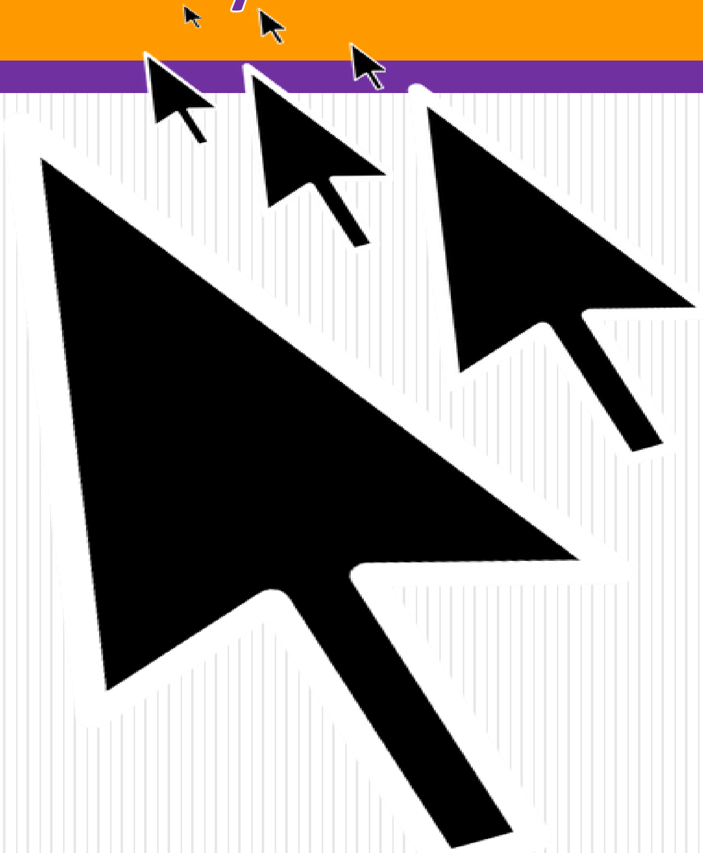
# Programming in C

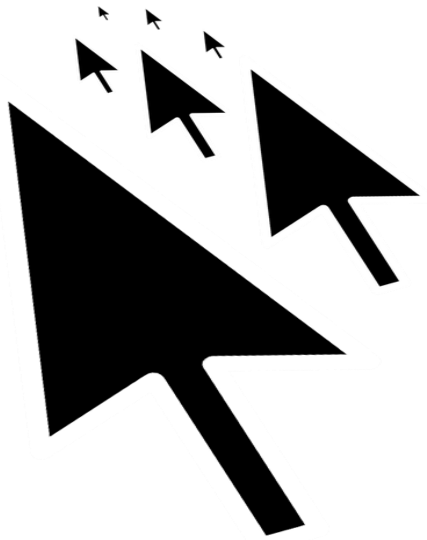
## Chapter 10 / 16a (p.384)

## Pointers / Dynamic Memory Allocation



## Pointer Variable

- A variable that stores a memory address
  - Allows C programs to simulate call-by-reference
  - Allows a programmer to create and manipulate dynamic data structures
- Must be defined before it can be used
  - Should be initialized to NULL or valid address

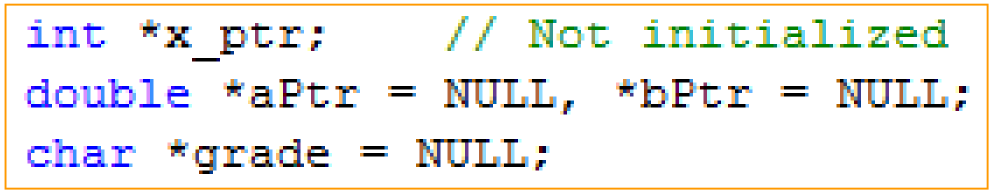


## Declaring Pointers

Declaration of pointers

```
<type> *variable
<type> *variable = initial-value
```

Examples:



```
int *x_ptr;     // Not initialized
double *aPtr = NULL, *bPtr = NULL;
char *grade = NULL;
```

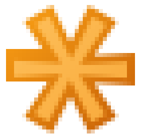
## Pointers

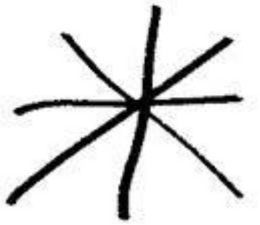
```
char *grade = NULL;
```

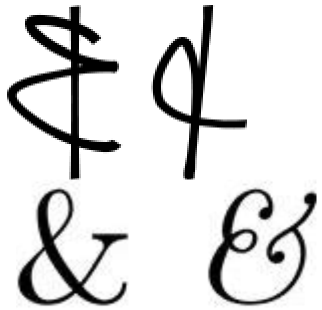
- A pointer variable has two associated values:
  - Direct value
    - address of another memory cell
    - Referenced by using the variable name
  - Indirect value
    - value of the memory cell whose address is the pointer's direct value.
    - Referenced by using the indirection operator *

# Pointer Operators

- Come before a variable name
  - * operator
    - Indirection operator or dereferencing operator
    - Returns a synonym, alias or nickname to which its operand points
  - & operator
    - Address of operator
    - Returns the address of its operand







## Pointer Variables

- One way to store a value in a pointer variable is to use the & operator

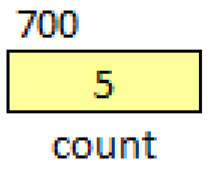
```
int count = 5;
int *countPtr = &count;
```

  - The address of count is stored in countPtr
  - We say, *countPtr points to count*

## Pointer Variables

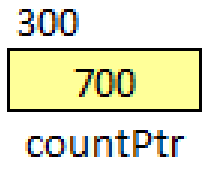
- Assume count will be stored in memory at location 700 and countPtr will be stored at location 300
  - `int count = 5;`

    causes 5 to be stored in count



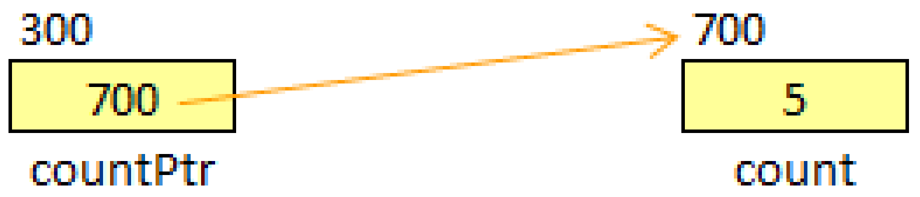
  - `int *countPtr = &count;`

    causes the address of count to be stored in countPtr



## Pointer Variables

We represent this graphically as

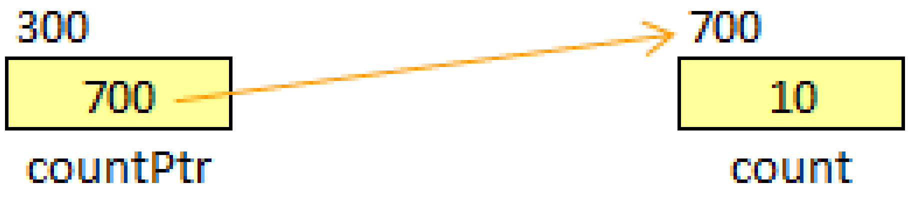


## Pointer Variables

- The indirection / dereferencing operator is *

  -
    ```
    *countPtr = 10;
    ```

  stores the value 10 in the address pointed to by countPtr



# Pointer Variables

- The character * is used in two ways:

1. To declare that a variable is a pointer
   - Pre-pending a variable with a * in a declaration declares that the variable will be a pointer to the indicated type instead of a regular variable of that type

2. To access the location pointed to by a pointer
   - Pre-pending a variable with a * in an expression indicates the value in the location pointed to by the address stored in the variable

## Simulating By Reference

- Invoked function uses * in formal parameters

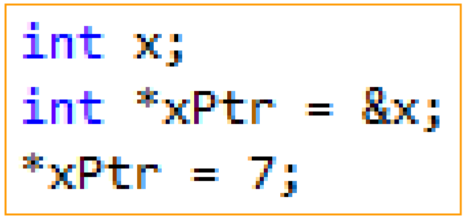
```
void increment(int *n) {
    *n += 1;  // or (*n)++;
}
```

- Invoking function uses & in actual parameters

```
int count = 0;
increment(&count);
printf("%d\n", count); // Prints 1
```

## Pointer Variables and Arrays

- Given

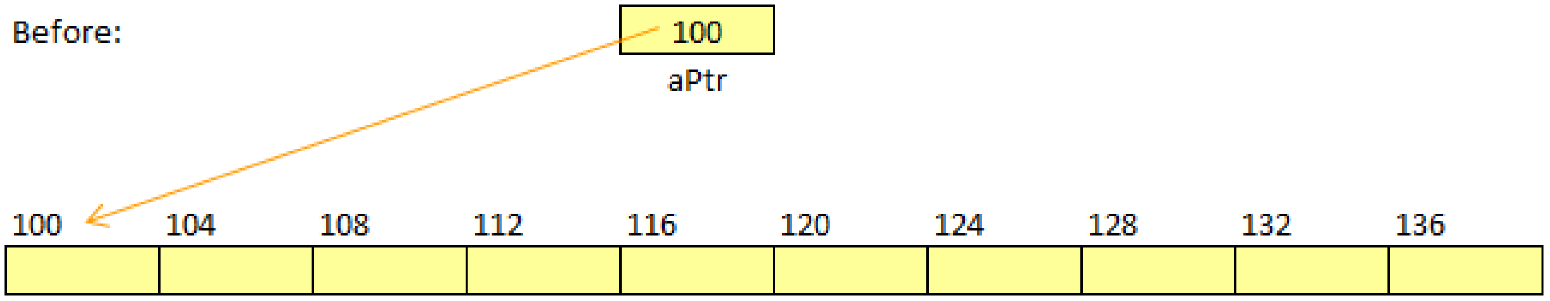


```
int x;
int *xPtr = &x;
*xPtr = 7;
```

  - The compiler will know how many bytes to copy into the memory location pointed to by *xPtr*
  - Defining the type that the pointer points to permits a number of other interesting ways a compiler can interpret code

## Pointer Variables and Arrays

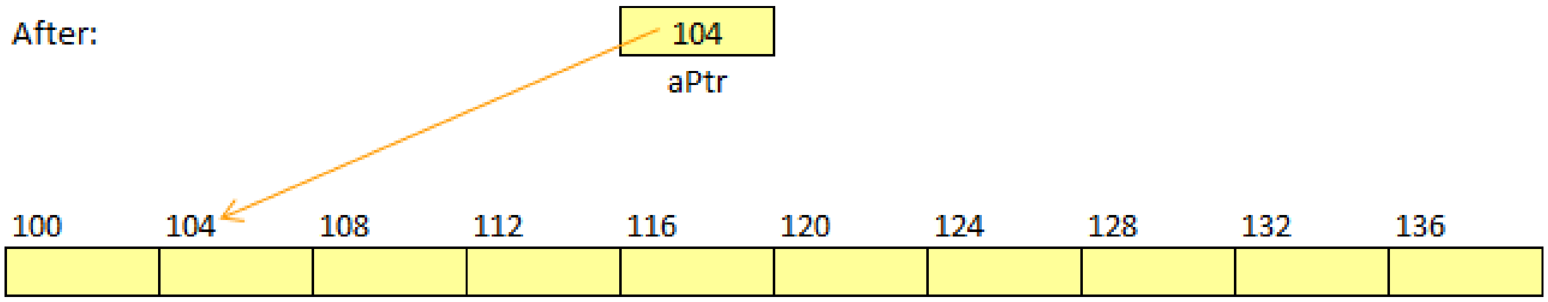
- Consider a block in memory consisting of ten integers of 4 bytes in a row at location $100_{10}$
- Now, let's say we point an integer pointer *aPtr* at the first of these integers



- What happens when we write `aPtr = aPtr + 1;` ?

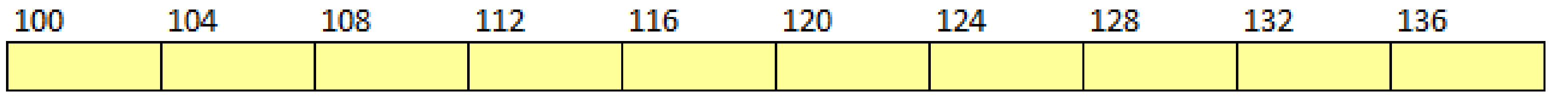
# Pointer Variables and Arrays

- Because the compiler "knows"
  - This is a pointer (i.e. its value is an address)
  - That it points to an integer of length 4 at location 100
- Instead of 1, `aPtr = aPtr + 1;` adds 4 to *aPtr*
  - Now aPtr "points to" the next integer at location 104
  - Same for: aPtr+=1, aPtr++, and ++aPtr



## Pointer Variables and Arrays

- Since a block of 10 integers located contiguously in memory is, by definition, an array of integers, this brings up an interesting relationship between arrays and pointers

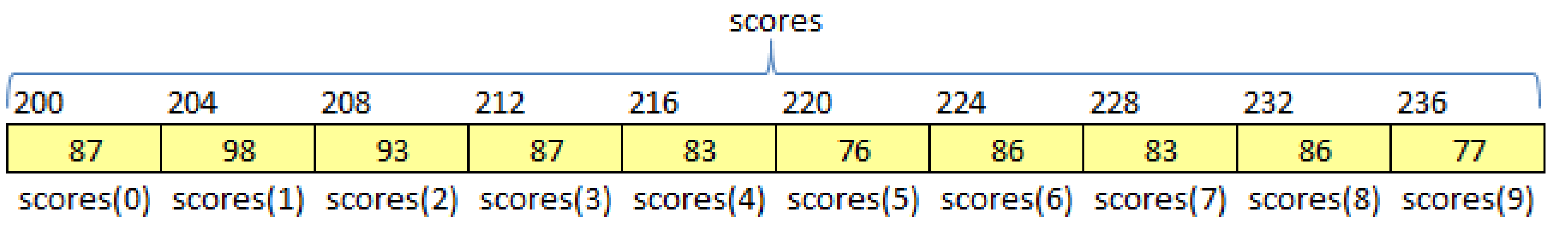


## Pointer Variables and Arrays

- Consider this array allocated at location 200

```
int scores[10] = {87,98,93,87,83,76,86,83,86,77};
```

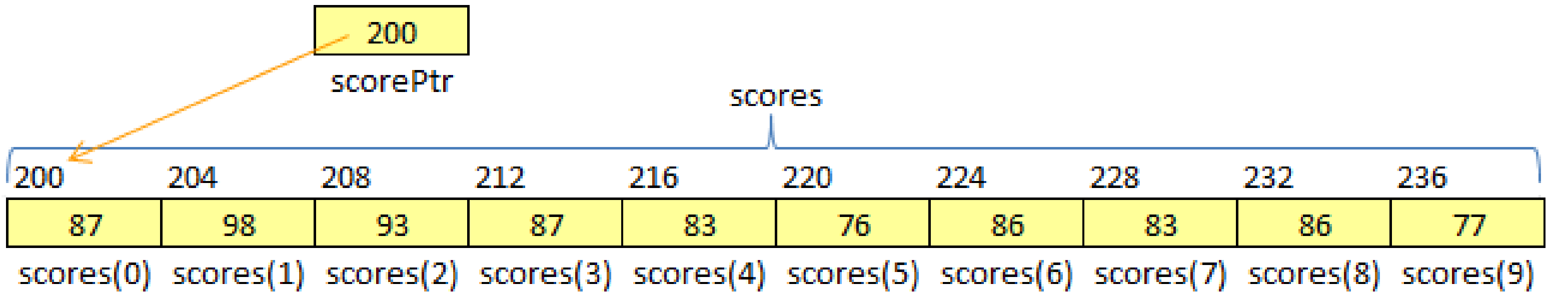
  - We have an array containing 10 integers
  - We refer to each of these integers by means of a subscript to *scores*
    - Using *scores[0]* through *scores[9]*



# Pointer Variables and Arrays

- The name of an array and the address of the first element in the array represent the same thing
- Consequently, we could alternatively access them via a pointer:

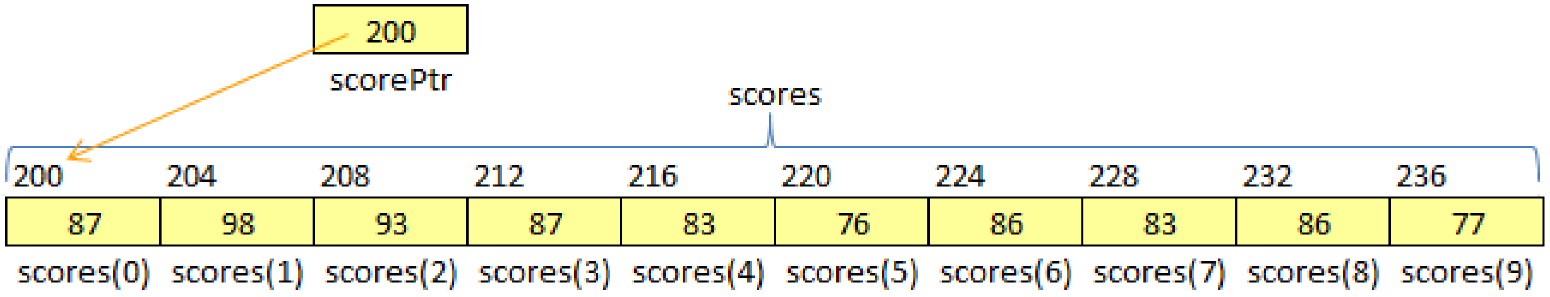
```
int scores[10] = {87,98,93,87,83,76,86,83,86,77};
...
int *scorePtr = NULL;
...
scorePtr = &scores[0];  // Points to first element
```



# Pointer Variables and Arrays

- The name of an array is a pointer constant to the first element of the array
- So, we could also use :

```
int scores[10] = {87,98,93,87,83,76,86,83,86,77};
...
int *scorePtr = NULL;
...
scorePtr = scores;    // Points to array
```



## Pointer Arithmetic and Arrays

- If scorePtr is pointing to a specific element in the array and n is an integer,

  `scorePtr + n`

  is the pointer value *n* elements away
- We can access elements of the array either using the array notation or pointer notation
  - If scorePtr points to the first element, the following two expressions are equivalent:

| | |
|---|---|
| `scores[n]` | **Array notation** |
| `*(scorePtr + n)` | **Pointer notation** |

## Pointers and
## Dynamic Allocation of Memory

- So far, we have always allocated memory for variables that are located on the **stack**
  - Size of such variables must be known at compile time
- Sometimes convenient to allocate memory at run time
  - System maintains a second storage area called the **heap**
  - Functions **calloc** and **malloc** allocate memory as needed of size needed

## Pointers and Dynamic Allocation of Memory

1. Use allocating function (such as **malloc()**, **calloc()**, etc.)
   - Returns void pointer
     - void * indicates a pointer to untyped memory
     - Will have to cast the returned value to the specific type needed
2. Use memory through the pointer notation
3. Release allocated space when no longer needed, so that it can be reused

## Pointers and Dynamic Allocation of Memory: calloc

- **calloc**
  - Used to dynamically create an array in the heap
  - Contiguous allocation
    - Initialized to binary zeros
  - Must `#include <stdlib.h>`
  - Takes two arguments
    1. Number of array elements
    2. Amount of memory required for one element
       - Use sizeof function / operator
  - Returns
    - Void pointer if successful
    - NULL if unsuccessful

## Pointers and Dynamic Allocation of Memory: calloc

- **Example 1: String**

```
const int str_len = 500;
char *str_ptr = NULL;
...
str_ptr = (char *) calloc(str_len, sizeof(char));
if (str_ptr == NULL) {
   printf("Halting: Unable to allocate string.\n");
   exit(1);
}
```

- **Example 2: Integers**

```
const int arraySize = 1000;
int *arrayPtr = NULL;
...
arrayPtr = (int *) calloc(arraySize, sizeof(int));
if (arrayPtr == NULL) {
   printf("Halting: Unable to allocate array.\n");
   exit(1);
}
```

## Pointers and Dynamic Allocation of Memory: malloc

- **malloc**
  - Used to dynamically get memory from heap
  - Contiguous allocation
    - No initialization
  - Must `#include <stdlib.h>`
  - Takes one argument
    - Total amount of memory required
  - Returns
    - Void pointer if successful
    - NULL if unsuccessful

## Pointers and Dynamic Allocation of Memory: malloc

- **Example 1: String**

```
const int str_len = 500;
char *str_ptr = NULL;
...
str_ptr = (char *) malloc(str_len);
if (str_ptr == NULL) {
   printf("Halting: Unable to allocate string.\n");
   exit(1);
}
```

- **Example 2: Integers**

```
const int arraySize = 1000;
int *arrayPtr = NULL;
...
arrayPtr = (int *) malloc(arraySize * sizeof(int));
if (arrayPtr == NULL) {
   printf("Halting: Unable to allocate array.\n");
   exit(1);
}
```

## Pointers and Dynamic Allocation of Memory: free

- **free**
  - Used to dynamically release memory back to heap
  - Contiguous deallocation
  - Must `#include <stdlib.h>`
  - Takes one argument
    - Pointer to beginning of allocated memory
  - Good idea to also NULL pointer if reusing

## Pointers and Dynamic Allocation of Memory: free

- **Example 2 with free**

```
const int arraySize = 1000;
int *arrayPtr = NULL;
...
arrayPtr = (int *) malloc(arraySize * sizeof(int));
if (arrayPtr == NULL) {
   printf("Halting: Unable to allocate array.\n");
   exit(1);
}
...
free(arrayPtr);
arrayPtr = NULL;
```

# Programming in C

Chapter 10 / 16a (p.384)

Pointers / Dynamic Memory Allocation

*THE END*