CS61C : Machine Structures

Lecture 5 – Introduction to C (pt 3)
C Memory Management

2007-01-26

There is **one** handout today at the front and back of the room!

Lecturer SOE Dan Garcia

`www.cs.berkeley.edu/~ddgarcia`

Norway: iTunes illegal! ⇒
Norway ruled that iTunes was illegal because it did not allow downloaded songs encoded with their proprietary Fairplay system to be played on non-iPods. They are asking Apple to open their system up by Oct 1.

`www.msnbc.msn.com/id/16793043/`

Review

- **Pointers and arrays are virtually same**

- **C knows how to increment pointers**

- **C is an efficient language, with little protection**
  - **Array bounds not checked**
  - **Variables not automatically initialized**

- **(Beware) The cost of efficiency is more overhead for the programmer.**
  - **"C gives you a lot of extra rope but be careful not to hang yourself with it!"**

C Strings

- **A string in C is just an array of characters.**

```
char string[] = "abc";
```

- **How do you tell how long a string is?**

  - **Last character is followed by a 0 byte (null terminator)**

```
int strlen(char s[])
{
    int n = 0;
    while (s[n] != 0) n++;
    return n;
}
```

- **Sometimes you want to have a procedure increment a variable?**

- **What gets printed?**

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

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

y = 5

- **Solved by passing in a pointer to our subroutine.**

- **Now what gets printed?**

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

int y = 5;
AddOne(&y);
printf("y = %d\n", y);
```
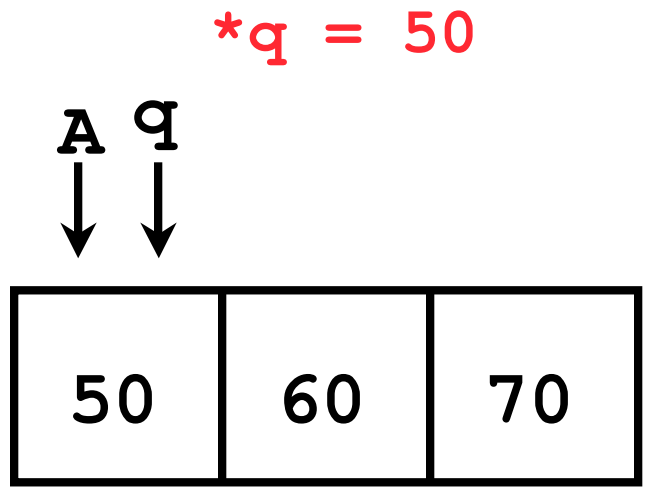
y = 6

Pointers (3/4)

- **But what if what you want changed is a pointer?**

- **What gets printed?**

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

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

*q = 50

A q

50	60	70

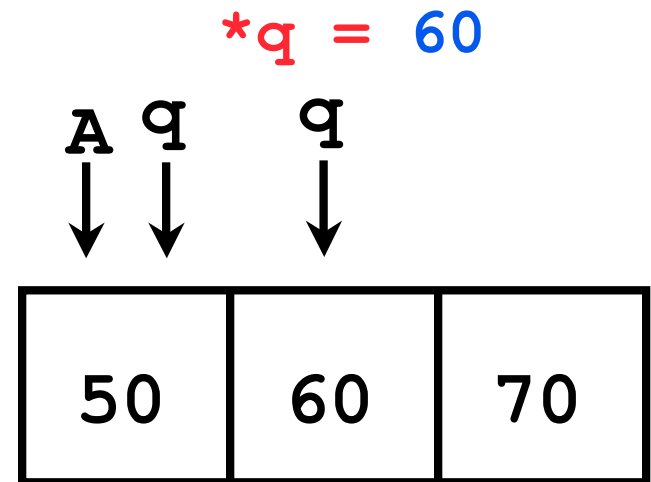# Pointers (4/4)

- **Solution! Pass a pointer to a pointer, declared as `**h`**

- **Now what gets printed?**

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

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

*q = 60

A q   q

50	60	70

Dynamic Memory Allocation (1/4)

- **C has operator `sizeof()` which gives size in bytes (of type or variable)**

- **Assume size of objects can be misleading and is bad style, so use `sizeof(type)`**
  - **Many years ago an `int` was 16 bits, and programs were written with this assumption.**
  - **What is the size of integers now?**

- **"`sizeof`" knows the size of arrays:**

  ```
  int ar[3]; // Or:   int ar[] = {54, 47, 99}
  sizeof(ar) ⟹ 12
  ```

  - **…as well for arrays whose size is determined at run-time:**

  ```
  int n = 3;
  int ar[n]; // Or: int ar[fun_that_returns_3()];
  sizeof(ar) ⟹ 12
  ```

Dynamic Memory Allocation (2/4)

- **To allocate room for something new to point to, use `malloc()` (with the help of a typecast and `sizeof`):**

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

- Now, `ptr` points to a space somewhere in memory of size `(sizeof(int))` in bytes.

- `(int *)` simply tells the compiler what will go into that space (**called a typecast**).

- **`malloc` is almost never used for 1 var**

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

- **This allocates an array of `n` integers.**

Dynamic Memory Allocation (3/4)

- **Once `malloc()` is called, the memory location contains garbage, so don't use it until you've set its value.**

- **After dynamically allocating space, we must dynamically free it:**

  `free(ptr);`

- **Use this command to clean up.**

  - **Even though the program `free`s all memory on `exit` (or when `main` returns), don't be lazy!**

  - **You never know when your `main` will get transformed into a subroutine!**
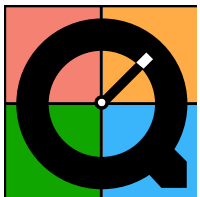
Dynamic Memory Allocation (4/4)

- **The following two things will cause your program to crash or behave strangely later on, and cause VERY VERY hard to figure out bugs:**

    - `free()`ing the same piece of memory twice

    - calling `free()` on something you didn't get back from `malloc()`

- **The runtime <u>does not</u> check for these mistakes**

    - Memory allocation is so performance-critical that there just isn't time to do this

    - The usual result is that you corrupt the memory allocator's internal structure

    - You won't find out until much later on, in a totally unrelated part of your code!

Binky Pointer Video (thanks to NP @ SU)
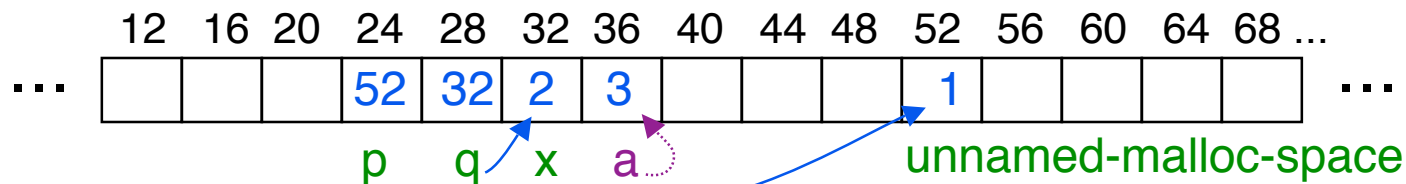
Arrays not implemented as you'd think

```
void foo() {
  int *p, *q, x, a[1]; // a[] = {3} also works here
  p = (int *) malloc (sizeof(int));
  q = &x;

  *p = 1; // p[0] would also work here
  *q = 2; // q[0] would also work here
  *a = 3; // a[0] would also work here

  printf("*p:%u, p:%u, &p:%u\n", *p, p, &p);
  printf("*q:%u, q:%u, &q:%u\n", *q, q, &q);
  printf("*a:%u, a:%u, &a:%u\n", *a, a, &a);

}
```



```
*p:1, p:52, &p:24
*q:2, q:32, &q:28
*a:3, a:36, &a:36
```

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)
{   As always in C, the argument is passed by "value" — a copy is made.
    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 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);
```

Kilo, Mega, Giga, Tera, Peta, Exa, Zetta, Yotta

1. Kind Meek Giggles Tease Peering Excited Zealous Youngsters. – Yiding J
2. Kissing me gives tears per extra zebra YO! – Peter D
3. Kiss me, gimme tea, persistently extol zee. You! – Hava E
4. Kia Mechanics (are) Giant Terrible People Exclaiming Zealous Yodels. – Gary M
5. Kiss me, gimme tea, pet exaltingly, zestful you. – Hava E
6. Kid meets giant Texas people exercising zen-like yoga. -Rolf O
7. Kicking methods gives teaching people extra zest, youbetcha! – Peter D
8. Kind men give ten percent extra, zestfully, youthfully. – Hava E
9. Kissing Mentors Gives Testy Persistent Extremists Zealous Youthfulness. – Gary M
10. Kindness means giving, teaching, permeating excess zeal yourself. – Hava E
11. Kissing me gives ten percent extra zeal & youth! – Dan (taking ideas from all)
    1. Killing messengers gives terrible people exactly zero, yo
    2. Kindergarten means giving teachers perfect examples (of) zeal (&) youth
    3. Kissing mediocre girls/guys teaches people (to) expect zero (from) you
    4. Kinky Mean Girls Teach Penis-Extending Zen Yoga
    5. Kissing Mel Gibson, Teddy Pendergrass exclaimed: "Zesty, yo!" – Dan Garcia

Pointer Arithmetic Peer Instruction Q

How many of the following are invalid?

		#invalid
I.	pointer + integer	1
II.	integer + pointer	2
III.	pointer + pointer	3
IV.	pointer – integer	4
V.	integer – pointer	5
VI.	pointer – pointer	6
VII.	compare pointer to pointer	7
VIII.	compare pointer to integer	8
IX.	compare pointer to 0	9
X.	compare pointer to `NULL`	(1) 0

Peer Instruction

Which are guaranteed to print out 5?

```
I:   main() {
       int *a-ptr; *a-ptr = 5; printf("%d", *a-ptr); }

II:  main() {
       int *p, a = 5;
       p = &a; ...
       /* code; a & p NEVER on LHS of = */
       printf("%d", a); }

III: main() {
       int *ptr;
       ptr = (int *) malloc (sizeof(int));
       *ptr = 5;
       printf("%d", *ptr); }
```
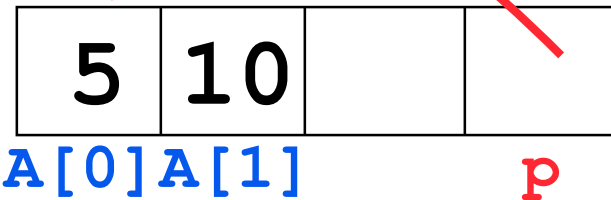
	I	II	III
0:	–	–	–
1:	–	–	YES
2:	–	YES	–
3:	–	YES	YES
4:	YES	–	–
5:	YES	–	YES
6:	YES	YES	–
7:	YES	YES	YES

Peer Instruction

```
int main(void){
  int A[] = {5,10};
  int *p = A;

  printf("%u %d %d %d\n",p,*p,A[0],A[1]);
   p =  p + 1;
  printf("%u %d %d %d\n",p,*p,A[0],A[1]);
  *p = *p + 1;
  printf("%u %d %d %d\n",p,*p,A[0],A[1]);
}
```

5	10		
A[0]	A[1]		p

If the first `printf` outputs <u>100 5 5 10</u>, what will the other two `printf` output?

```
1: 101 10 5 10         then 101 11 5 11
2: 104 10 5 10         then 104 11 5 11
3: 101 <other> 5 10  then 101 <3-others>
4: 104 <other> 5 10  then 104 <3-others>
5: One of the two printfs causes an ERROR
6: I surrender!
```

"And in Conclusion…"

- **Use handles to change pointers**

- **Create abstractions with structures**

- **Dynamically allocated heap memory must be manually deallocated in C.**
  - **Use `malloc()` and `free()` to allocate and deallocate memory from heap.**

Bonus slides

- **These are extra slides that used to be included in lecture notes, but have been moved to this, the "bonus" area to serve as a supplement.**

- **The slides will appear in the order they would have in the normal presentation**

Bonus

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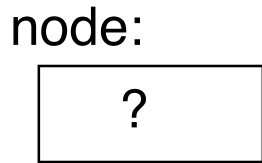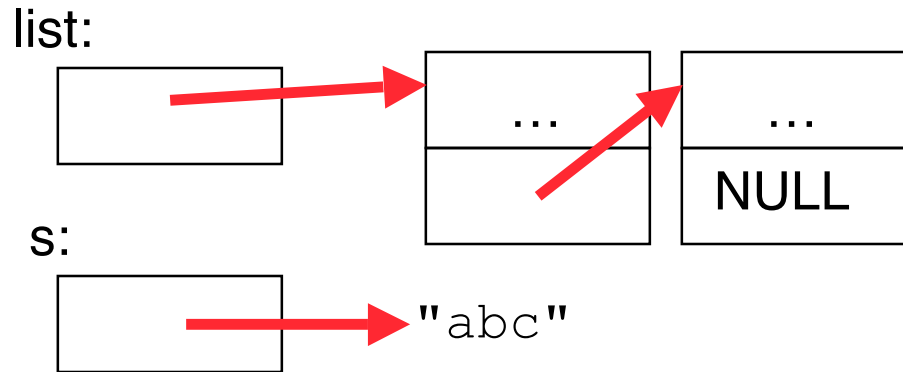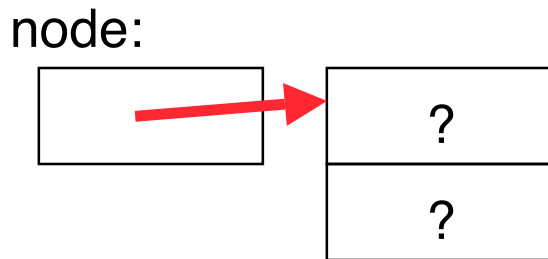
Linked List Example

```
/* 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;
}
```


node:

```
  ?
```

list:

```
...
...
NULL
```

s:

"abc"
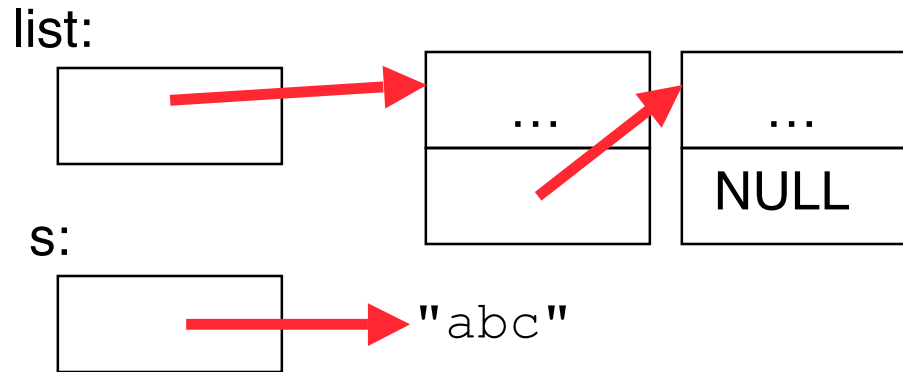
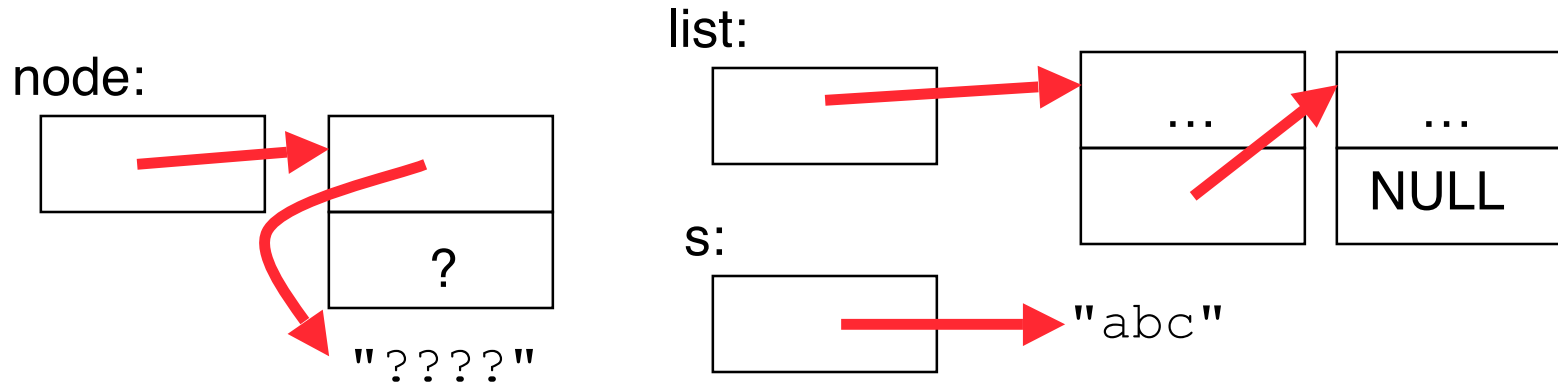# Linked List Example

```
/* 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;
}
```
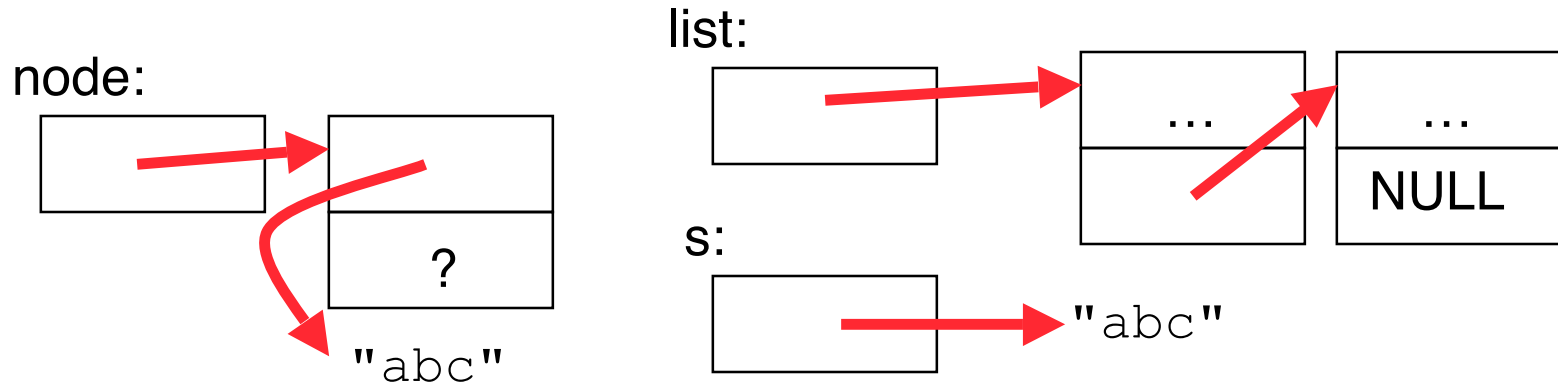
node:

"????"

list:

... | ... | NULL

s:
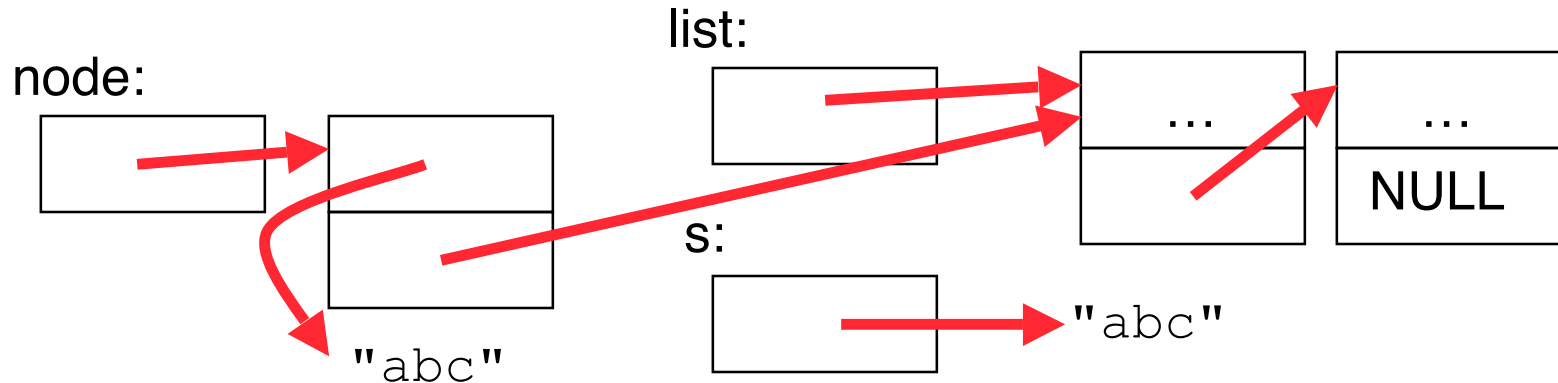
"abc"

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;
}
```

node:
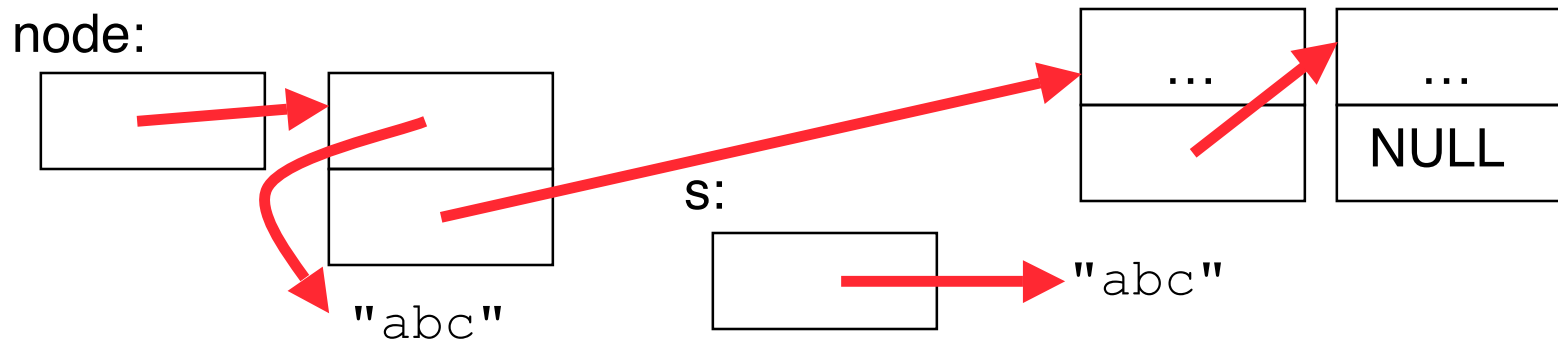
list:

... NULL

s:

? "abc"

"abc"

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;
}
```

node:

list:

s:

... NULL

"abc"

"abc"

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;
}
```

Pointer Arithmetic Summary

- `x = *(p+1)` **?**
  - ⟹ `x = *(p+1) ;`

- `x = *p+1` **?**
  - ⟹ `x = (*p) + 1 ;`

- `x = (*p)++` **?**
  - ⟹ `x = *p ; *p = *p + 1;`

- `x = *p++` **?** `(*p++)` **?** `*(p)++` **?** `*(p++)` **?**
  - ⟹ `x = *p ; p =  p + 1;`

- `x = *++p` **?**
  - ⟹ `p = p + 1 ; x = *p ;`

- Lesson?
  - These cause more problems than they solve!

Kilo, Mega, Giga, Tera, Peta, Exa, Zetta, Yotta

- Kim's melodious giddiness terrifies people, excepting zealous yodelers

- Kirby Messed Gigglypuff Terribly, (then) Perfectly Exterminated Zelda and Yoshi

- Killed meat gives teeth peace except zebra yogurt

- Kind Men Give Tense People Extra Zeal (for) Yoga

- Killing melee gives terror; peace exhibits Zen yoga

- Young Zebras Exclaim, "People Teach {Giraffes, Girls} Messy Kissing!" – Omar Akkawi

- "King me," Gina tells Perry, expert zebra yodeler – Diana Ko

- Kirk met Gibson's team, perilously expecting zealous youngsters – Diana Ko

- Kind Men Give Ten Percent Expressly Zee Yoorphans – Daniel Gallagher

- King Mel Gibson Tells People "Examine Ze Yoodle!" – Daniel Gallagher

- Kizzle Meh Gizzle The Pezzle Exizzle Zeh Yo! – Daniel Gallagher

- Killer Mechanical { Giraffe / Giant } Teaches Pet, Extinct Zebra, to Yodel – Larry Ly

- Kilted Men Given Testosterone Perform Exceedingly Zealous Yoga –David Wu