Topic 4: C Data Structures

CSE 30: Computer Organization and Systems Programming
Winter 2011

Prof. Ryan Kastner
Dept. of Computer Science and Engineering
University of California, San Diego

Arrays

- Declaration:

```
int ar[2];
```

declares a 2-element integer array.

```
int ar[] = {795, 635};
```

declares <u>and fills</u> a 2-element integer array.

- Accessing elements:

```
ar[num];
```

returns the `num`th element.

Arrays

- Arrays are (almost) identical to pointers
  - `char *string` and `char string[]` are nearly identical declarations
  - They differ in very subtle ways: incrementing, declaration of filled arrays
- Key Concept: An array variable is a pointer to the first element.

Arrays

- Consequences:
  - `ar` is a pointer
  - `ar[0]` is the same as `*ar`
  - `ar[2]` is the same as `*(ar+2)`
  - We can use pointer arithmetic to access arrays more conveniently.
- Declared arrays are only allocated while the scope is valid

```
char *foo() {
  char string[32]; ...;
  return string;}
```

is incorrect

Arrays

- Array size `n`; want to access from `0` to `n-1`, but test for exit by comparing to address one element past the array

```
int a[10], *p, *q, sum = 0;
...
p = &a[0]; q = &a[10];
while (p != q)
   /* sum = sum + p*; p = p + 1; */
   sum += *p++;
```

  - Is this legal?

- C defines that one element past end of array must be a valid address, i.e., not cause an bus error or address error

Arrays

- Array size `n`; want to access from `0` to `n-1`, so you should use counter AND utilize a constant for declaration & incr
  - Wrong
    ```
    int i, a[10];
    for(i = 0; i < 10; i++){ ... }
    ```
  - Right
    ```
    #define ARRAY_SIZE 10
    int i, a[ARRAY_SIZE];
    for(i = 0; i < ARRAY_SIZE; i++){ ... }
    ```
- Why? SINGLE SOURCE OF TRUTH
  - You're utilizing indirection and avoiding maintaining two copies of the number 10

Arrays

- Pitfall: An array in C does not know its own length, & bounds not checked!
  - Consequence: We can accidentally access off the end of an array.
  - Consequence: We must pass the array and its size to a procedure which is going to traverse it.
- Segmentation faults and bus errors:
  - These are VERY difficult to find, so be careful.

Pointer Arithmetic

- Since a pointer is just a memory address, we can add to it to traverse an array.
- `ptr+1` will return a pointer to the next array element.
- `*ptr+1` vs. `*ptr++` vs. `*(ptr+1)` ?
- What if we have an array of large structs (objects)?
  - C takes care of it: In reality, `ptr+1` doesn’t add `1` to the memory address, but rather adds the size of the array element.

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?**
  - Using anything but the standard `*p++` , `(*p)++` causes more problems than it solves!¯

Pointer Arithmetic

- C knows the size of the thing a pointer points to – every addition or subtraction moves that many bytes.
  - 1 byte for a char, 4 bytes for an int, etc.
- So the following are equivalent:

```
int get(int array[], int n)
{
    return  (array[n]);
    // OR...
    return *(array + n);
}
```

Pointer Arithmetic Question

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 (ptr == ptr)
VIII. compare pointer to integer (1 == ptr)
IX. compare pointer to 0 (ptr == NULL)
X. compare pointer to NULL (ptr == NULL)

```
#invalid
      1
      2
      3
      4
      5
      6
      7
      8
      9
   (1)0
```

Pointer Arithmetic 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 (ptr == ptr)
VIII. compare pointer to integer (1 == ptr)
IX. compare pointer to 0 (ptr == NULL)
X. compare pointer to NULL (ptr == NULL)

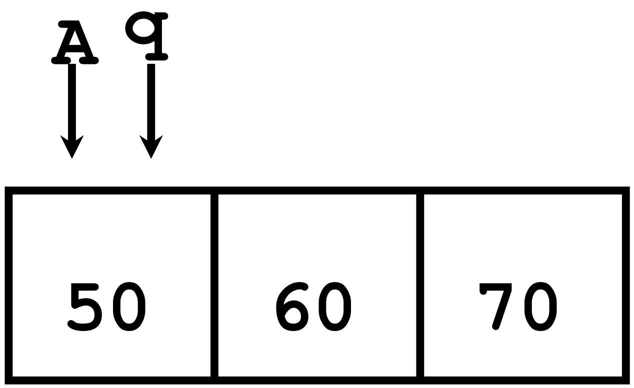
`#invalid`
1
2
3
4
5
6
7
8
9
(1) 0

Pointers to Pointers

- 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);
```



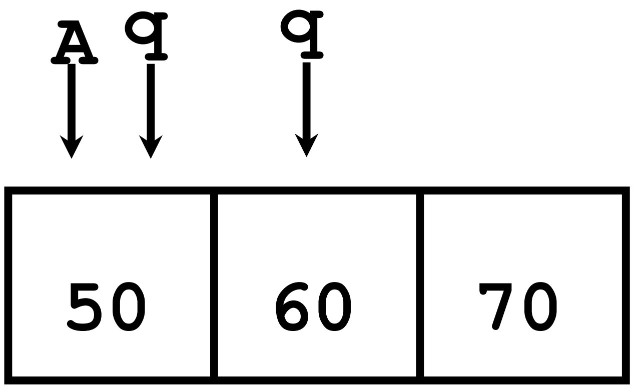
Pointers to Pointers

- 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);
```

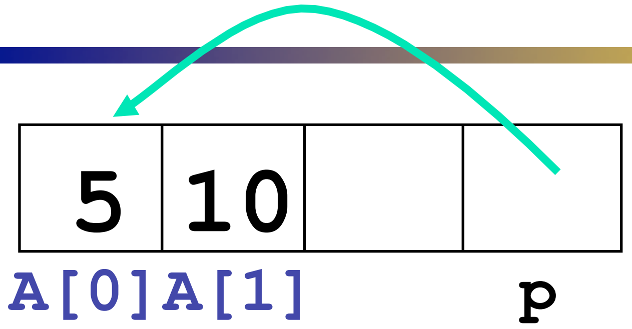
`*q = 60`



Pointers in C

- Why use pointers?
  - If we want to pass a huge struct or array, it's easier to pass a pointer than the whole thing.
  - In general, pointers allow cleaner, more compact code.
- So what are the drawbacks?
  - Pointers are probably the single largest source of bugs in software, so be careful anytime you deal with them.
  - Dangling reference (premature free)
  - Memory leaks (tardy free)

Array Question



```
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 `100 5 5 10`, 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!
```

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;
}
```

C Strings Headaches

- One common mistake is to forget to allocate an extra byte for the null terminator.
- More generally, C requires the programmer to manage memory manually (unlike Java or C ++).
  - When creating a long string by concatenating several smaller strings, the programmer must insure there is enough space to store the full string!
  - What if you don' t know ahead of time how big your string will be?

Copying strings

- Why not say

```
void copy (char sTo[ ], char sFrom[ ]) {
  sTo = sFrom;
}
```

- We need to make sure that space has been allocated for the destination string
- Similarly, you probably don’t want to compare two strings using ==

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.
- Defined in the header file string.h

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

Linked List Example

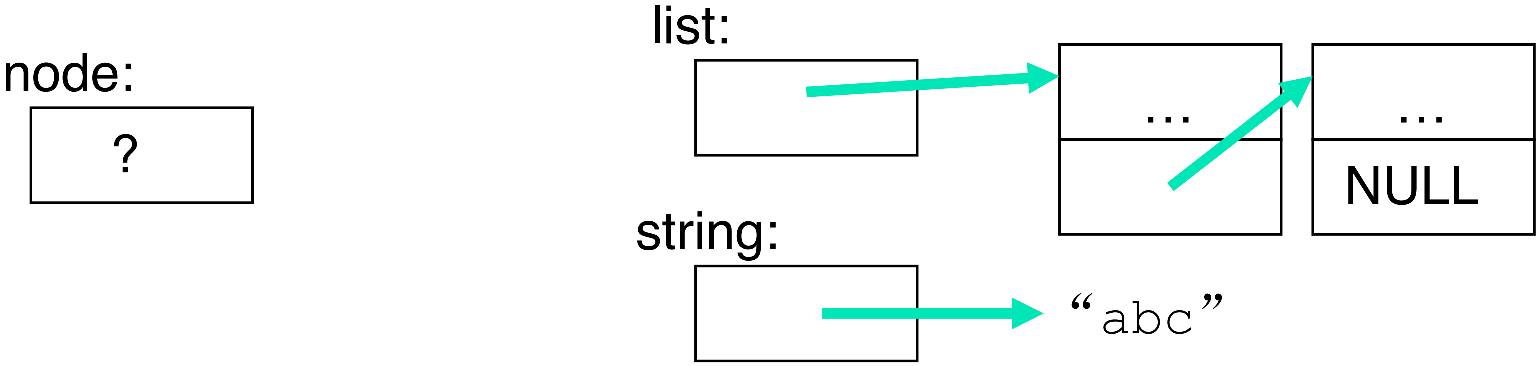
- Let' s look at an example of using structures, pointers, `malloc()`, and `free()` to implement a linked list of strings.

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

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

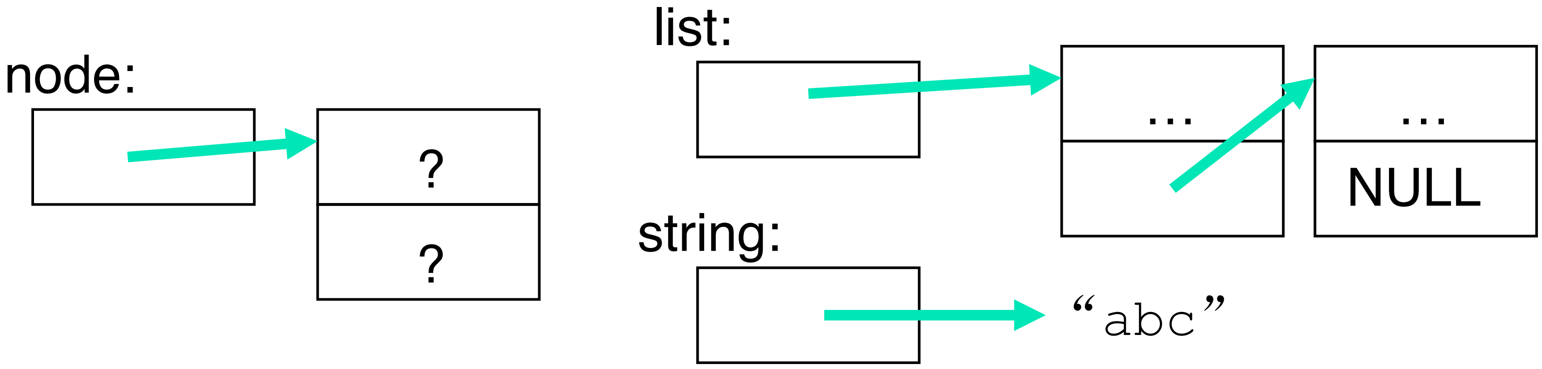
Linked List Example

```
/* 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;
}
```



node:

?

?

list:

…

…

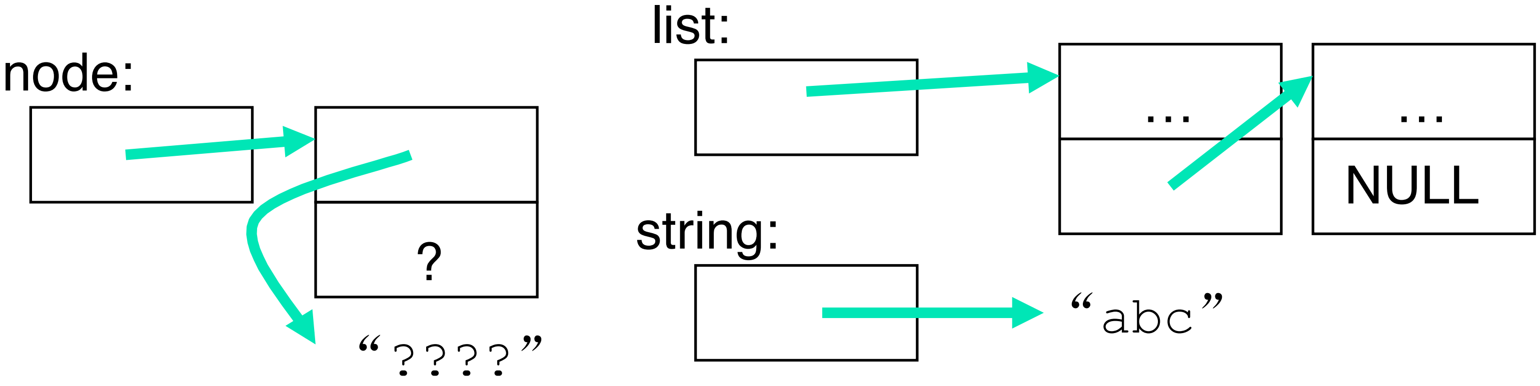
NULL

string:

“abc”

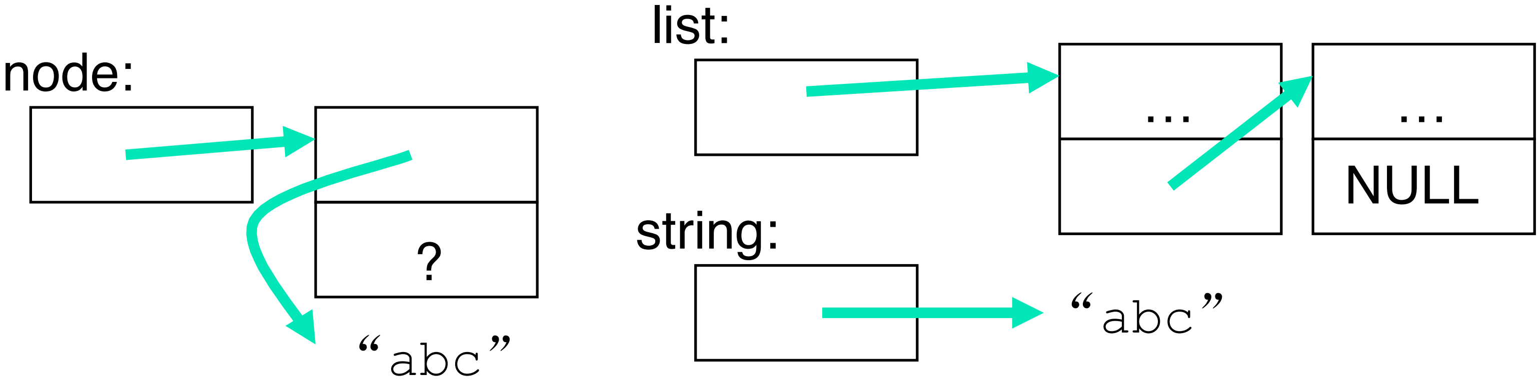
Linked List Example

```
/* 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;
}
```



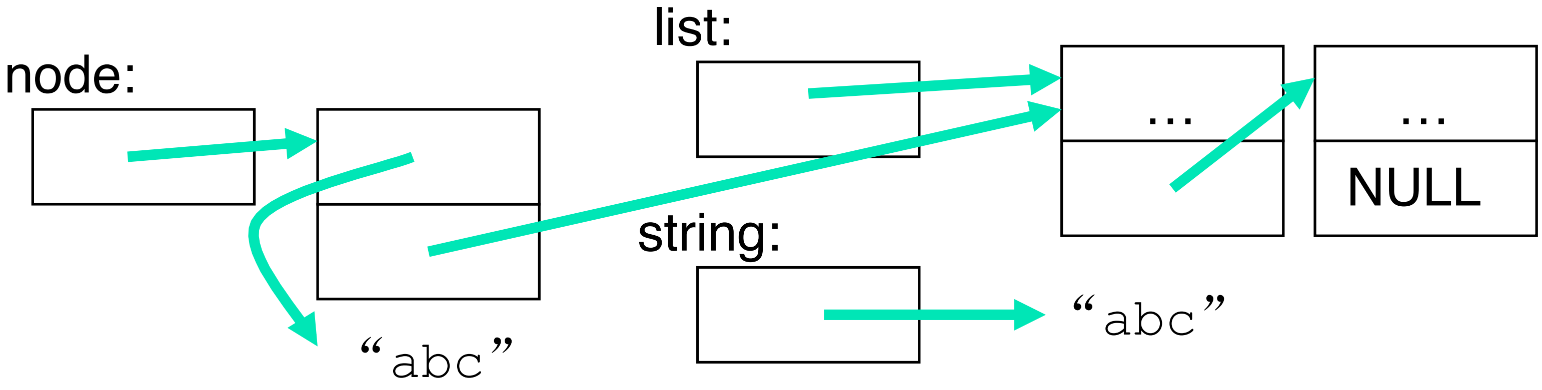
Linked List Example

```
/* 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;
}
```



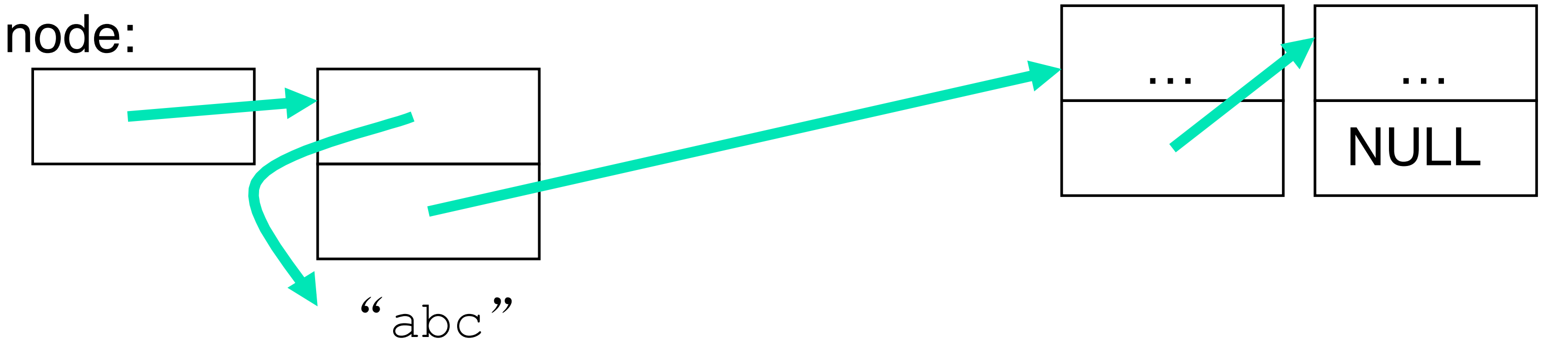
Linked List Example

```
/* 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;
}
```



Conclusion

- 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!"