Reference slides

You ARE responsible for the material on these slides (they're just taken from the reading anyway). These were the slides that generated the fewest questions in years past (i.e., those you could just read and fully understand.)

C Strings

String : an array of characters

- 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;
}
```

\0

Strlen(char s[]) function

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`?)
- `char *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.

length

compare

from right to left

Administrivia

- Read K&R 6 by the next lecture
- There is a language called D!
  - www.digitalmars.com/d/
- Homework expectations
  - Readers don't have time to fix your programs which have to run on lab machines.
  - Code that doesn't compile or fails all of the autograder tests ⇒ 0

Pointers & Allocation (1/2)

- **After declaring a pointer:**

```
int *ptr;
```

`ptr` doesn't actually point to anything yet *(it actually points somewhere - but don't know where!)*. We can either:

  - **make it point to something that already exists, or**
  - **allocate room in memory for something new that it will point to… (next time)**

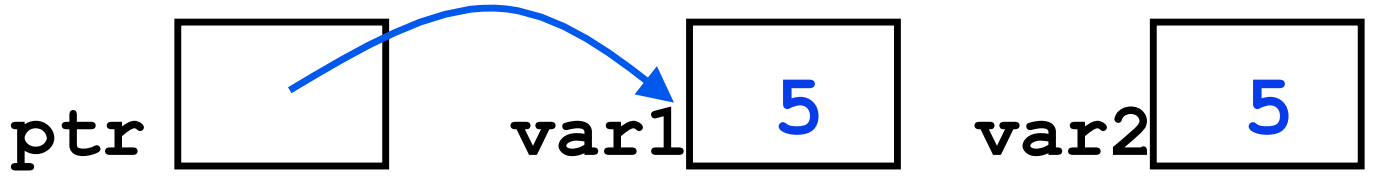
create a new one

Pointers & Allocation (2/2)

- **Pointing to something that already exists:**

```
int *ptr, var1, var2;
  var1 = 5;
  ptr  = &var1;
  var2 = *ptr;
```

- **`var1` and `var2` have room implicitly allocated for them.**



Arrays (one elt past array must be valid)

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

illegal

  - 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

Pointer Arithmetic 指针算术

- **So what's valid pointer arithmetic?**
  - **Add an integer to a pointer.** add an int
  - **Subtract 2 pointers (in the same array).** substract two pointers
  - **Compare pointers (<, <=, ==, !=, >, >=)** compare two pointers
  - **Compare pointer to `NULL` (indicates that the pointer points to nothing).** compare to NULL
- **Everything else is illegal since it makes no sense:** never add/multiply two pointers
  - **adding two pointers**
  - **multiplying pointers** never substract with an integer.
  - **subtract pointer from integer**

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++;
     }
}
```

assign an value first.

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

Pointer Arithmetic (1/2)

- Since a pointer is just a mem address, we can add to it to traverse an array.
- `p+1` returns a ptr to the next array elt.
- `*p++` vs `(*p)++` ?
  - `x = *p++` ⇒ `x = *p ; p = p + 1;`
  - `x = (*p)++` ⇒ `x = *p ; *p = *p + 1;`
- What if we have an array of large structs (objects)?
  - C takes care of it: In reality, `p+1` doesn't add `1` to the memory address, it adds the size of the array element.

Pointer Arithmetic (2/2)

- 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 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!

Arrays vs. Pointers

- An array name is a read-only pointer to the 0th element of the array.
- An array parameter can be declared as an array or a pointer; an array argument can be passed as a pointer.

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

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

Could be written:

```
while (s[n])
```

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.

C Pointer Dangers

- **Unlike Java, C lets you cast a value of any type to any other type without performing any checking.**

```
int x = 1000;
int *p = x;                 /* invalid */
int *q = (int *) x;         /* valid */
```

pointer

an value

cast x into a pointer

- **The first pointer declaration is invalid since the types do not match.**
- **The second declaration is valid C but is almost certainly wrong**
  - **Is it ever correct?**

C Strings Headaches

- **One common mistake is to forget to allocate an extra byte for the null terminator.**

allocate memory by hand

- **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?**
  - **Buffer overrun security holes!**

Common C Error

- There is a difference between assignment and equality

  `a = b` is assignment = assign

  `a == b` is an equality test == equal?

- This is one of the most common errors for beginning C programmers!
  - One solution (when comparing with constant) is to put the var on the right! If you happen to use =, it won't compile.

```
if (3 == a) { ...
```

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.

no methods or inheritance

```
struct point {  /* type definition */
    int x;
    int y;
};

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

struct point p1 = {0,10}; /* x=0, y=10 */

PrintPoint(p1);
```

struct

As always in C, the argument is passed by "value" – a copy is made.

passed by value

in order

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);
```

dot operator

arrow operator

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

malloc()
free()

- 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;
```

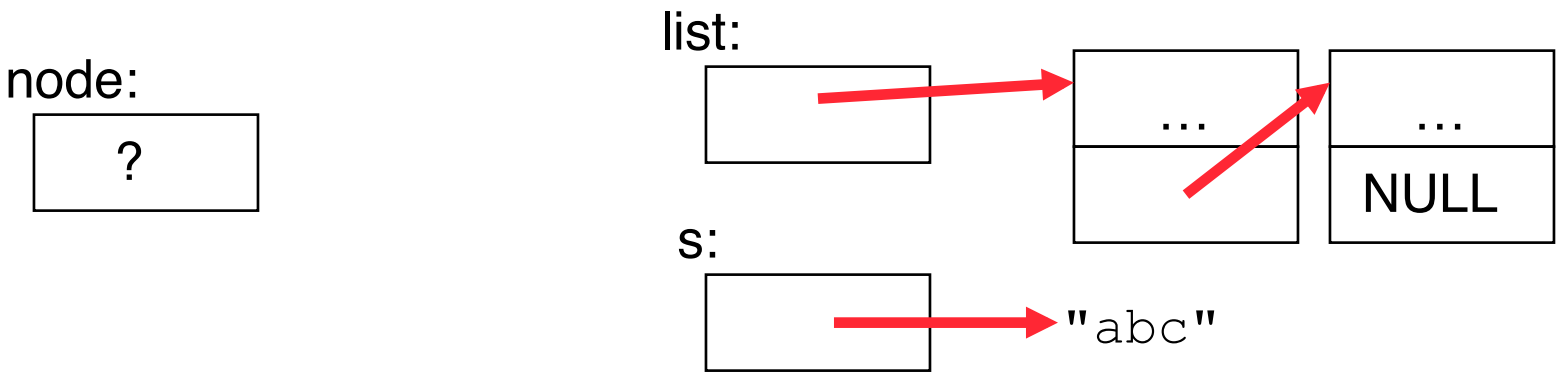
```
/* Note similarity!  */
/* To define 2 nodes */

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));
```

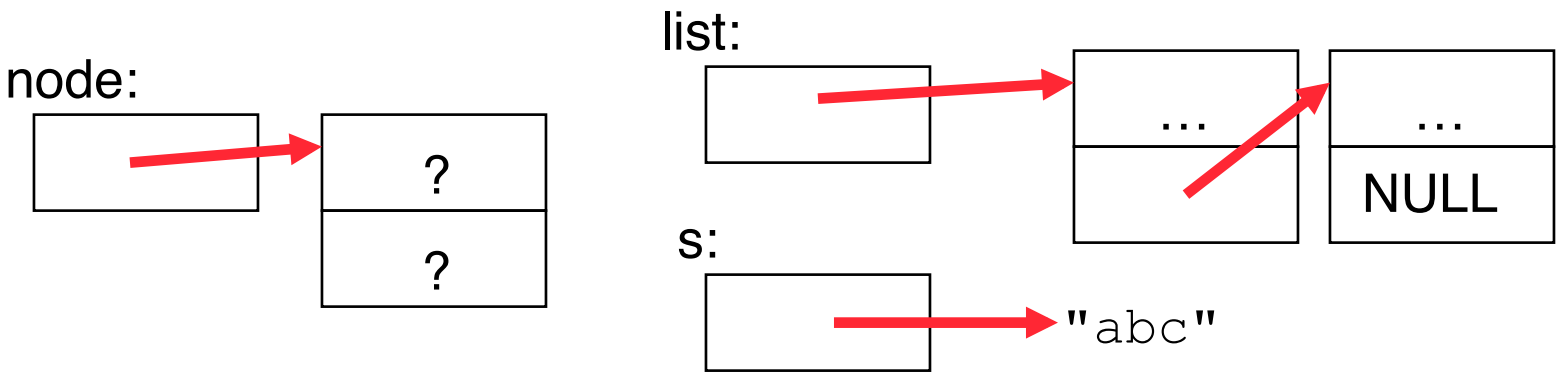
Linked List Example

```
/* 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;
}
```



Linked List Example

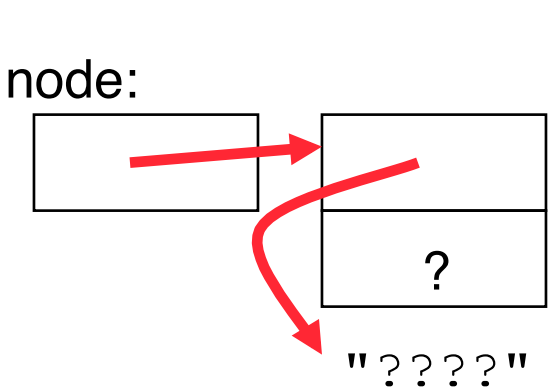
```
/* Add a string to an existing list, 2nd call */
List cons(String s, List list)
{
  List node = (List) malloc(sizeof(NodeStruct));

  node->value = (String) malloc (strlen(s) + 1);
  strcpy(node->value, s);
  node->next = list;
  return node;
}
```

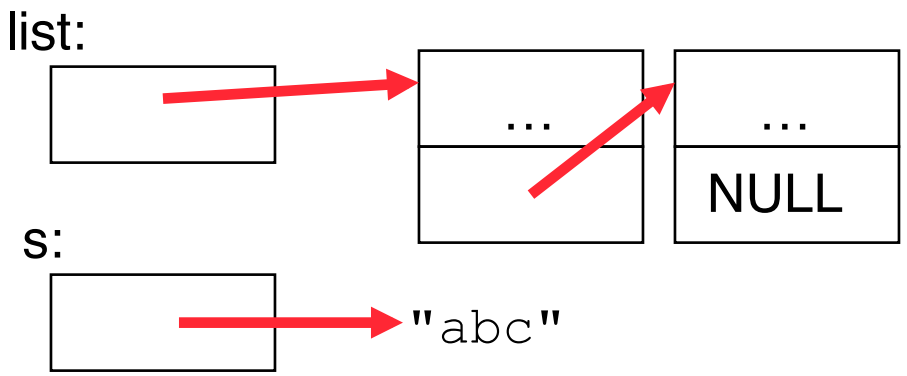


Linked List Example

```
/* 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;
}
```

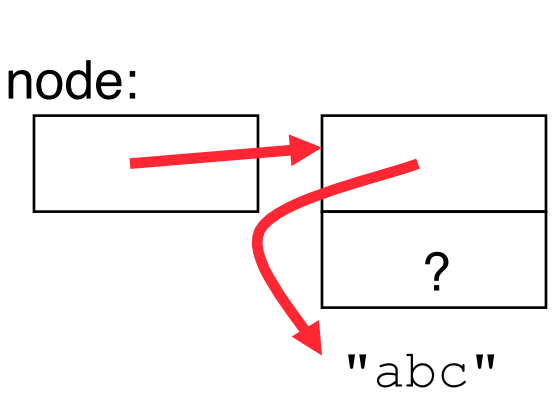
Linked List Example

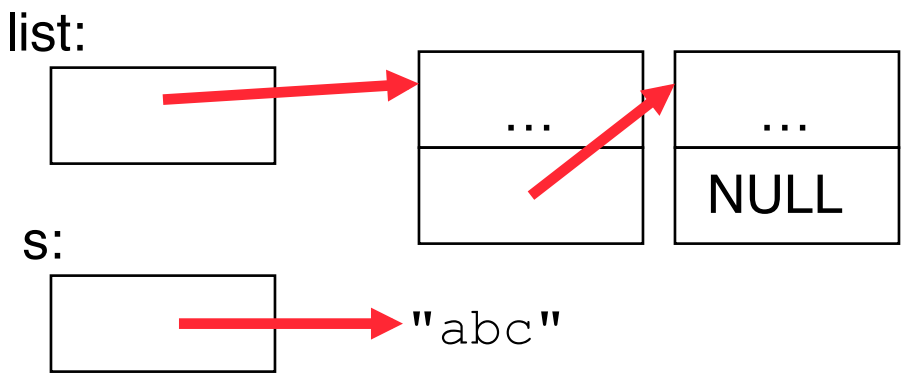




Linked List Example

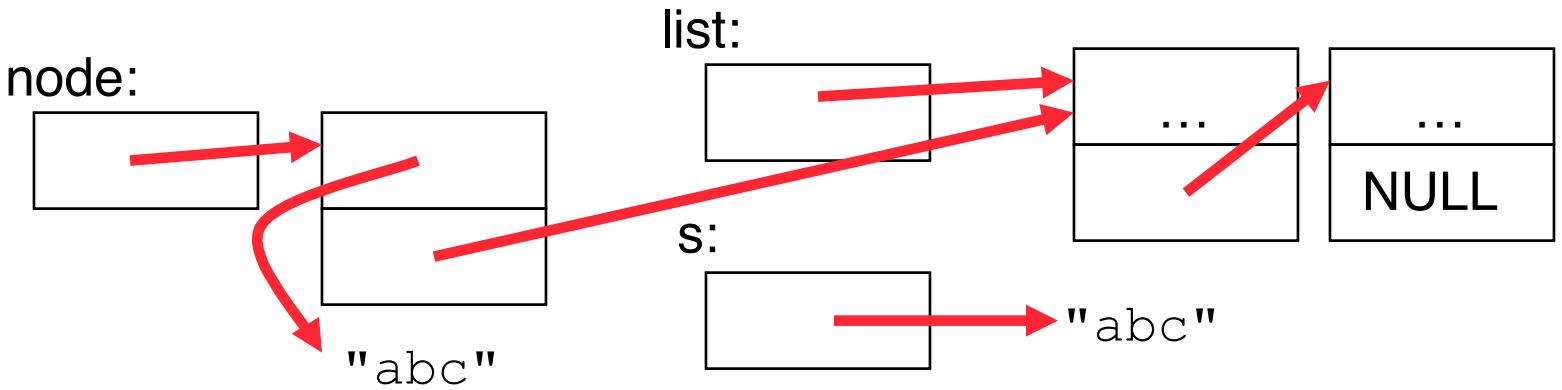
```
/* Add a string to an existing list, 2nd call */
List cons(String s, List list)
{
  List node = (List) malloc(sizeof(NodeStruct));

  node->value = (String) malloc (strlen(s) + 1);
  strcpy(node->value, s);
  node->next = list;
  return node;
}
```





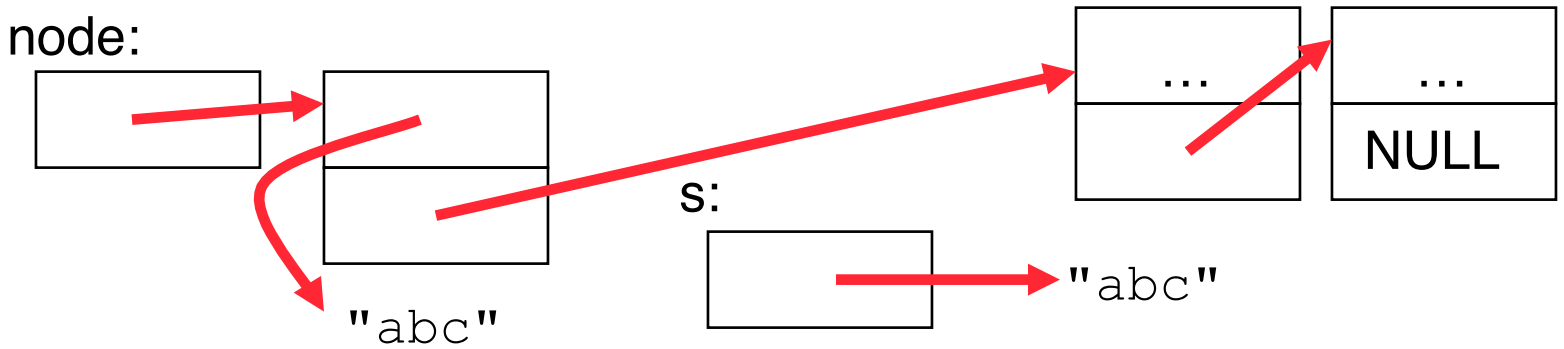
Linked List Example

```
/* 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;
}
```

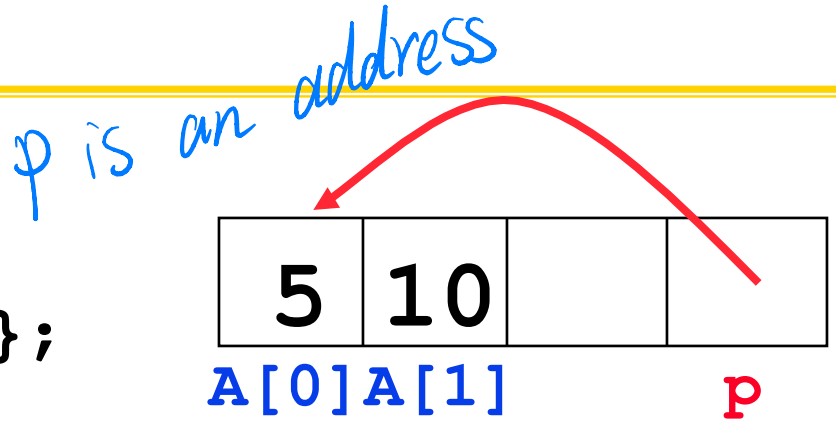


Linked List Example

```
/* 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;
}
```



Peer Instruction

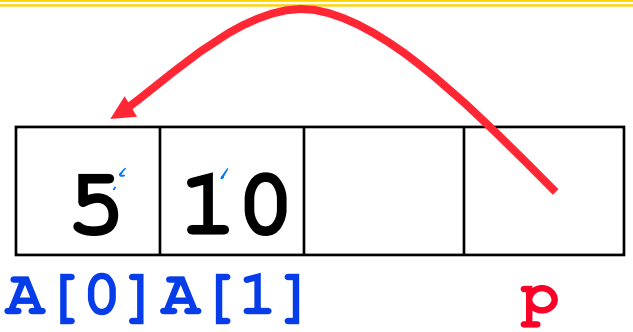



```
int main(void){
  int A[] = {5,10};
  int *p = A;

  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]);
  }
```

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

a) 101 10 5 10 then 101 11 5 11
b) 104 10 5 10 then 104 11 5 11
c) 101 <other> 5 10 then 101 <3-others>
d) 104 <other> 5 10 then 104 <3-others>
e) One of the two printfs causes an ERROR

Peer Instruction Answer




```
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?

a) 101 10 5 10 then 101 11 5 11
b) 104 10 5 10 then 104 11 5 11
c) 101 <other> 5 10 then 101 <3-others>
d) 104 <other> 5 10 then 104 <3-others>
e) One of the two printfs causes an ERROR

Pointer Arithmetic Peer Instruction Q

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 Ans

- 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

address

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

"And in Conclusion…"

- Pointers and arrays are virtually same
- C knows how to increment pointers
- Create abstraction with structures