(int o) {
   printf("c"); }
void d(int p) {
   printf("d"); }
```



The Stack

- Last In, First Out (LIFO) data structure

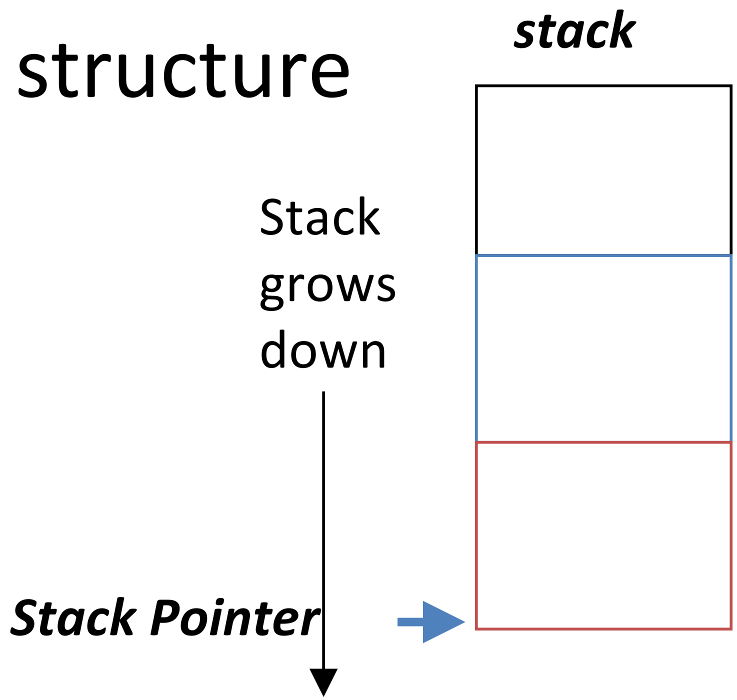
```
int main() {
   a(0);
   return 1; }
void a(int m) {
   b(1); }
void b(int n) {
   c(2);
   d(4); }
void c(int o) {
   printf("c"); }
void d(int p) {
   printf("d"); }
```



The Stack

- Last In, First Out (LIFO) data structure

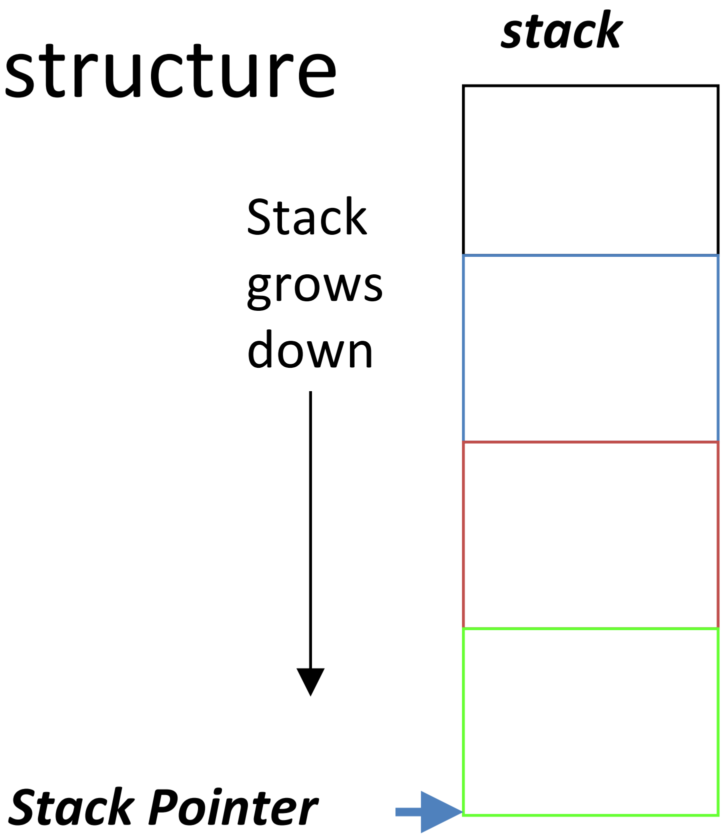
```
int main() {
  a(0);
  return 1; }
void a(int m) {
  b(1); }
void b(int n) {
  c(2);
  d(4); }
void c(int o) {
  printf("c"); }
void d(int p) {
  printf("d"); }
```



The Stack

- Last In, First Out (LIFO) data structure

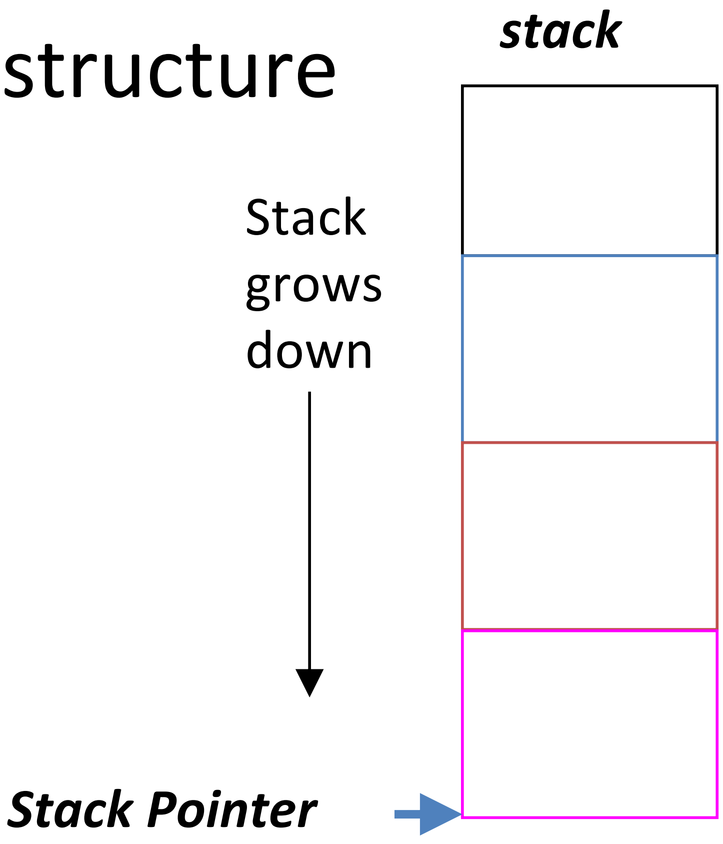
```
int main() {
   a(0);
   return 1; }
void a(int m) {
   b(1); }
void b(int n) {
   c(2);
   d(4); }
void c(int o) {
   printf("c"); }
void d(int p) {
   printf("d"); }
```



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"); }
```



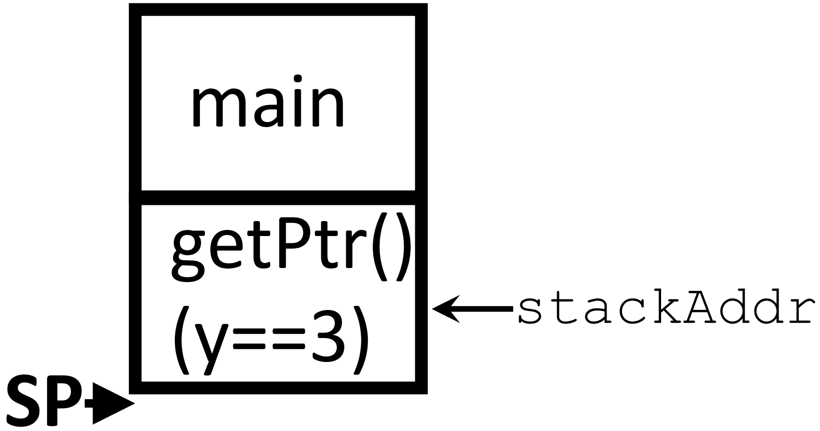
Stack Misuse Example

```
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); /* 0 */
};
```

What's BAD about this function?

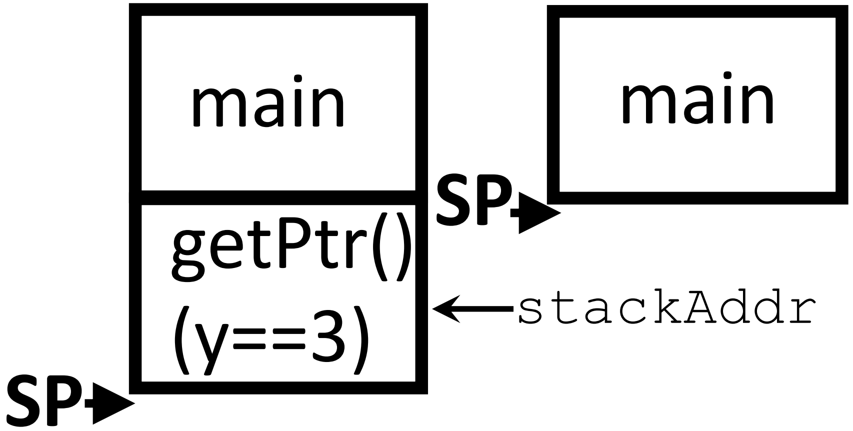
Stack Misuse Example

```
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); /* 0 */
};
```



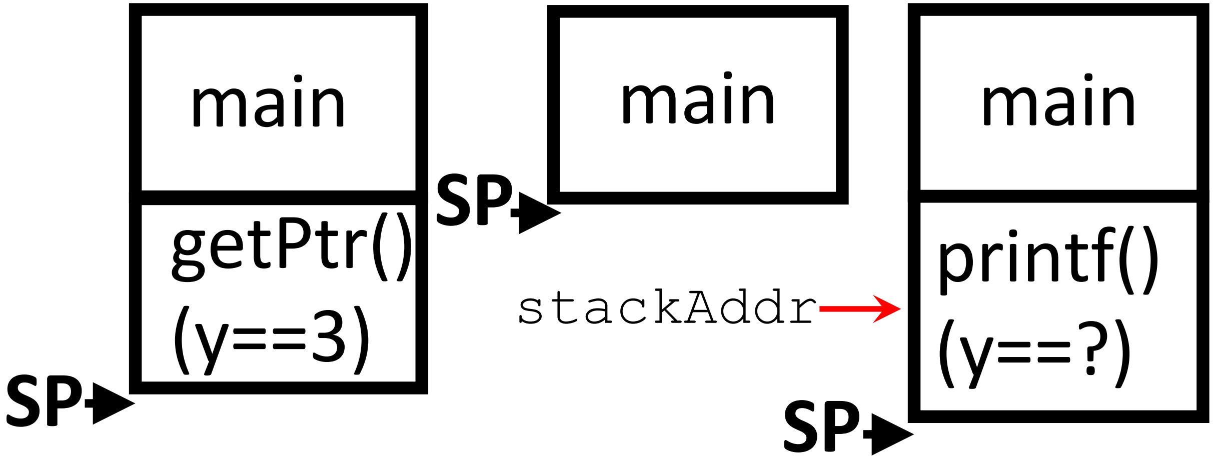
Stack Misuse Example

```
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); /* 0 */
};
```



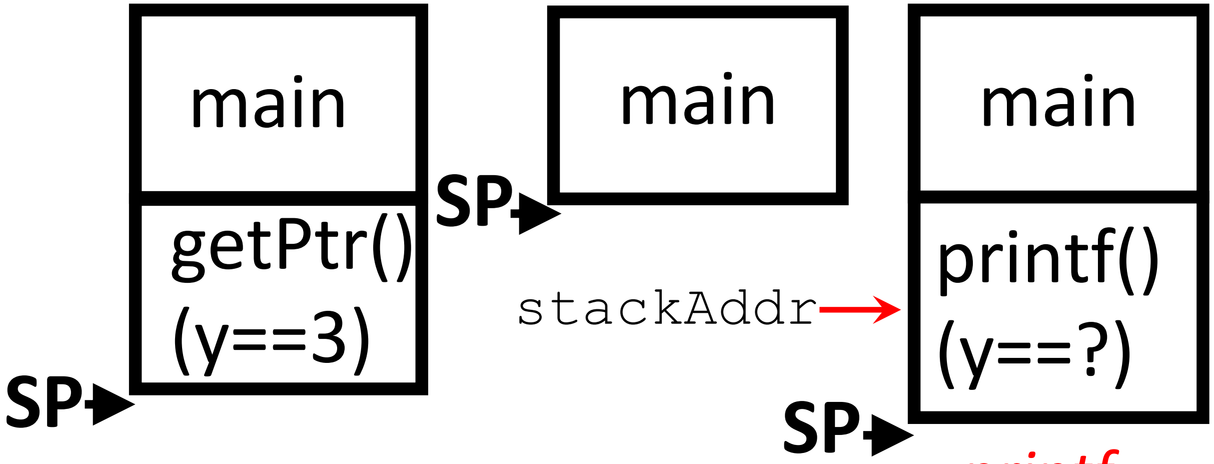
Stack Misuse Example

```
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); /* 0 */
};
```



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); /* 0 */
};
```



printf overwrites stack frame

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); /* 0 */
};
```

Never return pointers to local variable from functions

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

Static Data

- Place for variables that persist
  - Data not subject to comings and goings like function calls
  - Examples: String literals, global variables
- Size does not change, but its data can

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

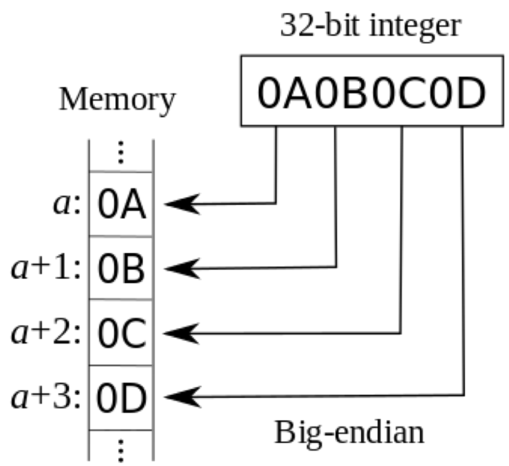
Note: We're talking about `*s`, not `s`, i.e. the location where `s` points!

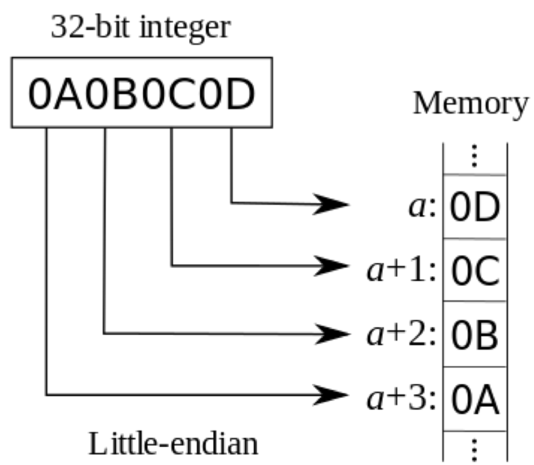
Addresses

- The size of an address (and thus, the size of a pointer) in bytes depends on architecture (eg: 32-bit Windows, 64-bit Mac OS)
  - eg: for 32-bit, have 2^{32} possible addresses
- If a machine is **byte-addressed**, then each of its addresses points to a unique **byte**
  - word-addresses = address points to a word
- Question: on a byte-addressed machine, how can we order the bytes of an integer in mem?
  - Answer: it depends

Endianness

- Big Endian:
  - Descending numerical significance with ascending memory addresses
- Little Endian
  - Ascending numerical significance with ascending memory addresses

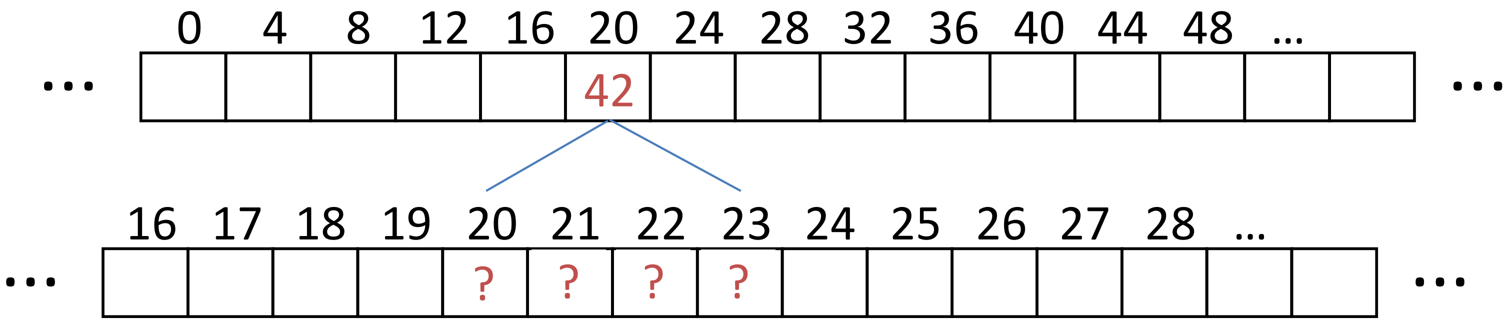




Source: https://en.wikipedia.org/wiki/Endianness

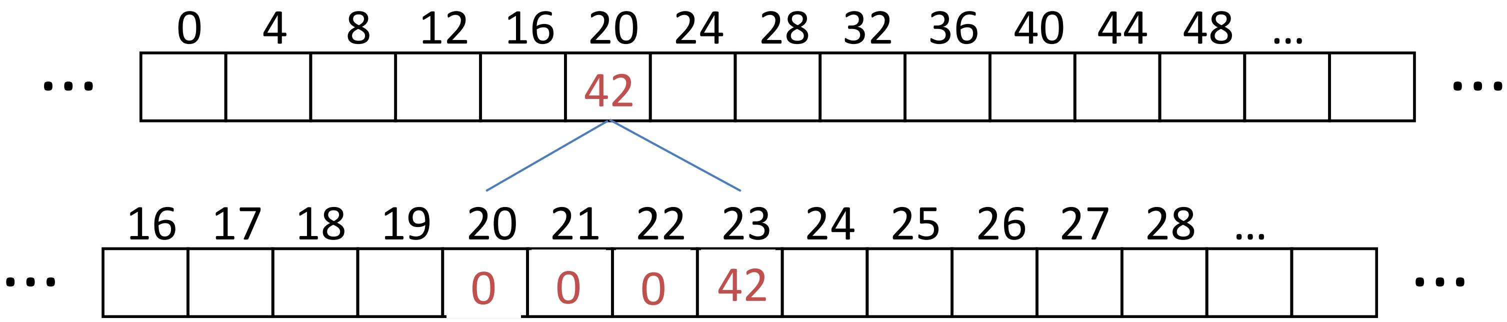
Endianness

- In what order are the bytes within a data type stored in memory?



Endianness

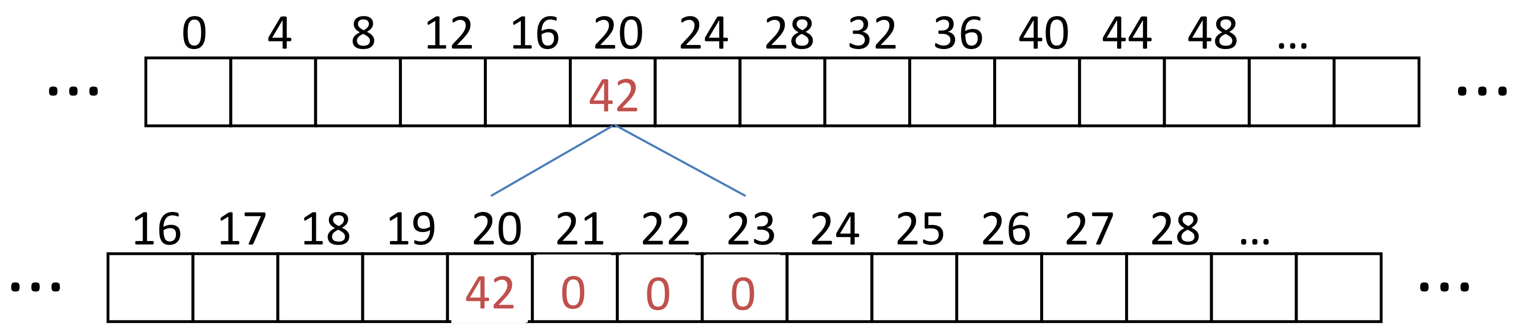
- In what order are the bytes within a data type stored in memory?



- Big Endian:
  – Descending numerical significance with ascending memory addresses

Endianness

- In what order are the bytes within a data type stored in memory?



- Big Endian:
  – Descending numerical significance with ascending memory addresses
- Little Endian
  – Ascending numerical significance with ascending memory addresses

Common Mistakes

- Endianness ONLY APPLIES to values that occupy multiple bytes
- Endianness refers to STORAGE IN MEMORY NOT number representation
- Ex: char c = 97
  - c == 0b01100001 in both big and little endian
- Arrays and pointers still have the same order
  - int a[5] = {1, 2, 3, 4, 5} (assume address 0x40)
  - &(a[0]) == 0x40 && a[0] == 1
    - in both big and little endian

Agenda

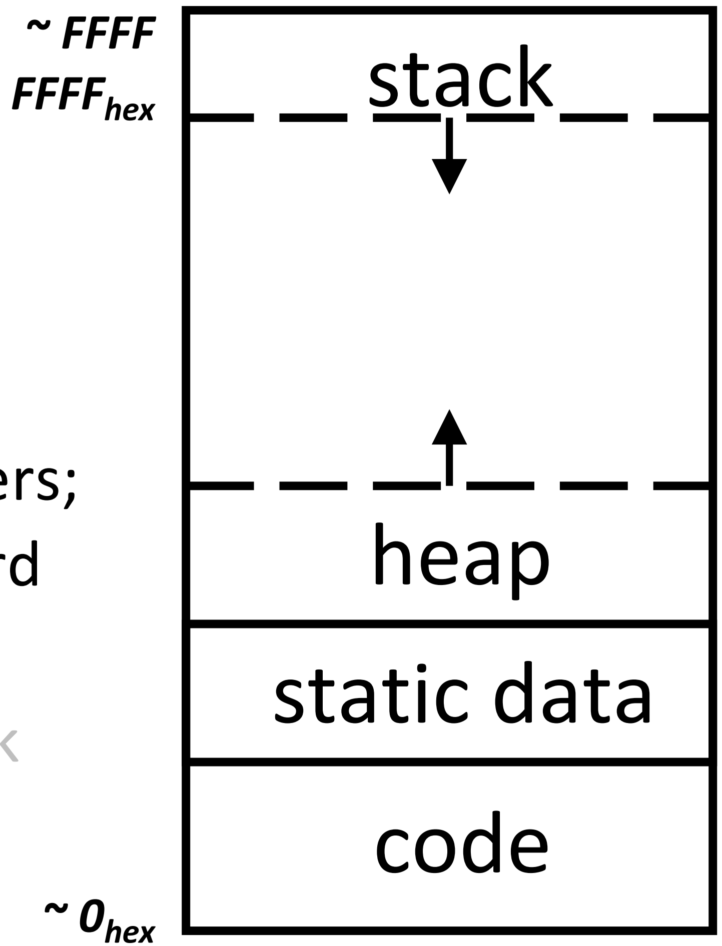
- C Memory Layout
  - Stack, Static Data, and Code
- Dynamic Memory Allocation
  - Heap
- Common Memory Problems
- C Wrap-up: Linked List Example

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)

Dynamic Memory Allocation

- Sometimes you don't know how much memory you need beforehand
  - e.g. input files, user input
- 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()` operator
  - Returns size <u>in bytes</u> of variable or data type name

    Examples: `int x; sizeof(x); sizeof(int);`

Allocating Memory

- 3 functions for requesting memory: `malloc()`, `calloc()`, and `realloc()`
  - http://en.wikipedia.org/wiki/C_dynamic_memory_allocation#Overview_of_functions
- **`malloc(`*`n`*`)`**
  - Allocates a continuous block of ***n* bytes** of uninitialized memory (contains garbage!)
  - Returns a pointer to the beginning of the allocated block; NULL indicates failed request (check for this!)
  - Different blocks not necessarily adjacent

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
- Make sure you don't lose the original address
  - `p++` is a **BAD IDEA**; use a separate pointer

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

Calloc

- `void *calloc(size_t nmemb, size_t size)`
  - Like `malloc`, except it initializes the memory to 0
  - `nmemb` is the number of members
  - `size` is the size of each member
  - Ex for allocating space for 5 integers

```
int *p = (int *)calloc(5, sizeof (int));
```

Realloc

- What happens when I need more or less memory in an array
- `void *realloc(void *ptr, size_t size)`
  - Takes in a `ptr` that has been the return of `malloc`/`calloc`/`realloc` and a new `size`
  - Returns a pointer with now size space (or `NULL`) and copies any contents from `ptr`
- `Realloc` can move or keep the address the same
- DO NOT rely on old `ptr` values

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);
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

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
- Dynamic Memory Allocation
  - Heap
- Common Memory Problems
- C Wrap-up: Linked List Example

Know Your Memory Errors

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

- Segmentation Fault ← More common in this course

  "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 this course

  "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) Using memory you haven’t allocated
4) Freeing invalid memory
5) 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 (j=0; 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

Pointer to Stack (array) no longer valid once function returns

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(10);
  strncpy(str, name, 10);
  str[10] = '\0';
  printf("%s\n", str);
}
```

Write beyond array bounds

Read beyond array bounds

Using Memory You Haven't Allocated

- What is wrong with this code?

```
char buffer[1024]; /* global */

int main(int argc,char *argv[]) {
   strcpy(buffer,argv[1]);
   ...
}
```

What if more than
a kibi characters?

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

C String Standard Functions Revised

- Accessible with `#include <string.h>`
- `int strnlen(char *string,size_t n);`
  - Returns the length of string (not including null term), searching up to n
- `int strncmp(char *str1, char *str2, size_t n);`
  - Return 0 if `str1` and `str2` are identical (how is this different from `str1 == str2`?), comparing up to n bytes
- `char *strncpy(char *dst, char *src, size_t n);`
  - Copy up to the first n bytes of string `src` to the memory at `dst`. Caller must ensure that `dst` has enough memory to hold the data to be copied
  - Note: `dst = src` only copies *pointer* (the address)

A Safer Version

```
#define ARR_LEN 1024;
char buffer[ARR_LEN]; /* global */

  int foo(char *str) {
    strncpy(buffer,str, ARR_LEN);
    ...
  }
```

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();
}
```

Overwrite 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++;
```

← Typically happens through incrementation or reassignment

Debugging Tools

- Runtime analysis tools for finding memory errors
  - Dynamic analysis tool: Collects information on memory management while program runs
  - No tool is guaranteed to find ALL memory bugs; this is a very challenging programming language research problem

http://valgrind.org

Agenda

- C Memory Layout
  - Stack, Static Data, and Code
- 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;
};
```

The link of the linked list

Simplify Code with `typedef`

- It gets annoying to type out `struct ListNode` over and over again
  - Define new variable type for struct:

Method 1:

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

Method 2:

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

- Can further rename pointers:

```
typedef ListNode * List;
typedef char * String;
```

```
List myLinkedList;
String value;
```

Adding a Node to the List

- Want functionality as shown:

In what part of memory are these stored?

```
String s1 = "start", s2 = "middle";
String s3 = "end";
List theList = NULL;
theList = addNode(s3, theList);
theList = addNode(s2, theList);
theList = addNode(s1, theList);
```

Must be able to handle a NULL input

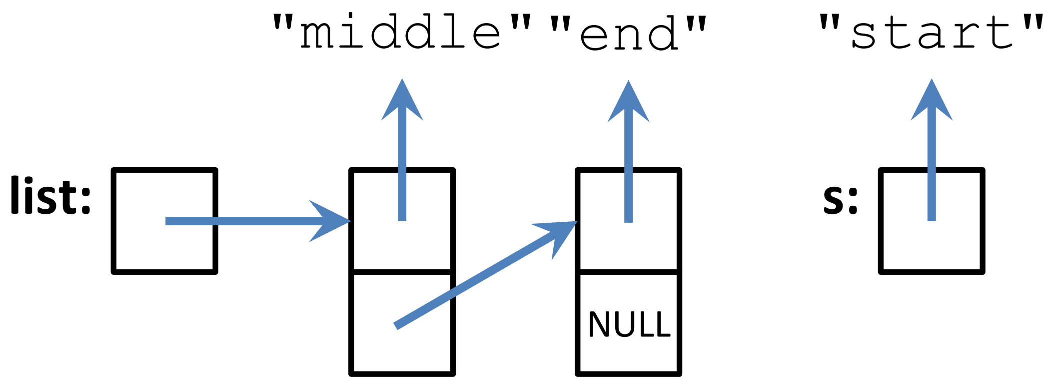
If you're more familiar with Lisp/Scheme, you could name this function `cons` instead.

Adding a Node to the List

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

```
List addNode(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;
}
```

Don't forget this for the null terminator!

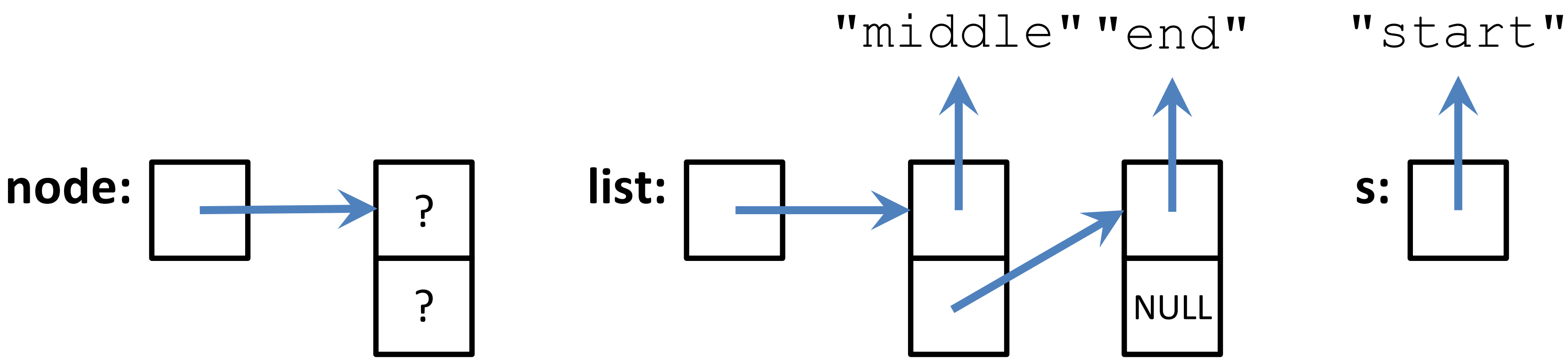


Adding a Node to the List

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

```
List addNode(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;
}
```

Don't forget this for the null terminator!

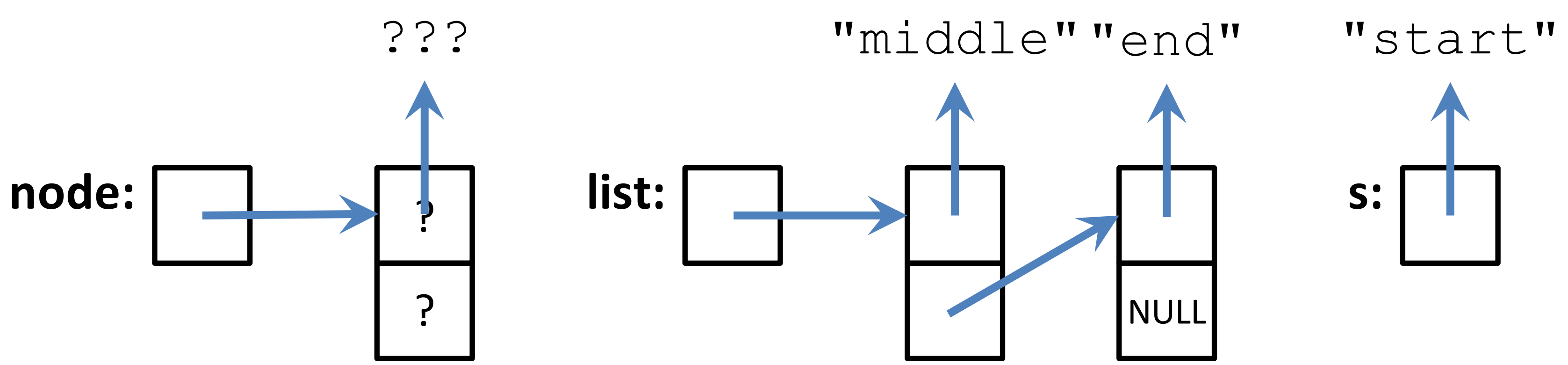


Adding a Node to the List

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

```
List addNode(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;
}
```

Don't forget this for the null terminator!

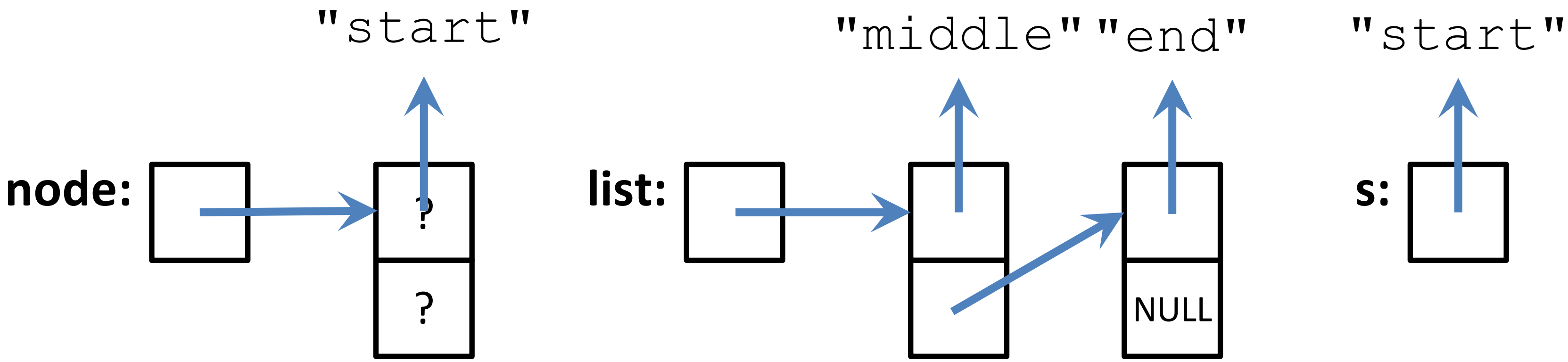


Adding a Node to the List

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

```
List addNode(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;
}
```

Don't forget this for the null terminator!

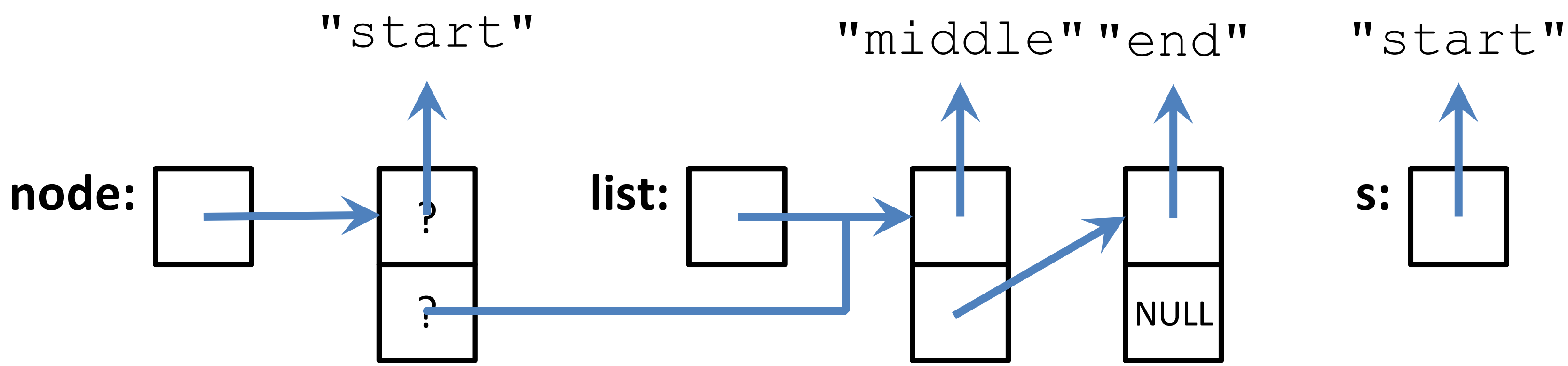


Adding a Node to the List

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

```
List addNode(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;
}
```

Don't forget this for the null terminator!

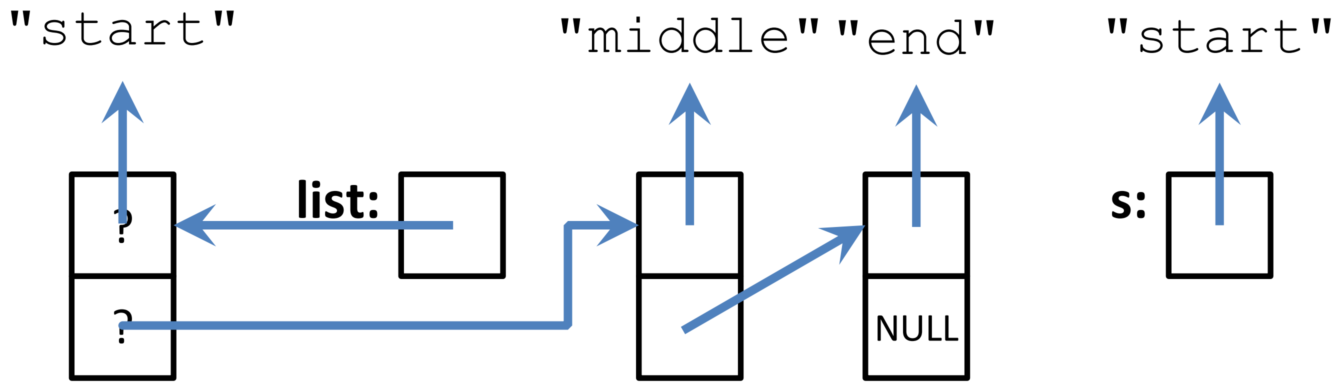


Adding a Node to the List

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

```
List addNode(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;
}
```

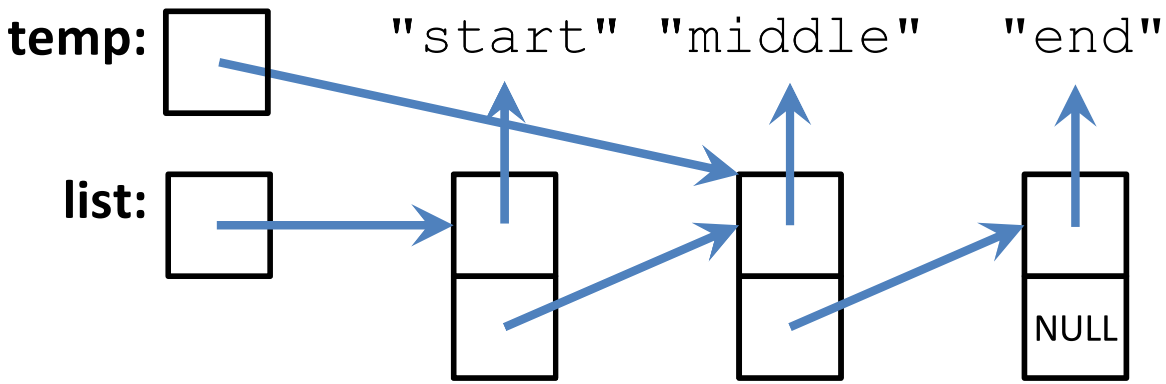
Don't forget this for the null terminator!



Removing a Node from the List

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

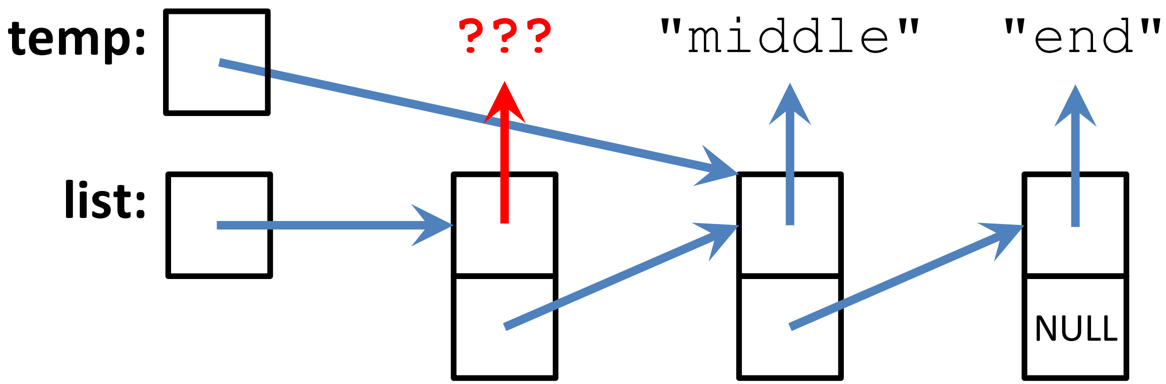
```
List deleteNode(List list) {
  List temp = list->next;
  free(list->value);
  free(list);
  return temp;
}
```



Removing a Node from the List

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

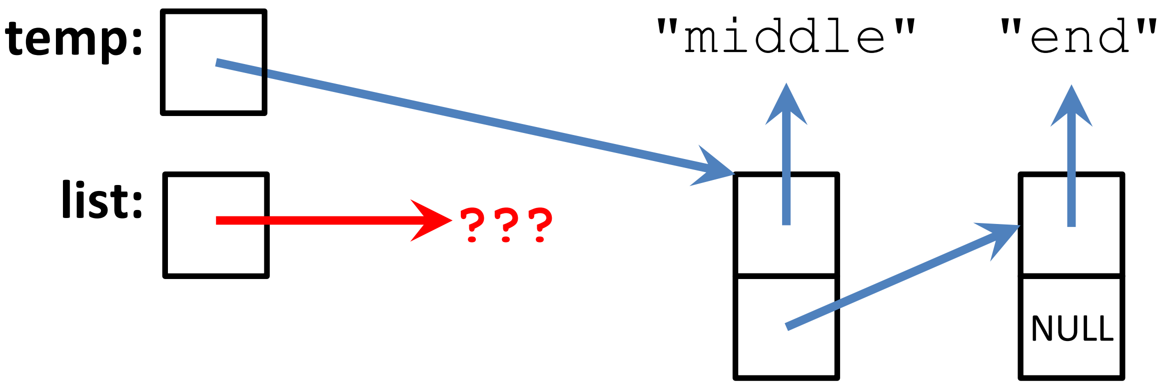
```
List deleteNode(List list) {
  List temp = list->next;
  free(list->value);
  free(list);
  return temp;
}
```



Removing a Node from the List

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

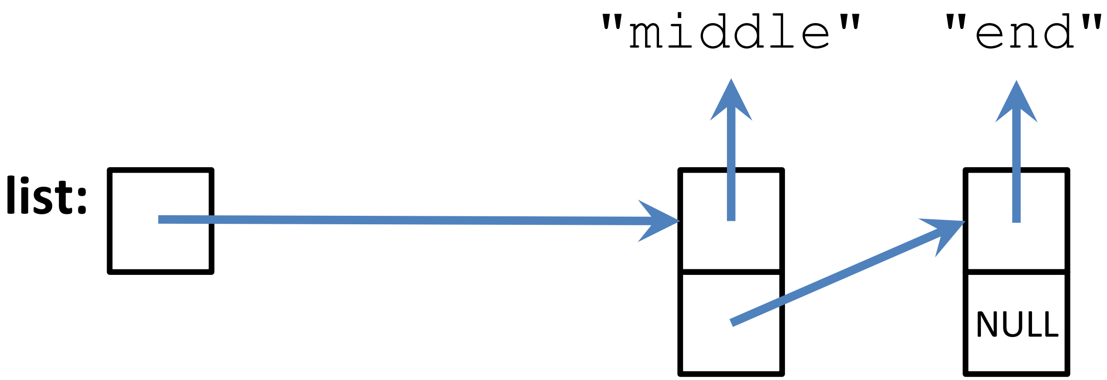
```
List deleteNode(List list) {
  List temp = list->next;
  free(list->value);
  free(list);
  return temp;
}
```



Removing a Node from the List

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

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

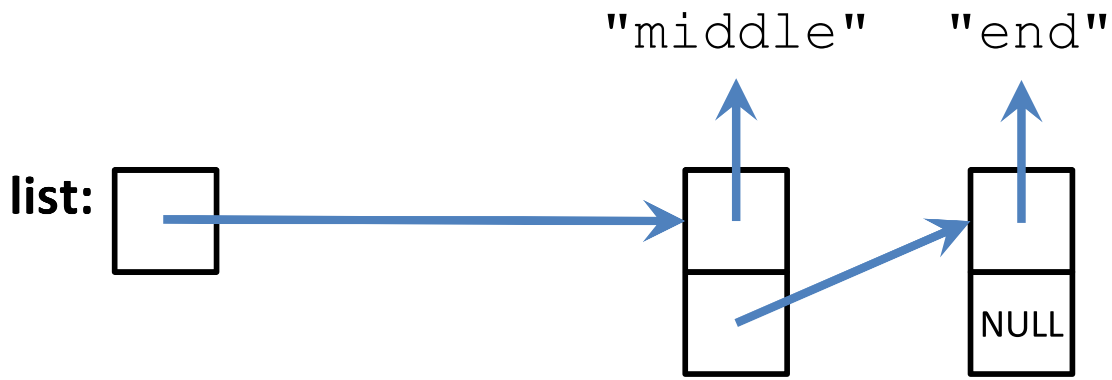


Removing a Node from the List

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

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

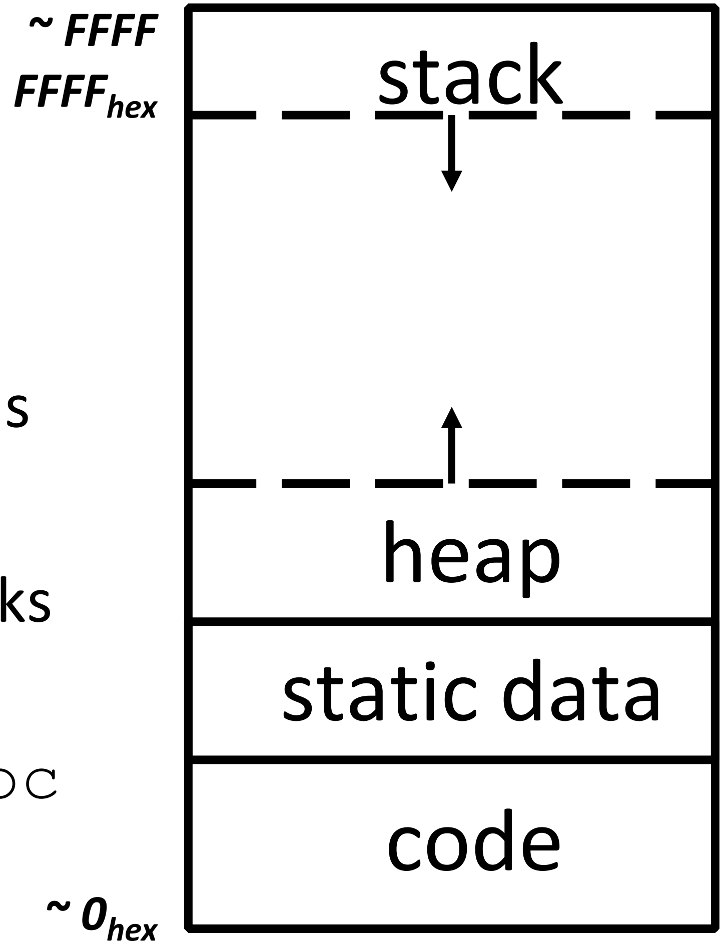
What happens if you do these in the wrong order?



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



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

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

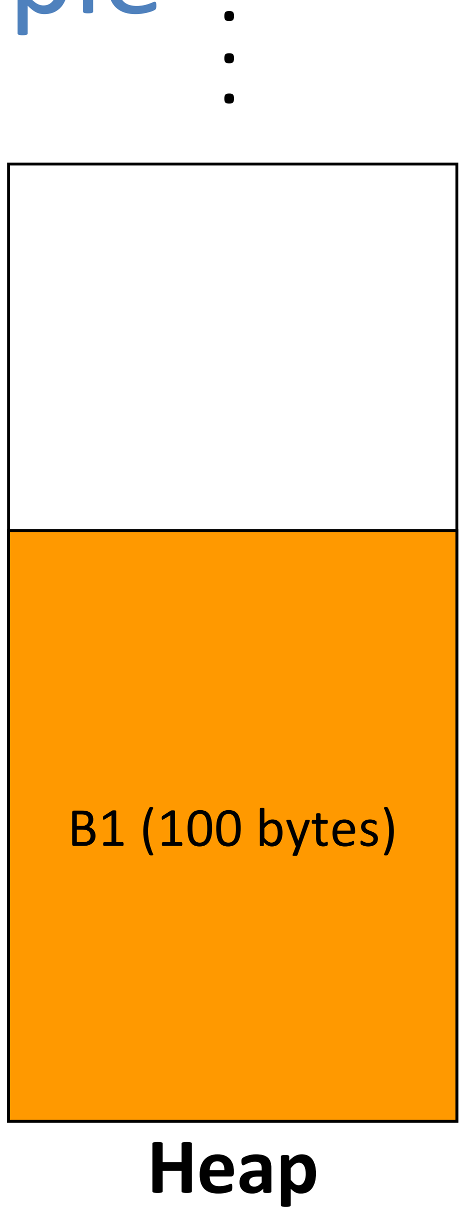
Bonus Slides

Memory Management

- Many calls to `malloc()` and `free()` with many different size blocks – where are they placed?
- Want system to be fast with minimal memory overhead
  - Versus automatic garbage collection of Java
- Want to avoid *fragmentation*, the tendency of free space on the heap to get separated into small chunks

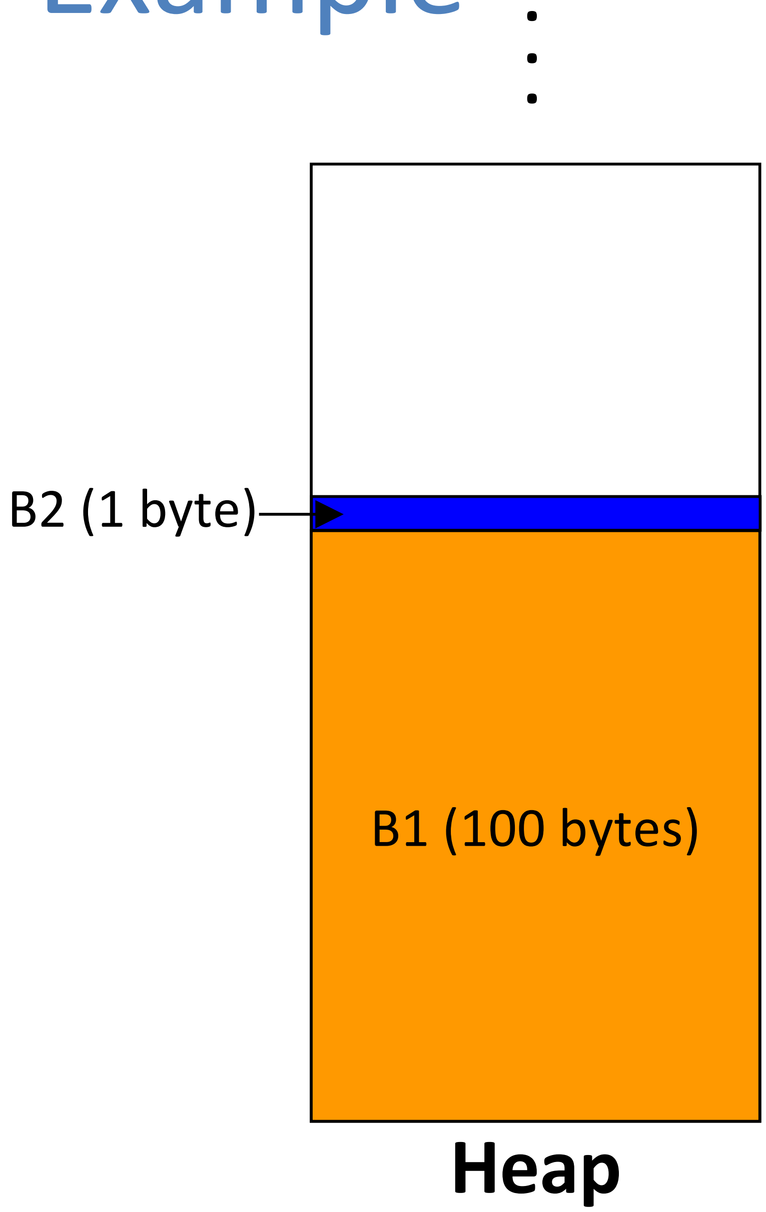
Fragmentation Example

1) Block 1: malloc(100)



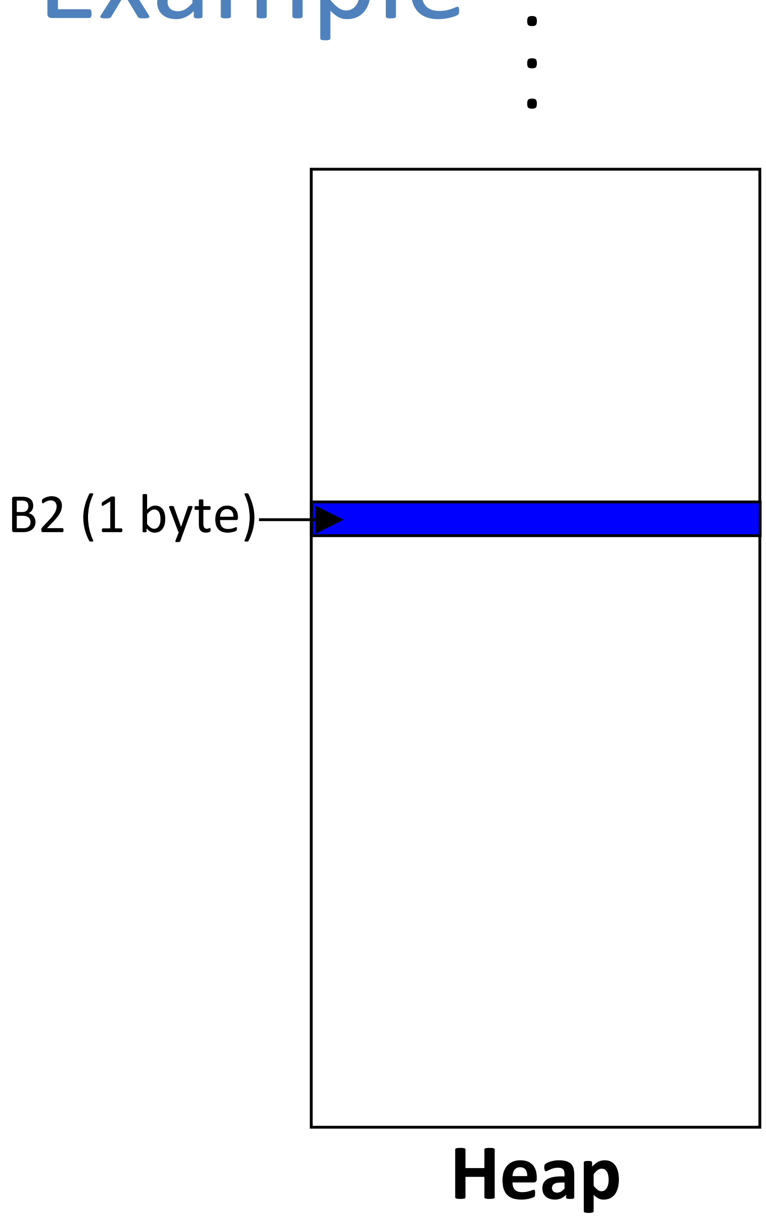
Fragmentation Example

1) Block 1: malloc(100)
2) Block 2: malloc(1)



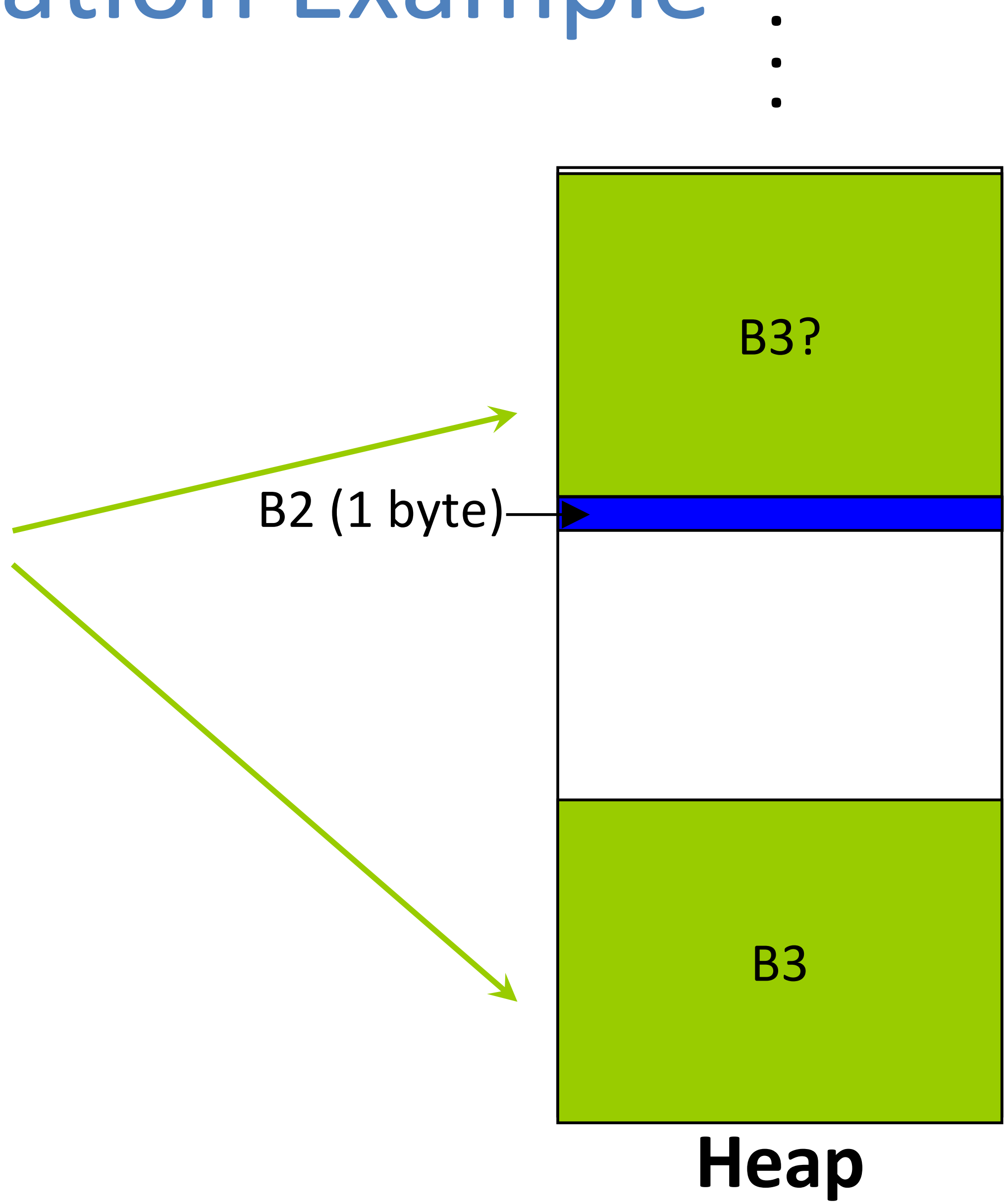
Fragmentation Example

1) Block 1: malloc(100)
2) Block 2: malloc(1)
3) Block 1: free(B1)



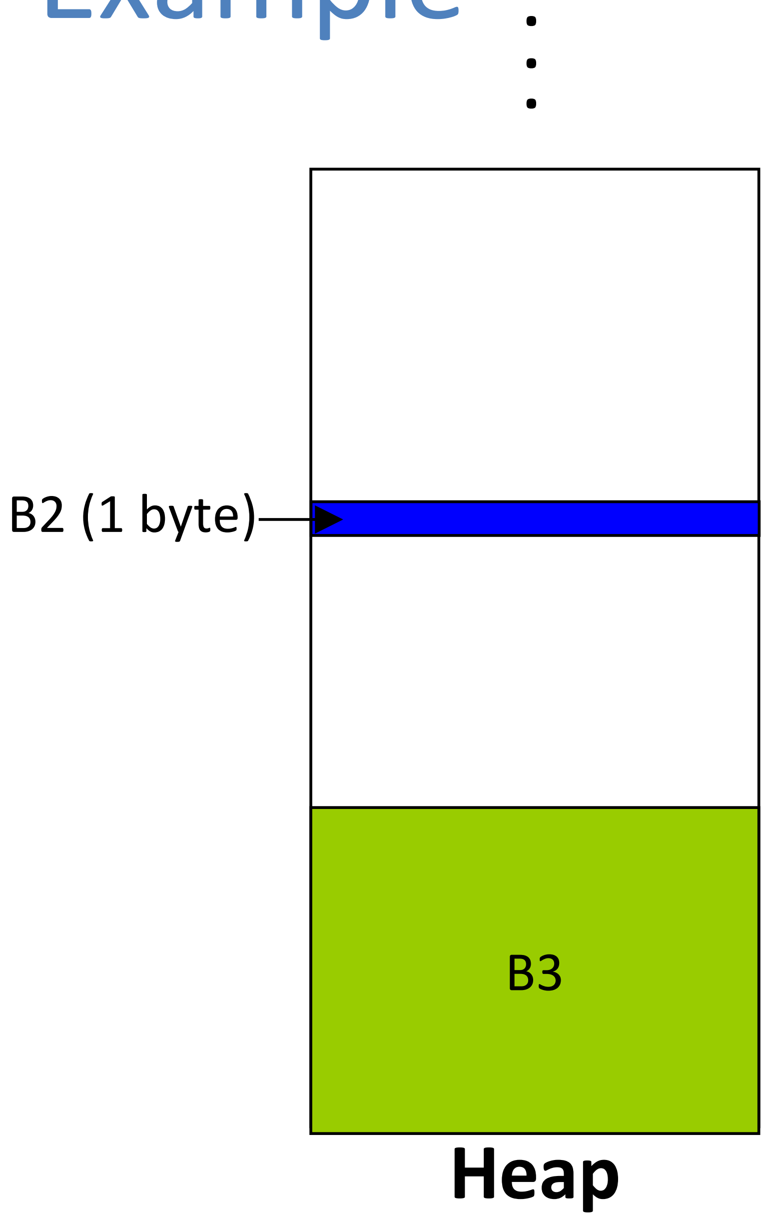
Fragmentation Example

1) Block 1: malloc(100)
2) Block 2: malloc(1)
3) Block 1: free(B1)
4) Block 3: malloc(50)
   – What if malloc(101)?



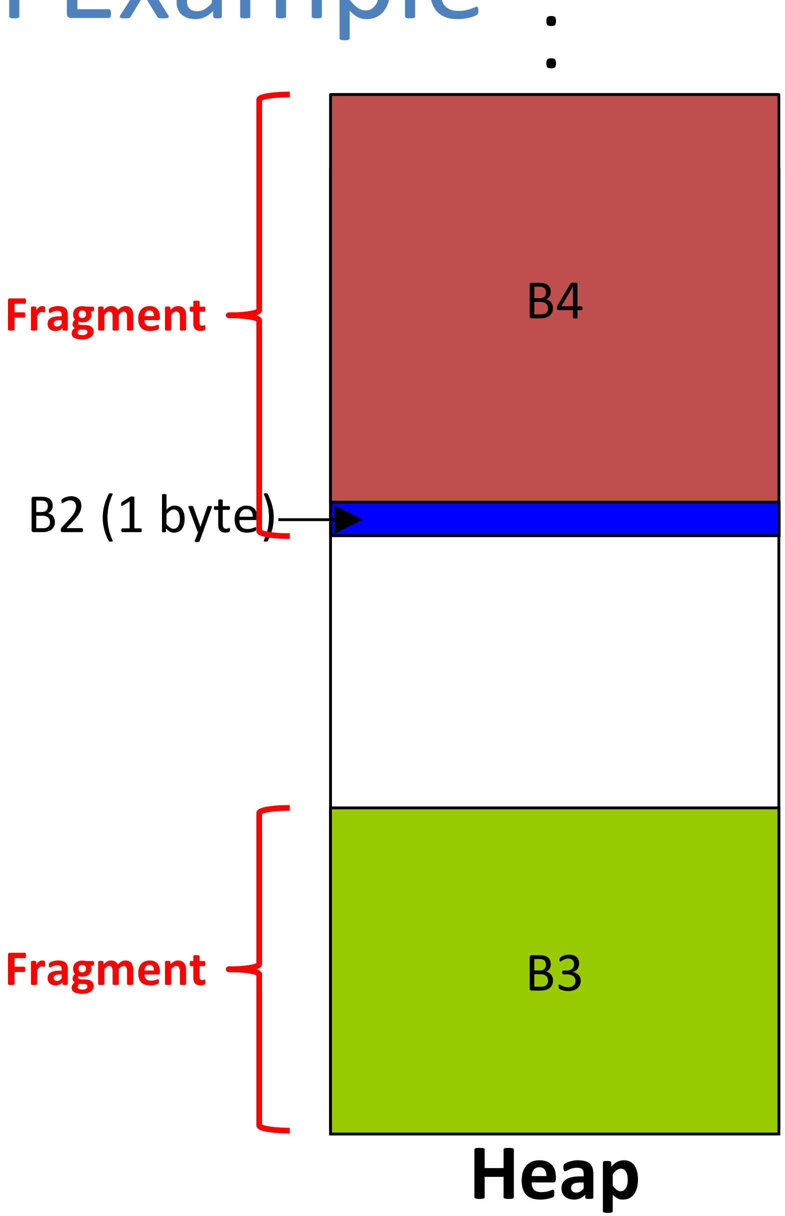
Fragmentation Example

1) Block 1: malloc(100)
2) Block 2: malloc(1)
3) Block 1: free(B1)
4) Block 3: malloc(50)
   – What if malloc(101)?



Fragmentation Example

1) Block 1: malloc(100)
2) Block 2: malloc(1)
3) Block 1: free(B1)
4) Block 3: malloc(50)
   - What if malloc(101)?
5) Block 4: malloc(60)



Basic Allocation Strategy: K&R

- Section 8.7 offers an implementation of memory management (linked list of free blocks)
  - If you can decipher the code, you're well-versed in C!
- This is just one of many possible memory management algorithms
  - Just to give you a taste
  - No single best approach for every application

K&R Implementation

- Each block holds its own size and pointer to next block
- `free()` adds block to the list, combines with adjacent free blocks
- `malloc()` searches free list for block large enough to meet request
  - If multiple blocks fit request, which one do we use?

Choosing a Block in malloc()

- **Best-fit:** Choose smallest block that fits request
  - Tries to limit wasted fragmentation space, but takes more time and leaves lots of small blocks
- **First-fit:** Choose first block that is large enough (always starts from beginning)
  - Fast but tends to concentrate small blocks at beginning
- **Next-fit:** Like first-fit, but resume search from where we last left off
  - Fast and does not concentrate small blocks at front