inst.eecs.berkeley.edu/~cs61c

CS61C : Machine Structures

Lecture 5 – C Memory Management

2004-09-10

Lecturer PSOE Dan Garcia

www.cs.berkeley.edu/~ddgarcia

Barry Bonds nears 700! ⇒ We are witness to perhaps the greatest baseball player of all time, and he plays weekly 10 miles from here! Years from now you'll know where you were when he passed 755.

More from Wednesday's lecture

1. `#define` macros may go anywhere. Thereafter the name is replaced with the replacement text. It is usually good style to put all `#define`s at the top so that reordering code doesn't cause bugs.

2. `void *` pointers used to be `char *` pointers (before ANSI C). Therefore, partially to maintain compatibility, `++` incrementing a `void *` pointer via increments it by 1 byte.

3. `const` type qualifier announces objects are not to be changed. Implementation-dependent storage and violation penalty.

C String Standard Functions

- `int strlen(char *string);`
  - compute the length of `string`
- `int strcmp(char *str1, char *str2);`
  - return 0 if `str1` and `str2` are identical (how is this different from `str1 == str2`?)
- `int strcpy(char *dst, char *src);`
  - copy the contents of string `src` to the memory at `dst`. The caller must ensure that `dst` has enough memory to hold the data to be copied.

Pointers to pointers (1/4) ...review...

- 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

Pointers to pointers (2/4) ...review...

- 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);
```

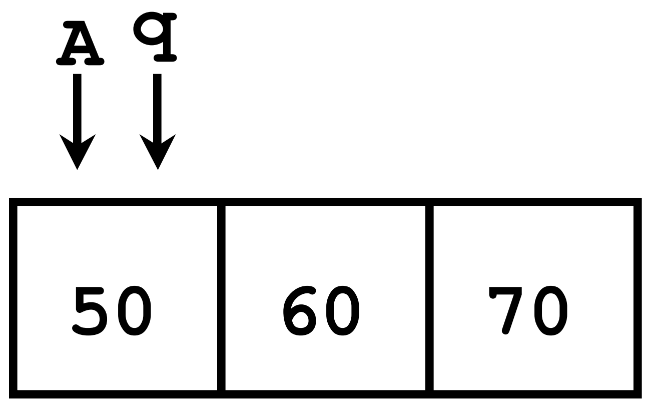
Pointers to pointers (3/4)

- 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



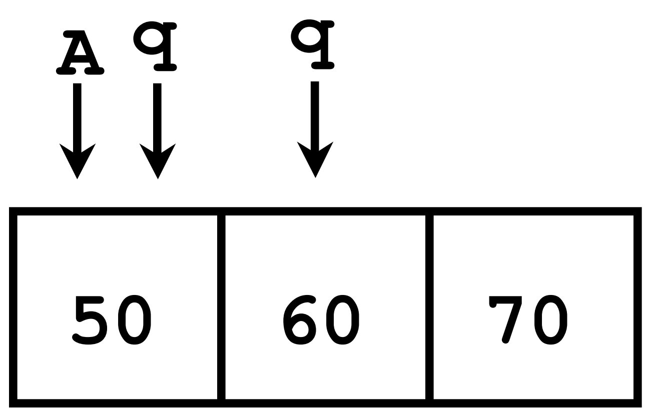
Pointers to pointers (4/4)

- **Solution! Pass a pointer to a pointer, called a handle, 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`



Administrivia

- One extra credit lab checkoff pt!
  - Sign up to get your lab checked off by the first hour and you will get 1 bonus checkoff point to count toward final grade. (Not 1/300, +1 out of 4 for that lab)

Dynamic Memory Allocation (1/3)

- C has operator `sizeof()` which gives size in bytes (of type or variable)
- Assume size of objects can be misleading & is bad style, so use `sizeof(type)`
  - Many years ago an `int` was 16 bits, and programs assumed it was 2 bytes

Dynamic Memory Allocation (2/3)

- 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.

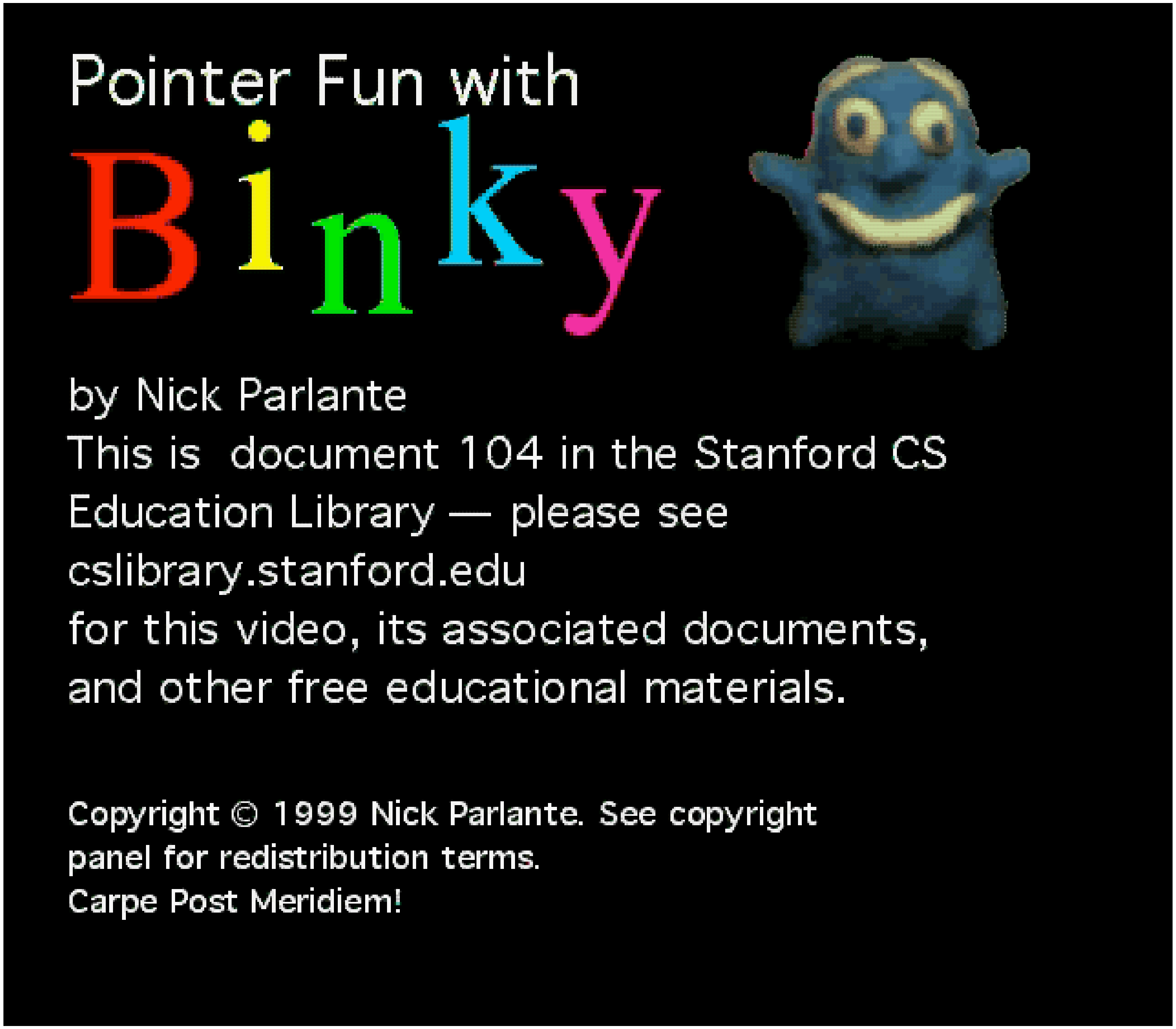
Dynamic Memory Allocation (3/3)

- 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.

Binky Pointer Video (thanks to NP @ SU)



C structures : Overview

- A `struct` is a data structure composed for simpler data types.
  - Like a class in Java/C++ but without methods or inheritance.

```
struct point {
    int x;
    int y;
}
void PrintPoint(point p)
{
    printf("(%d,%d)", p.x, p.y);
}
```

C structures: Pointers to them

- The C arrow operator (`->`) dereferences and extracts a structure field with a single operator.
- The following are equivalent:

```
struct point *p;

printf("x is %d\n", (*p).x);
printf("x is %d\n", p->x);
```

How big are structs?

- Recall C operator `sizeof()` which gives size in bytes (of type or variable)
- How big is `sizeof(p)`?

```
struct p {
    char x;
    int y;
};
```

  - 5 bytes? 8 bytes?
  - Compiler may word align integer y

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
1:	-	-	-
2:	-	-	YES
3:	-	YES	-
4:	-	YES	YES
5:	YES	-	-
6:	YES	-	YES
7:	YES	YES	-
8:	YES	YES	YES

Linked List Example

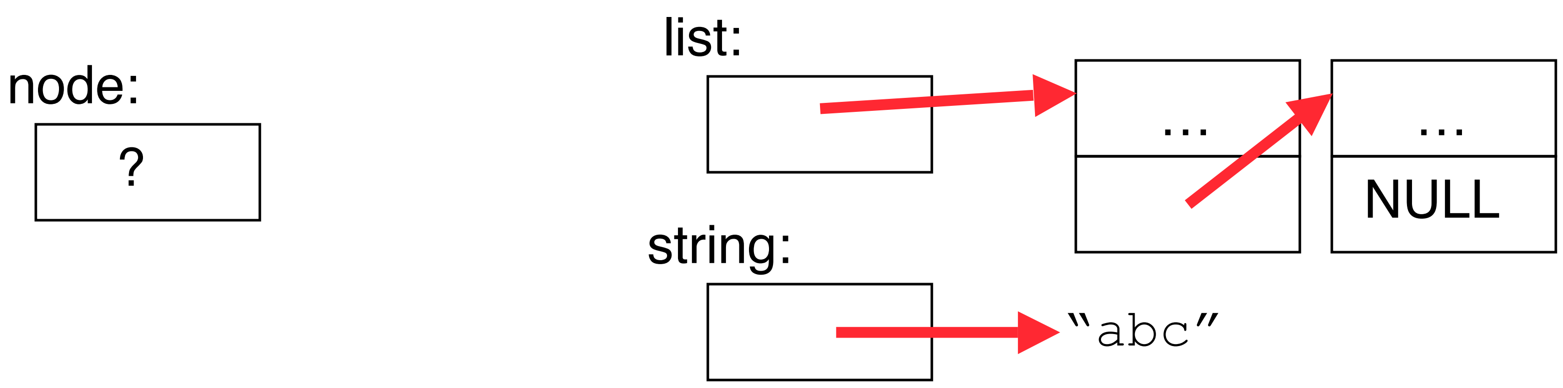
- Let's look at an example of using structures, pointers, `malloc()`, and `free()` to implement a linked list of strings.

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

/* Create a new (empty) list */
List ListNew(void)
{ return NULL; }
```

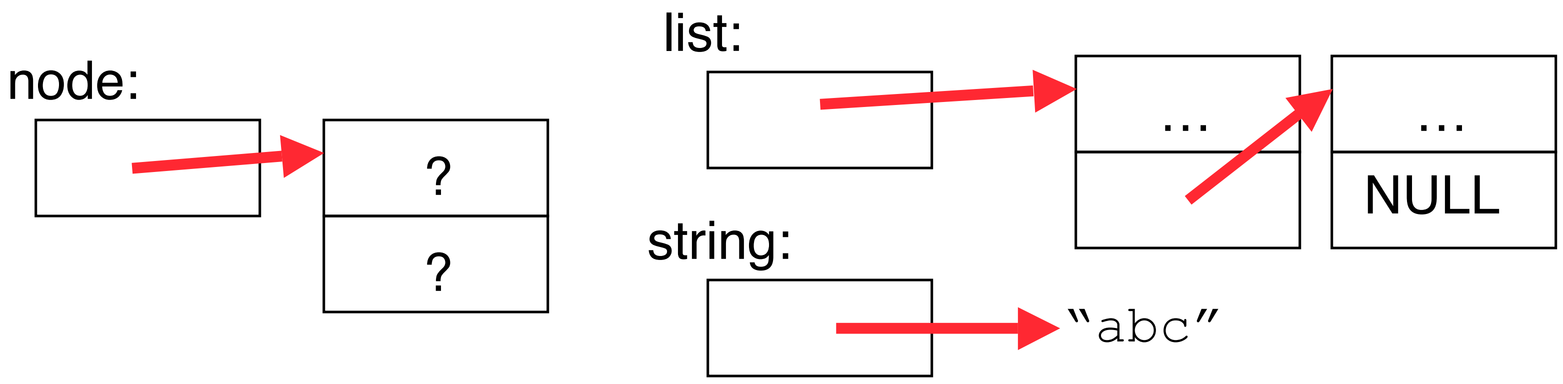
Linked List Example

```
/* add a string to an existing list */
List list_add(List list, char *string)
{
  struct Node *node =
    (struct Node*) malloc(sizeof(struct Node));
  node->value =
    (char*) malloc(strlen(string) + 1);
  strcpy(node->value, string);
  node->next = list;
  return node;
}
```



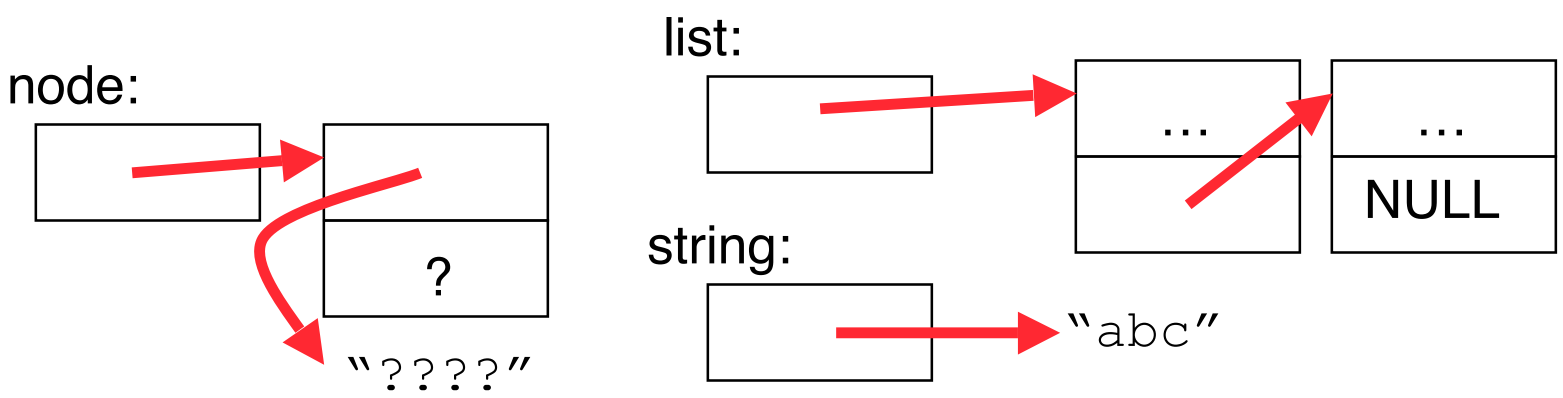
Linked List Example

```
/* add a string to an existing list */
List list_add(List list, char *string)
{
  struct Node *node =
     (struct Node*) malloc(sizeof(struct Node));
  node->value =
     (char*) malloc(strlen(string) + 1);
  strcpy(node->value, string);
  node->next = list;
  return node;
}
```



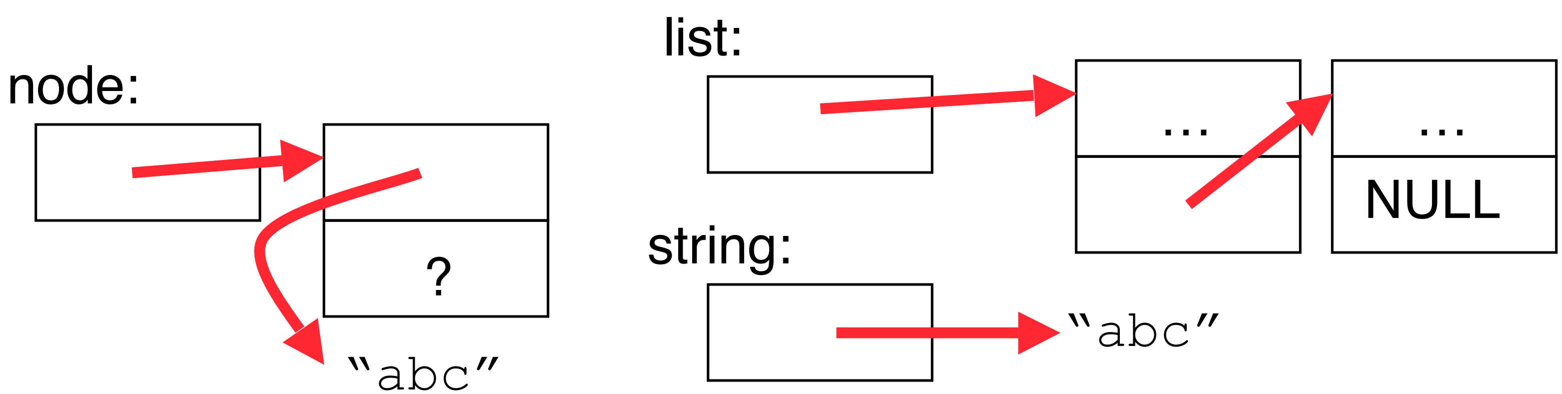
Linked List Example

```
/* add a string to an existing list */
List list_add(List list, char *string)
{
  struct Node *node =
    (struct Node*) malloc(sizeof(struct Node));
  node->value =
    (char*) malloc(strlen(string) + 1);
  strcpy(node->value, string);
  node->next = list;
  return node;
}
```



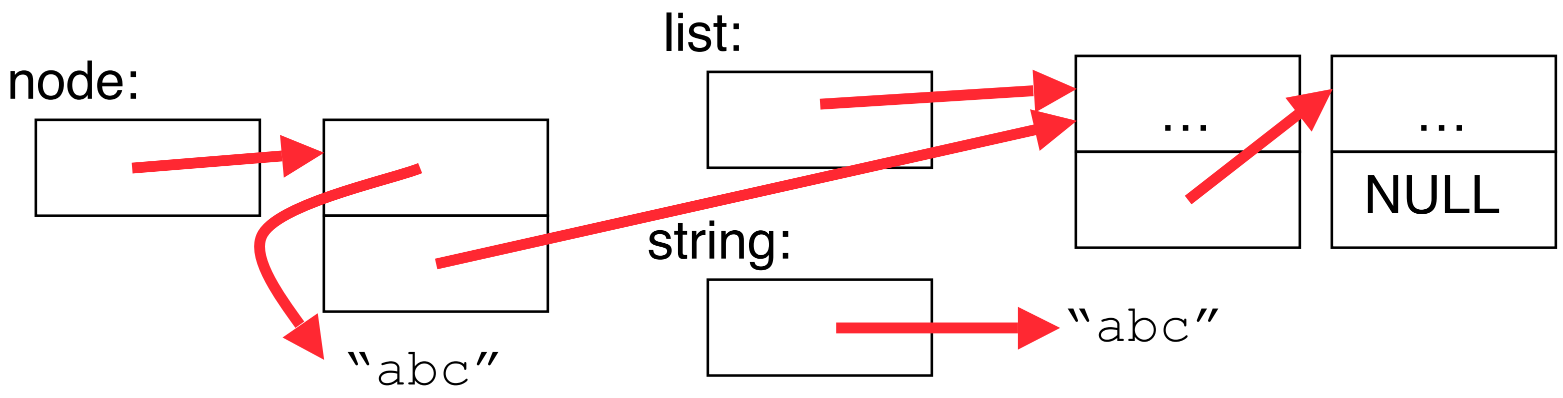
Linked List Example

```
/* add a string to an existing list */
List list_add(List list, char *string)
{
  struct Node *node =
    (struct Node*) malloc(sizeof(struct Node));
  node->value =
    (char*) malloc(strlen(string) + 1);
  strcpy(node->value, string);
  node->next = list;
  return node;
}
```



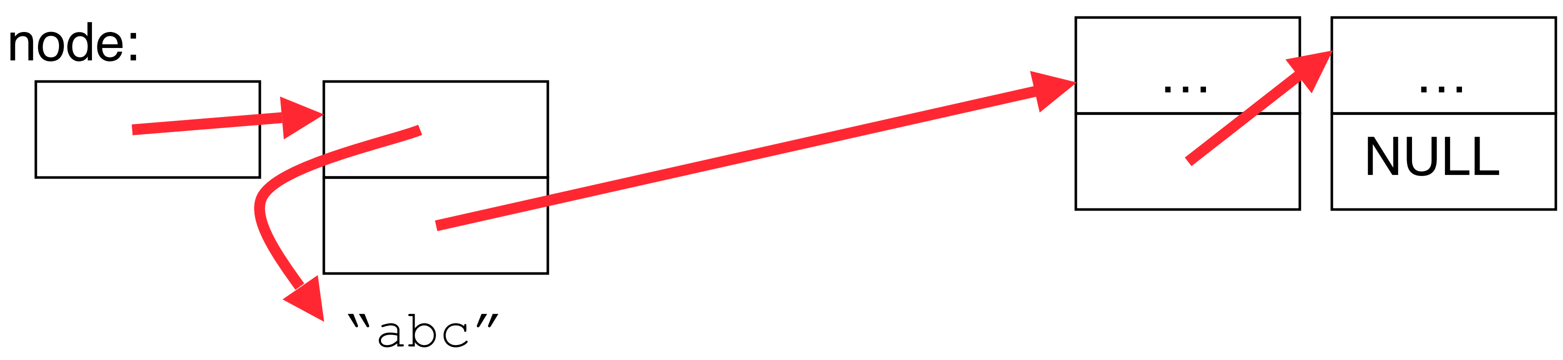
Linked List Example

```
/* add a string to an existing list */
List list_add(List list, char *string)
{
  struct Node *node =
    (struct Node*) malloc(sizeof(struct Node));
  node->value =
    (char*) malloc(strlen(string) + 1);
  strcpy(node->value, string);
  node->next = list;
  return node;
}
```



Linked List Example

```
/* add a string to an existing list */
List list_add(List list, char *string)
{
  struct Node *node =
    (struct Node*) malloc(sizeof(struct Node));
  node->value =
    (char*) malloc(strlen(string) + 1);
  strcpy(node->value, string);
  node->next = list;
  return node;
}
```



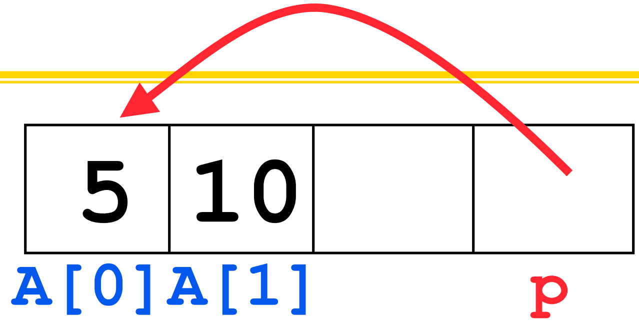
"And in Conclusion…"

- 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.

Peer Instruction

```c
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]);
}
```



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!
```