CS 61C: Great Ideas in Computer Architecture (Machine Structures)
Memory Management and Malloc

Instructors:
Michael Greenbaum
http://inst.eecs.Berkeley.edu/~cs61c/su11

Agenda

- Pointers Review
- C Memory Management
- Administrivia
- Malloc and Free
- Break
- Common Memory Problems

Pointer Arithmetic

pointer + *number*, *pointer* – *number*

E.g., *pointer* + `1` adds 1 something to a pointer

(in both examples, pretend variable **a** sits at address 100)

```
char  *p;
char   a;

p = &a;
```

```
int   *p;
int    a;

p = &a;
```

What does the following line yield?

```
printf("%u %u\n",p,p+1);
```

`100 101`

Adds `1*sizeof(char)` to the memory address

`100 104`

Adds `1*sizeof(int)` to the memory address

Pointer arithmetic should be used cautiously

Pointer Arithmetic

- `p+1` correctly computes a ptr to the next array element automatically depending on `sizeof(type)`
- What if we have an array of large structs (objects)?
  - C takes care of it in the same way it handles basic types.

Pointer Arithmetic to Copy Memory

- We can use pointer arithmetic to "walk" through memory:

```
void copy(int *from, int *to, int n)
    {
      int i;
      for (i=0; i<n; i++) {
           *to++ = *from++;
    }
 }
```

- Note we had to pass size (`n`) to `copy`

Pointer Arithmetic: Peer Instruction Question

How many of the following are invalid?

I. pointer + integer
II. integer + pointer
III. pointer + pointer
IV. pointer – integer
V. integer – pointer
VI. pointer – pointer
VII. compare pointer to pointer
VIII. compare pointer to integer
IX. compare pointer to 0
X. compare pointer to NULL

```
#invalid
  a: 1
  b: 2
  c: 3
  d: 4
  e: 5
```

Pointer Arithmetic: Peer Instruction Answer

How many of the following are invalid?

I.	pointer + integer	ptr + 1
II.	integer + pointer	1 + ptr
III.	pointer + pointer	ptr + ptr
IV.	pointer – integer	ptr - 1
V.	integer – pointer	1 - ptr
VI.	pointer – pointer	ptr - ptr
VII.	compare pointer to pointer	ptr1 == ptr2
VIII.	compare pointer to integer	ptr == 1
IX.	compare pointer to 0	ptr == NULL
X.	compare pointer to NULL	ptr == NULL

```
#invalid
     a: 1
     b: 2
     c: 3
     d: 4
     e: 5
```


Pointer Arithmetic

- What is valid pointer arithmetic?
  - Add an integer to a pointer
  - Subtract 2 pointers (in the same array)
  - Compare pointers (<, <=, ==, !=, >, >=)
  - Compare pointer to NULL (indicates that the pointer points to nothing)
- Everything else is illegal since it makes no sense:
  - Adding two pointers
  - Multiplying pointers
  - Subtract pointer from integer

Pointers and Functions (1/4)

- Sometimes you want to have a procedure increment a variable …
- What gets printed?

```
void AddOne(int  x)              y = 5
{     x =  x + 1;    }

int y = 5;
AddOne(y);
printf("y = %d\n", y);
```


Pointers and Functions (2/4)

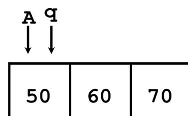
- Solved by passing in a *pointer* to our subroutine.
- Now what gets printed?

```
void AddOne(int *p)              y = 6
{     *p = *p + 1;    }

int y = 5;
AddOne(&y);
printf("y = %d\n", y);
```


Pointers and Functions (3/4)

- But what if the thing you want changed is a *pointer*?
- What gets printed?

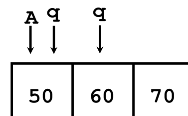
```
void IncrementPtr(int  *p)       *q = 50
{     p =  p + 1;    }

int A[3] = {50, 60, 70};
int *q = A;
IncrementPtr(q);
printf("*q = %d\n", *q);
```



Pointers and Functions (4/4)

- Solution! Pass a *pointer to a pointer*, declared as `**h`
- Now what gets printed?

```
void IncrementPtr(int **h)       *q = 60
{    *h = *h + 1;    }

int A[3] = {50, 60, 70};
int *q = A;
IncrementPtr(&q);
printf("*q = %d\n", *q);
```

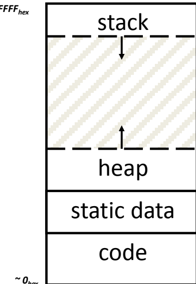


Agenda

- Pointers Review
- C Memory Management
- Administrivia
- Malloc and Free
- Break
- Common Memory Problems

C Memory Layout

- 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.



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

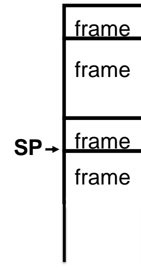
Where are Variables Allocated?

- If declared outside a procedure, allocated in "static" storage
- If declared inside procedure, allocated on the "stack" and freed when procedure returns
  - main() is treated like a procedure

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


The Stack

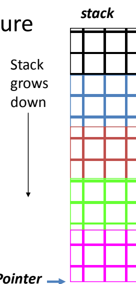
- Stack frame includes:
  - Location of who called the function
  - Parameters
  - Space for local variables
- Stack frames contiguous blocks of memory; stack pointer tells where bottom stack frame is
- When procedure ends, stack frame is tossed off the stack; frees memory for future stack frames



The 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)
        {
        }
```



What's Wrong with this Code?

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

main () {
 int *stackAddr,content;
 stackAddr = getPtr();
 content = *stackAddr;
 printf("%d", content); /* 3 */
 content = *stackAddr;
 printf("%d", content); /*13451514 */
};
```

What's Wrong with this Code?

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

main () {
 int *stackAddr,content;
 stackAddr = getPtr();
 content = *stackAddr;
 printf("%d", content); /* 3 */
 content = *stackAddr;
 printf("%d", content); /*13451514 */
};
```

main | getPtr() (y==3) | SP→

main | SP→

main | printf() (y==?) | SP→

Red: printf overwrites stack frame

Never return pointers to local variable from functions
Your compiler will warn you about this – don't ignore such warnings!

Arrays not implemented as you'd think

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

  *p = 1;
  printf("*p:%u, p:%u, &p:%u\n", *p, p, &p);
  *q = 2;
  printf("*q:%u, q:%u, &q:%u\n", *q, q, &q);
  *a = 3;
  printf("*a:%u, a:%u, &a:%u\n", *a, a, &a);
}
```

0	4	8	12	16	20	24	28	32	36	40	44	48	52	56	60 ...
			40	20	2	3				1					

p q x unnamed-malloc-space

? 24 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"

Agenda

- Pointers Review
- C Memory Management
- **Administrivia**
- Malloc and Free
- Break
- Common Memory Problems

Administrivia

- Future readings will be added soon. Definitely by tomorrow.
- Remember, Piazzza's a great resource for questions about assignments, material, or the course.
- Start HW1 if you haven't already. TA and Instructor response time increase drastically over the weekend!

Agenda

- Pointers Review
- C Memory Management
- Administrivia
- **Malloc and Free**
- Break
- Common Memory Problems

Using the Heap (1/3)

- C supports five functions for heap management: **malloc()**, calloc(), **free()**, cfree(), realloc()
- malloc(n):
  - Allocate a block of uninitialized memory
  - NOTE: Subsequent calls need not yield blocks in continuous sequence
  - n is an integer, indicating size of allocated memory block in bytes
  - sizeof determines size of given type (or var) in bytes, produces more portable code
  - Returns a pointer to that memory location; NULL return indicates no more memory
- Example:

```
int *ip;
ip = malloc(sizeof(int));

struct treeNode *tp;
tp = malloc(sizeof(struct tnode));
```

Using the Heap (2/3)

- Most often, malloc used to allocate space for an array of items.

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

  - Allocates space for n integers.
  - Can use pointer or array syntax on this, as usual

```
p[0] = 5;
p++; //VERY BAD! Don't lose the
  original address malloc returned or
  you can't free it!
```


Using the Heap (3/3)

- free(p):
  - Releases memory allocated by malloc()
  - p is pointer containing the address *originally* returned by malloc()

```
int *ip;
ip = malloc(sizeof(int));
... .. ..
free(ip);

struct treeNode *tp;
tp = malloc(sizeof(struct treeNode));
... .. ..
free(tp);
```

  - When insufficient free memory, malloc() returns NULL pointer; **Check for it!**

```
if ((ip = malloc(sizeof(int))) == NULL){
    printf("\nMemory is FULL\n");
    exit(1);
}
```

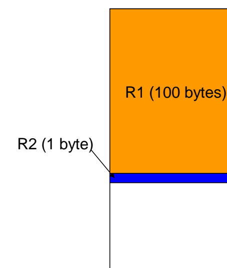
  - When you free memory, you must be sure that you pass the **original address** returned from malloc() to free(); Otherwise, system exception!

Malloc Internals

- Many calls to malloc and free with many different size blocks. Where to place them?
- Want system to be fast, minimal memory overhead.
- Want to avoid *Fragmentation*, the tendency of free space on the heap to get separated into small chunks.
- Some clever algorithms to do this, we will not cover them here.

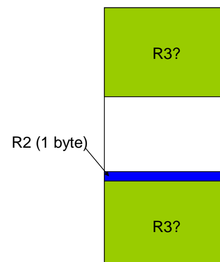
Fragmentation

- An example
  - Request R1 for 100 bytes
  - Request R2 for 1 byte
  - Memory from R1 is freed
  - Request R3 for 50 bytes
  - What if R3 was a request for 101 bytes?



Fragmentation

- **An example**
  - **Request R1 for 100 bytes**
  - **Request R2 for 1 byte**
  - **Memory from R1 is freed**
  - **Request R3 for 50 bytes**
  - **What if R3 was a request for 101 bytes?**



Agenda

- Pointers Review
- C Memory Management
- Administrivia
- Malloc and Free
- Break
- Common Memory Problems

Agenda

- Pointers Review
- C Memory Management
- Administrivia
- Malloc and Free
- Break
- Common Memory Problems

Segmentation Fault vs. Bus Error

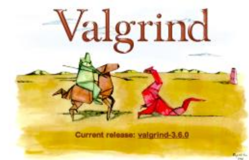
- `http://www.hyperdictionary.com/`
- Bus Error
  - A fatal failure in the execution of a machine language instruction resulting from the processor detecting an anomalous condition on its bus. Such conditions include invalid address alignment (accessing a multi-byte number at an odd address), accessing a physical address that does not correspond to any device, or some other device-specific hardware error. A bus error triggers a processor-level exception which Unix translates into a "SIGBUS" signal which, if not caught, will terminate the current process.
- Segmentation Fault
  - 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.

Common Memory Problems

- Using uninitialized values
- Using memory that you don't own
  - Deallocated stack or heap variable
  - Out of bounds reference to stack or heap array
  - Using NULL or garbage data as a pointer
- Improper use of free/realloc by messing with the pointer handle returned by malloc/calloc
- Memory leaks (you allocated something you forgot to later free)

Debugging Tools

- Runtime analysis tools for finding memory errors
  - Dynamic analysis tool: collects information on memory management while program runs
  - Contrast with static analysis tool like `lint`, which analyzes source code without compiling or executing it
  - No tool is guaranteed to find ALL memory bugs – this is a very challenging programming language research problem
- *You will be introduced to Valgrind in Lab #3!*



http://valgrind.org

Using Uninitialized Values

- What is wrong with this code?

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

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


Using Uninitialized Values

- What is wrong with this code?

```
void foo(int *pi) {
    int j;
    *pi = j;
        /* j is uninitialized, copied into *pi */
}

void bar() {
    int i=10;
    foo(&i);
    printf("i = %d\n", i);
        /* Using i, which now has junk value */
}
```

Valgrind Output (Highly Abridged!)

```
==98863== Memcheck, a memory error detector
==98863== Using Valgrind-3.6.0 and LibVEX; rerun with -h for copyright info

==98863== Conditional jump or move depends on uninitialised value(s)
==98863==    at 0x100031A1E: __vfprintf (in /usr/lib/libSystem.B.dylib)
==98863==    by 0x100073BCA: vfprintf_l (in /usr/lib/libSystem.B.dylib)
==98863==    by 0x1000A11A6: printf (in /usr/lib/libSystem.B.dylib)
==98863==    by 0x100000EEE: main (slide21.c:13)
==98863==  Uninitialised value was created by a stack allocation
==98863==    at 0x100000EB0: foo (slide21.c:3)
==98863==
```


Valgrind Output (Highly Abridged!)

```
==98863== HEAP SUMMARY:
==98863==     in use at exit: 4,184 bytes in 2 blocks
==98863==   total heap usage: 2 allocs, 0 frees, 4,184 bytes allocated
==98863==
==98863== LEAK SUMMARY:
==98863==    definitely lost: 0 bytes in 0 blocks
==98863==    indirectly lost: 0 bytes in 0 blocks
==98863==      possibly lost: 0 bytes in 0 blocks
==98863==    still reachable: 4,184 bytes in 2 blocks
==98863==         suppressed: 0 bytes in 0 blocks
==98863== Reachable blocks (those to which a pointer was found) are not shown.
==98863== To see them, rerun with: --leak-check=full --show-reachable=yes
```


Using Memory You Don't Own

- 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;
}
```


Using Memory You Don't Own

- Following a NULL pointer to mem addr 0!

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

int findLastNodeValue(Node* head) {
    while (head->next != NULL) {
    /* What if head happens to be NULL? */
        head = head->next;
    }
    return head->val; /* What if head is NULL? */
}
```


Using Memory You Don't Own

- What is wrong with this code?

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

struct Profile person = malloc(sizeof(struct 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);
```


Using Memory You Don't Own

- What is wrong with this code?

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

struct Profile person = malloc(sizeof(struct Profile));
char* name = getName();
person.name = malloc(sizeof(char)*(strlen(name)+1));
strcpy(person.name,name);
… //Do stuff (that isn't buggy)
free(person);
free(person.name);
```

strlen only counts # of characters in a string. Need to add 1 for null terminator!

Accessing memory after you've already freed it. These statements should be in the reverse order

Using Memory You Don't Own

- What is wrong with this code?

```
int* init_array(int *ptr, int new_size) {
    ptr = realloc(ptr, new_size*sizeof(int)); //Returns new block of memory
    memset(ptr, 0, new_size*sizeof(int));
    return ptr;
}

int* fill_fibonacci(int *fib, int size) {
    int i;
    init_array(fib, size);
    /* fib[0] = 0; */ fib[1] = 1;
    for (i=2; i<size; i++)
     fib[i] = fib[i-1] + fib[i-2];
    return fib;
}
```


Using Memory You Don't Own

- Improper matched usage of mem handles

```
int* init_array(int *ptr, int new_size) {
    ptr = realloc(ptr, new_size*sizeof(int));
    memset(ptr, 0, new_size*sizeof(int));
    return ptr;
}

int* fill_fibonacci(int *fib, int size) {
    int i;
    /* oops, forgot: fib = */ init_array(fib, size);
    /* fib[0] = 0; */ fib[1] = 1;
    for (i=2; i<size; i++)
     fib[i] = fib[i-1] + fib[i-2];
    return fib;
}
```

Remember: `realloc` may move entire block

What if array is moved to new location?

Using Memory You Don't Own

- What's wrong with this code?

```
char *append(const char* s1, const char *s2) {
   const int MAXSIZE = 128;
   char result[128];
   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;
}
```


Using Memory You Don't Own

- Beyond stack read/write

```
char *append(const char* s1, const char *s2) {
   const int MAXSIZE = 128;
   char result[128];
   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;
}
```

`result` is a local array name – stack memory allocated

Function returns pointer to stack memory – won't be valid after function returns

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);
}
```


Using Memory You Haven't Allocated

- Reference beyond array bounds

```
void StringManipulate() {
  const char *name = "Safety Critical";
  char *str = malloc(10);
  strncpy(str, name, 10);
  str[10] = '\0';
  /* Write Beyond Array Bounds */
  printf("%s\n", str);
  /* Read Beyond Array Bounds */
}
```

Faulty Heap Management

- What is wrong with this code?

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

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


Faulty Heap Management

- Memory leak: *more mallocs than frees*

```
int *pi;
void foo() {
    pi = malloc(8*sizeof(int));
    /* Allocate memory for pi */
    /* Oops, leaked the old memory pointed to by pi */
    …
    free(pi); /* foo() is done with pi, so free it */
}

void main() {
    pi = malloc(4*sizeof(int));
    foo(); /* Memory leak: foo leaks it */
}
```


Faulty Heap Management

- What is wrong with this code?

```
int *plk = NULL;
void genPLK() {
    plk = malloc(2 * sizeof(int));
  … … …
    plk++;
}
```


Faulty Heap Management

- Potential memory leak – handle has been changed, do you still have copy of it that can correctly be used in a later free?

```
int *plk = NULL;
void genPLK() {
    plk = malloc(2 * sizeof(int));
    … … …
    plk++;    /* Potential leak: pointer variable incremented past
    beginning of block! */
}
```


Faulty Heap Management

- What is wrong with this code?

```
void FreeMemX() {
    int fnh = 0;
    free(&fnh);
}

void FreeMemY() {
    int *fum = malloc(4 * sizeof(int));
    free(fum+1);
    free(fum);
    free(fum);
}
```


Faulty Heap Management

- Can't free non-heap memory; Can't free memory that hasn't been allocated

```
void FreeMemX() {
    int fnh = 0;
    free(&fnh); /* Oops! freeing stack memory */
}

void FreeMemY() {
    int *fum = malloc(4 * sizeof(int));
    free(fum+1);
    /* fum+1 is not a proper handle; points to middle
    of a block */
    free(fum);
    free(fum);
    /* Oops! Attempt to free already freed memory */
}
```

In Conclusion..

- C has three pools of data memory (+ code memory)
  - Static storage: global variable storage, basically permanent over entire program run
  - The Stack: local variable storage, parameters, return address
  - *The Heap (dynamic storage): malloc() allocates space from here, free() returns it.*
- Common (Dynamic) Memory Problems
  - Using uninitialized values
  - Accessing memory beyond your allocated region
  - Improper use of free/realloc by messing with the pointer handle returned by malloc/calloc
  - Memory leaks: mismatched malloc/free pairs

Reference slides

You ARE responsible for the material on these slides (they're just taken from the reading anyway) ; we've moved them to the end and off-stage to give more breathing room to lecture!

Using the Heap

- calloc(n, size):
  - Allocate n elements of same data type; n can be an integer variable, use calloc()to allocate a dynamically size array
  - n is the # of array elements to be allocated
  - size is the number of bytes of each element
  - *calloc() guarantees that the memory contents are initialized to zero*

  E.g.: allocate an array of 10 elements

```
int *ip;
ip = calloc(10, sizeof(int));
```

  *(ip+1) refers to the 2nd element, like ip[1]

  *(ip+i) refers to the i+1th element, like ip[i]

  Beware of referencing beyond the allocated block: e.g., *(ip+10)
  - calloc() returns NULL if no further memory is available
- cfree(p):
  - cfree() releases the memory allocated by calloc(); E.g.: cfree(ip);

Using the Heap

- realloc(p,size):
  - Resize a previously allocated block at p to a new size
  - If p is NULL, then realloc behaves like malloc
  - If size is 0, then realloc behaves like free, deallocating the block from the heap
  - Returns new address of the memory block; NOTE: it is likely to have moved!

  E.g.: allocate an array of 10 elements, expand to 20 elements later

```
int *ip;
ip = malloc(10*sizeof(int));
/* always check for ip == NULL */
… … …
ip = realloc(ip,20*sizeof(int));
/* always check for ip == NULL */
/* contents of first 10 elements retained */
… … …
realloc(ip,0); /* identical to free(ip) */
```

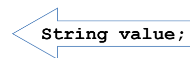

Linked List Example

- 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

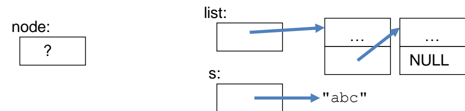
```
typedef NodeStruct *List;
typedef char *String;
```

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, just like (cons s1 (cons s2 nil)) */
   theList = cons(s1, cons(s2, NULL));
```

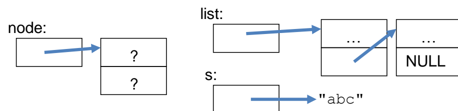
Linked List Example

```
/* 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;
}
```



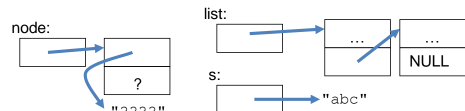
Linked List Example

```
/* 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;
}
```



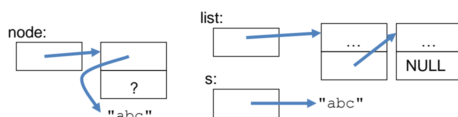
Linked List Example

```
/* 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;
}
```



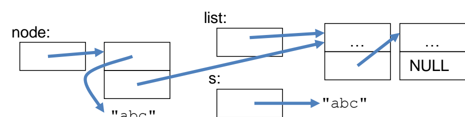
Linked List Example

```
/* 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;
}
```