



Variables, Pointers, and Arrays

CS 217

<http://www.cs.princeton.edu/courses/archive/fall05/cos217/>



Overview of Today's Lecture

- **Pointers**

- Differences between value, variable, and pointer
- Using pointers to do call-by-reference in C

- **Struct**

- Multiple values grouped together
- Dereferencing to access individual elements

- **Arrays**

- List of elements of the same type
- Relationship between arrays and pointers
- Example program to reverse an array

- **Strings**

- Array of characters ending in '\0'

Values, Variables, and Pointers



- Value

- E.g., `'M'`

- Variable

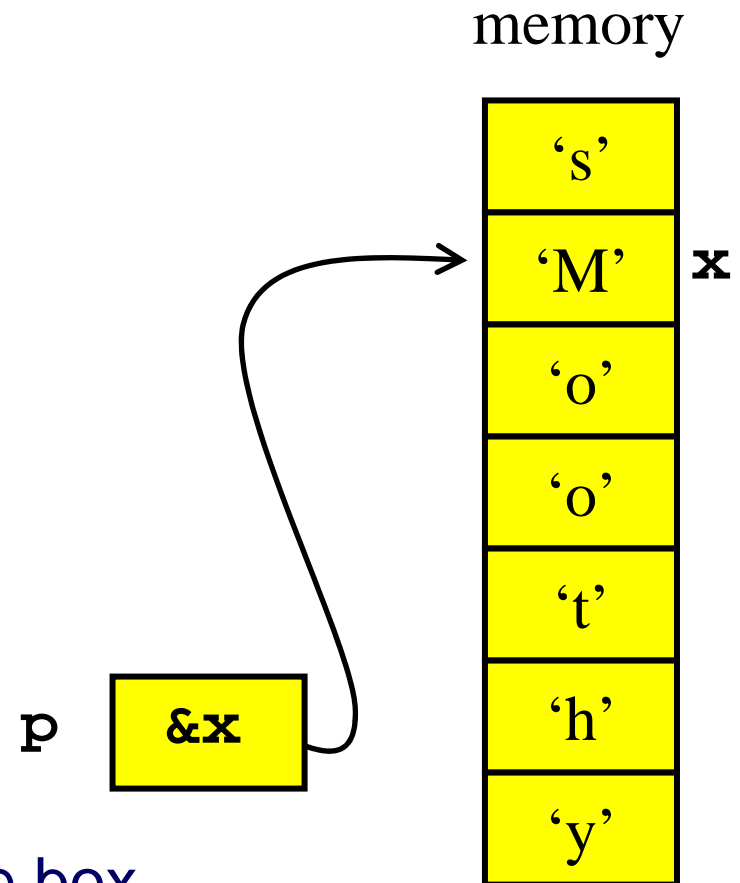
- A named box that holds a value
- E.g., `char x = 'M';`

- Pointer value

- Address of the box
- E.g., `&x`

- Pointer variable

- A box holding the address of the box
- E.g., `char* p = &x;`





Example Program

```
#include <stdio.h>
main() {
    char x = 'M';
    char* p = &x;
    printf("Value of x is %c\n", x);
    printf("Address of x is %u\n", p);
    printf("Address of p is %u\n", &p);
}
```

- Output

- Value of x is M
- Address of x is 4290770463
- Address of p is 4290770456



Values vs. Variables

```
int n;
```



```
n = 217;
```



```
n = n + 9;
```



```
3 = n;
```

??

```
&n
```

a pointer value

```
&3
```

??

What is this?

`*(&n)`



Call by Value is Not Enough

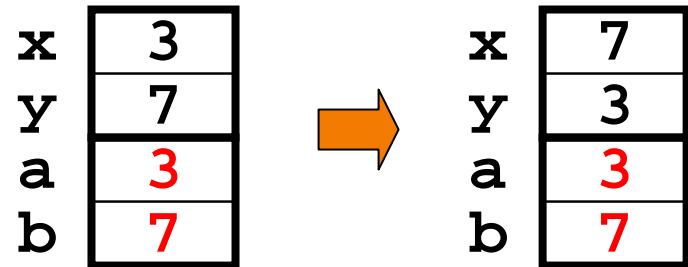
- Function parameters are transmitted by value
 - Values copied into “local variables”

```
void swap(int x, int y)
{
    int t;

    t = x;
    x = y;
    y = t;
}
```

No!

```
main() {
    ...
    swap(a,b);
    ...
}
```



Call by Reference Using Pointers



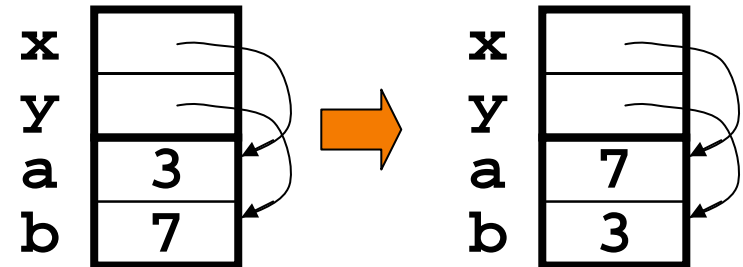
- Use pointers to pass variables “by reference”

```
void swap(int *x, int *y)
{
    int t;

    t = *x;
    *x = *y;
    *y = t;
}

main() {
    ...
    swap(&a, &b);
    ...
}
```

Yes

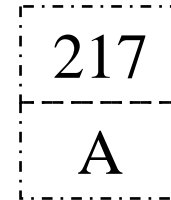




Structures

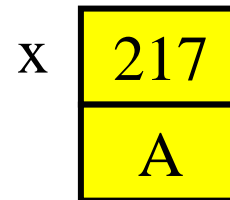
A struct value is a bunch of values glued together

```
struct pair {  
    int number;  
    char grade;  
};
```



A struct variable is a box holding a struct value

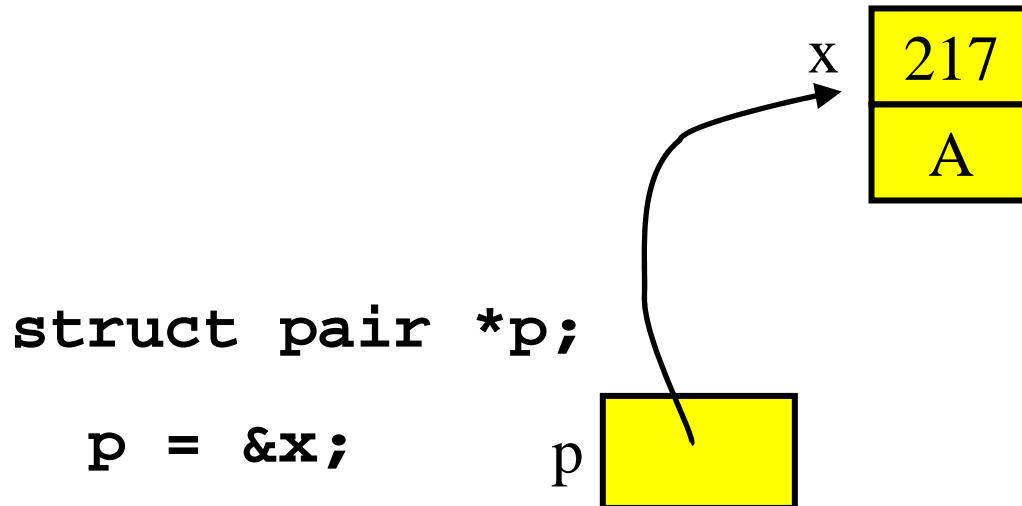
```
struct pair x;  
x.number = 217;  
x.grade = 'A';
```





Pointers to structs

```
struct pair {int number; char grade;};  
struct pair x;    x.number=217;    x.grade='A';
```



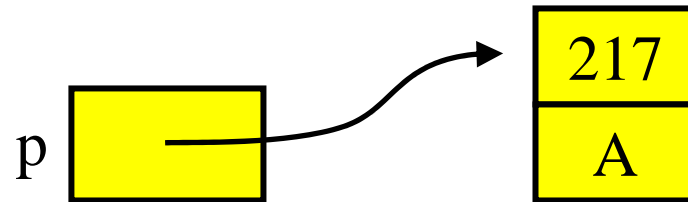
```
int n = (*p).number;  
char g = (*p).grade;
```

The diagram shows the result of dereferencing the pointer. Two new variables are shown: `n` and `g`. `n` is shown as a yellow box containing the value `217`, and `g` is shown as a yellow box containing the value `A`. This represents the data extracted from the struct pointed to by `p`.

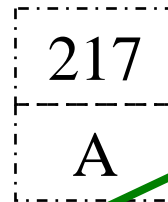


Dereferencing Fields

```
struct pair {int number; char grade;} *p;
```



`*p`



Easier-to-use
notation

```
int n = (*p).number;
```

```
int n = p->number;
```

```
char g = (*p).grade;
```

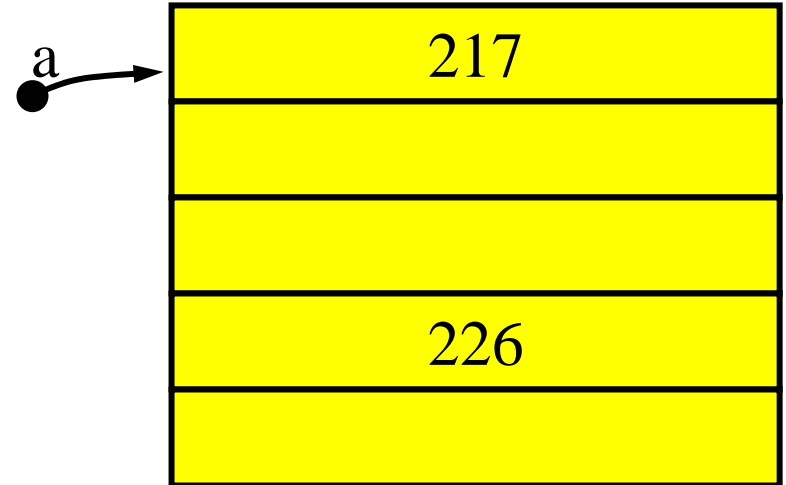
```
char g = p->grade;
```



Arrays in C



```
int a[5];
```



a is a *value* of type “pointer to int”

What is “**a**” in the picture above?

a is the pointer *constant*, not the five consecutive memory locations!

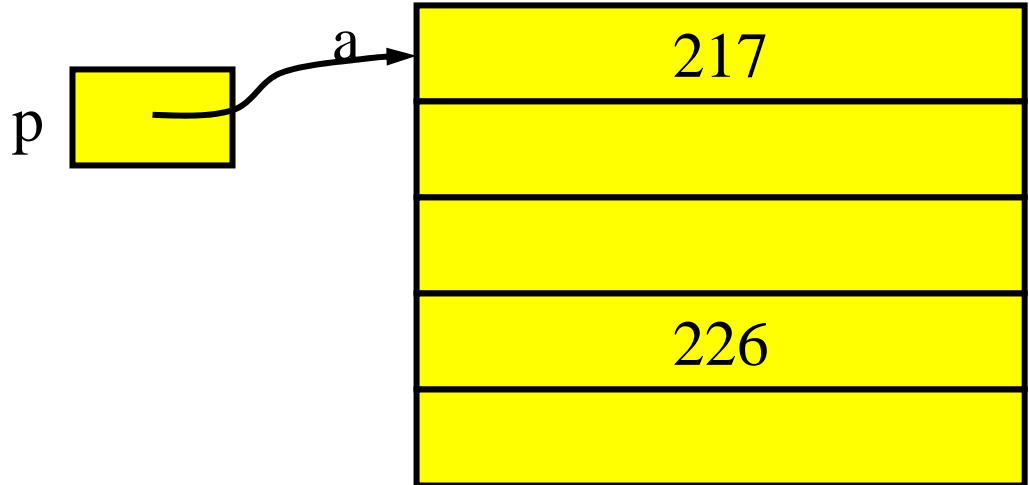
Arrays and Pointers



```
int a[5];
```

```
int *p;
```

```
p = a;
```



a is a *value* of type “pointer to int” (int *)

p is a *variable* of type “pointer to int” (int *)

OK: `p = a; if (a == p)...; a[i] = p[j];`

Wrong: `a = p; 3 = i;`

C Does Not Do Bounds Checking!



```
int a[5];
```

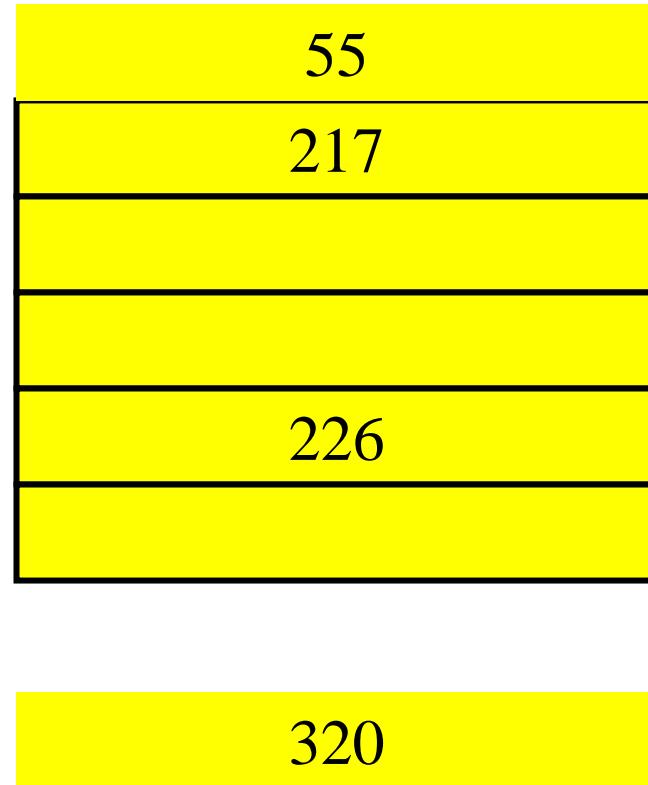
```
a[0] = 217;
```

```
a[3] = 226;
```

```
a[-1] = 55;
```

```
a[7] = 320;
```

a



Unpleasant if you happened to have another variable before the array variable **a**, or after it!

Arrays and Pointers



```
int a[5];
```

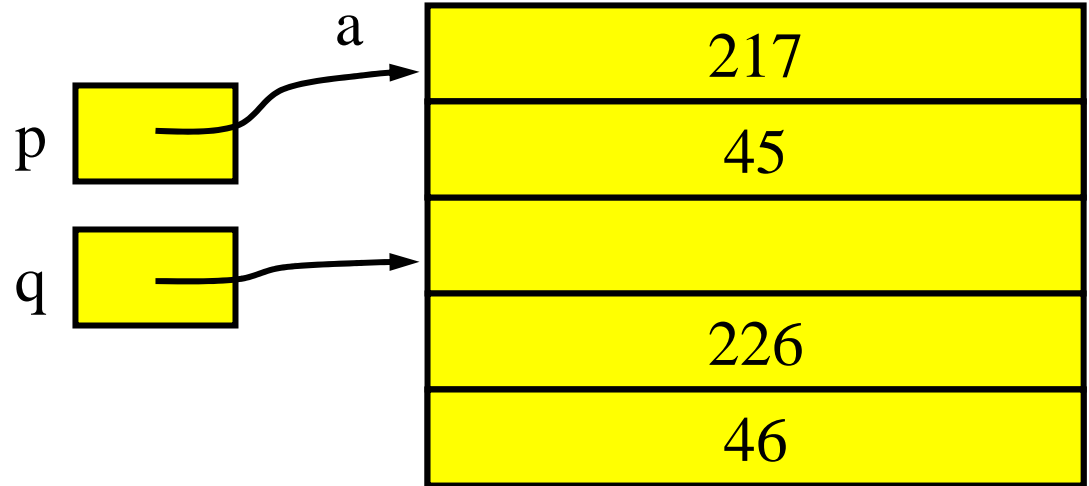
```
int *p, *q;
```

```
p = a;
```

```
p[1] = 44;
```

```
q = p + 2;
```

```
q[-1] = 45; q[2] = 46;
```



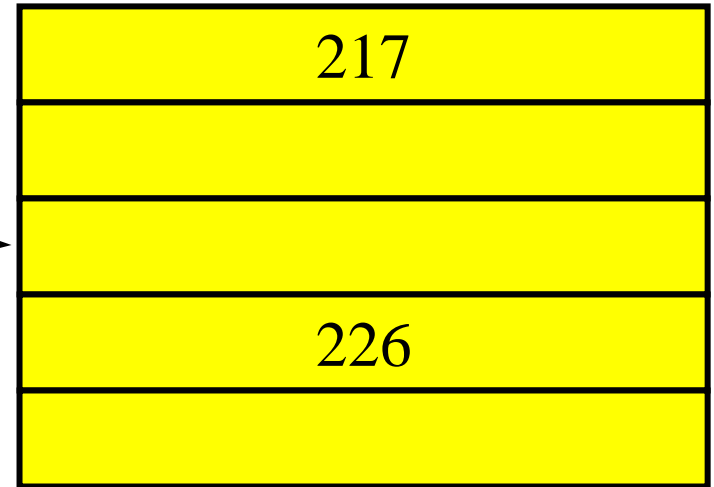


Pointer Arithmetic

```
int a[5];
```



a



4 bytes

Subscript: $a[i]$ “means” $*(a+i)$

```
int *p;
```

```
p = a + 2;
```

Note: arithmetic scales by data size (e.g., int of 4 bytes)

Quaint usage of pointer arithmetic



Add up the elements of an array:

```
#define N 1000
```

```
int addup(int a[N]) {  
    int sum, *p;  
    for (p=a; p<a+N; p++)  
        sum += *p;  
    return sum;  
}
```

More straightforwardly:

```
int addup(int a[N]) {  
    int sum, i;  
    for (i=0; i<N; i++)  
        sum += a[i];  
    return sum;  
}
```




Array Parameters to Functions

```
void printArray(int *p, int n) {  
    int i;  
    for (i=0; i<n; i++)  
        printf("%d\n",p[i]);  
}  
  
int fib[5] = {1, 1, 2, 3, 5};  
  
int main(...) {  
    printArray(fib, 5);  
}
```



Array Params \equiv Pointer Params

```
void printArray(int *p, int n) { ... }
```

```
void printArray(int p[5], int n) { ... }
```

```
void printArray(int p[ ], int n) { ... }
```

```
void printArray(int p[1000], int n) { ... }
```

All these declarations are equivalent!

```
int main(...) {
```

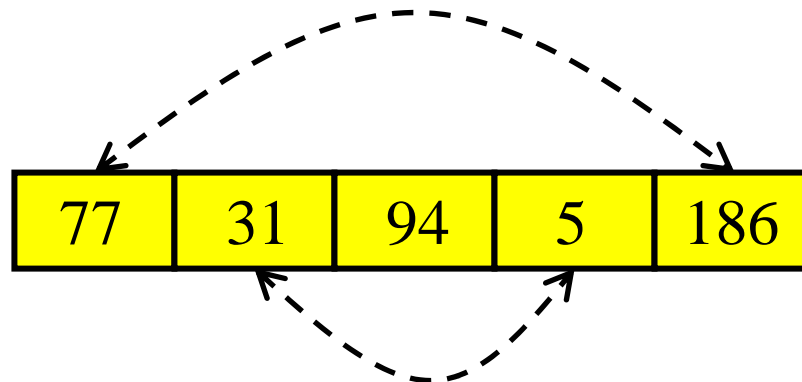
```
    printArray(fib, 5);
```

```
}
```

Example Program: Reverse Array



- Reverse the values in an array
 - Inputs: integer array **a**, and number of elements **n**
 - Output: values of **a** stored in reverse order
- Algorithm
 - Swap the first and last elements in the array
 - Swap the second and second-to-last elements
 - ...





Example of Array by Reference

```
void reverse (int a[], int n) {  
    int l, r, temp;  
    for (l=0, r=n-1; l<r; l++, r--) {  
        temp = a[l];  
        a[l] = a[r];  
        a[r] = temp;  
    }  
}  
  
int main(...) {  
    reverse(fib, 5);  
}
```

Strings



A string is just an array of characters (pointer to character), terminated by a '\0' char (a null, ASCII code 0).

```
char mystring[6] = {'H','e','l','l','o','\0'};
```

```
char mystring[6] = "Hello";
```

```
char mystring[] = "Hello";
```

} Equivalent

mystring

H	e	l	l	o	\0
---	---	---	---	---	----

```
char *yourstring = "Hello";
```

} Different

yourstring

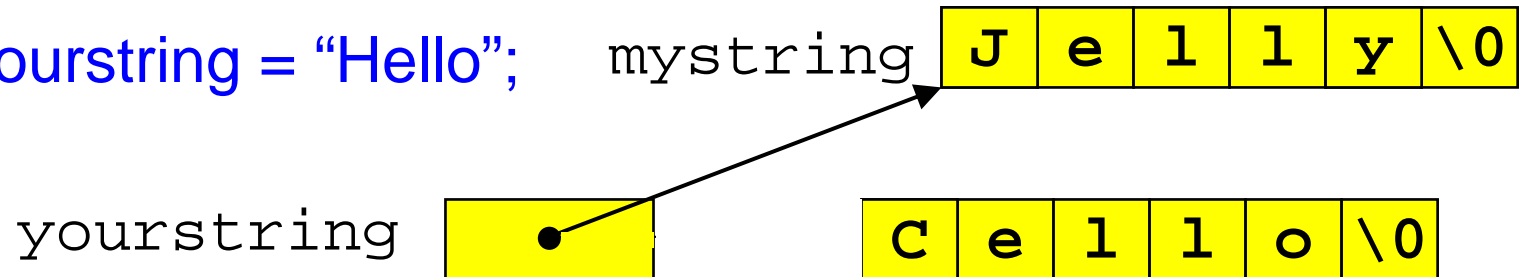
•	→	H	e	l	l	o	\0
---	---	---	---	---	---	---	----

Char Array and Pointer Manipulation



```
char mystring[] = "Hello";
```

```
char *yourstring = "Hello";
```



```
mystring[0] = 'J';
```

```
yourstring[0] = 'C';
```

```
yourstring = mystring;
```

```
yourstring[4] = 'y';
```

```
mystring = yourstring;
```



Printing a String

```
printf("%s",mystring);
```

mystring

H	e	l	l	o	\0
---	---	---	---	---	----

```
int i;
```

```
for (i=0; mystring[i]; i++)
```

```
    putchar(mystring[i]);
```

or,

```
char *p;
```

```
for (p=mystring; *p; p++)
```

```
    putchar(*p);
```



String termination

```
char mystring[] = "Hello";
```

mystring

H	e	x	l	o	!
---	---	---	---	---	---

```
mystring[2] = 0;      equivalently, mystring[2]='\0';
```

```
printf("%s\n",mystring);
```

He

```
mystring[2] = 'x'; mystring[5] = '!';
```

```
printf("%s\n",mystring);
```

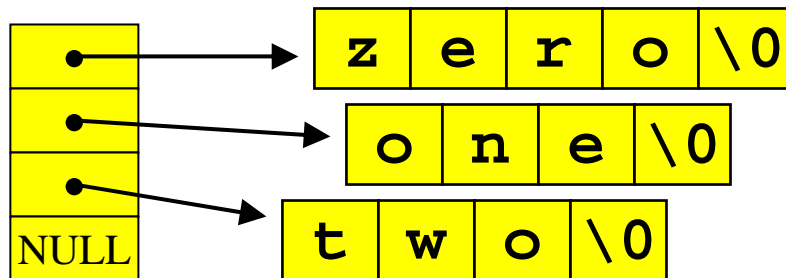
What will happen?



Boxes and Arrows

In designing and analyzing your data structures, draw pictures!

Example: you want an array of strings



```
char *query[4] =  
    {"zero", "one", "two", NULL};
```

how to parse it: `*(query[4])`

postfix operators bind tighter than prefix; whenever you're not sure, just put the parentheses in

Summary of Today's Class



- C variables
 - Pointer
 - Struct
 - Array
 - String
- First programming assignment (due Sun Oct 2 at 8:59pm)
 - Use of pointers is optional
 - Using a global variable instead is okay
- Reading for next week
 - Chapters 2, 3, and 4 of *The C Programming Language*
 - Chapter 4 of *Practice of Programming*