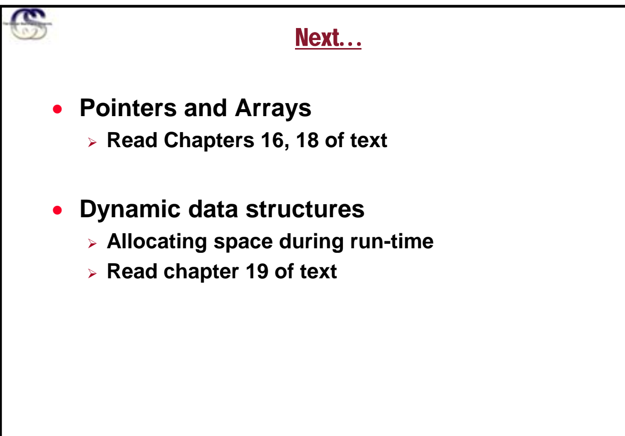
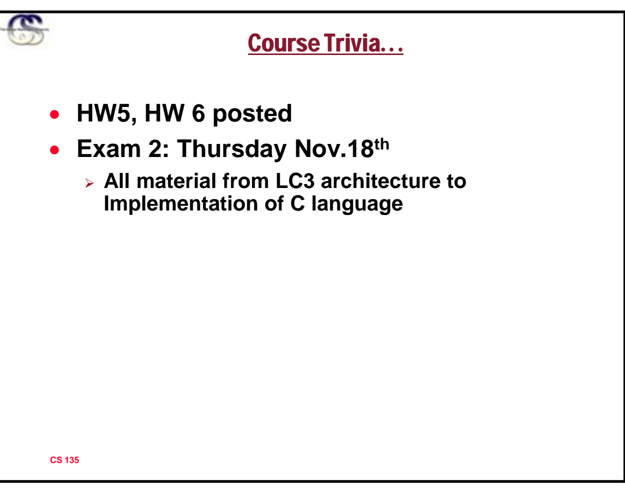
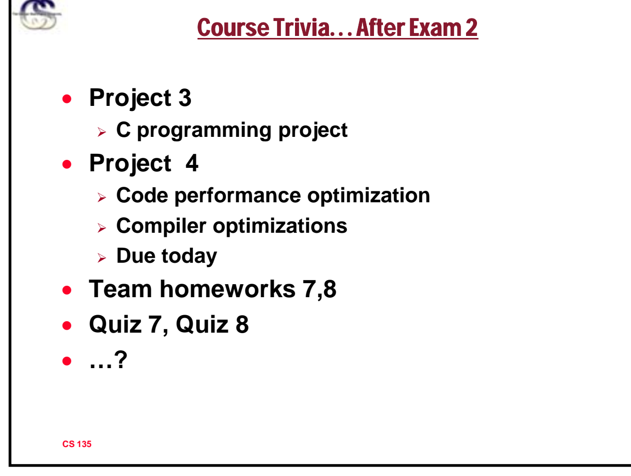
CS 135: Computer Architecture I
Instructor: Prof. Bhagi Narahari
Dept. of Computer Science
Course URL: www.seas.gwu.edu/~bhagiweb/cs135/


Next…
Pointers and Arrays
Read Chapters 16, 18 of text
Dynamic data structures
Allocating space during run-time
Read chapter 19 of text
CS 135


Course Trivia…
HW5, HW 6 posted
Exam 2: Thursday Nov.18th
All material from LC3 architecture to Implementation of C language
CS 135

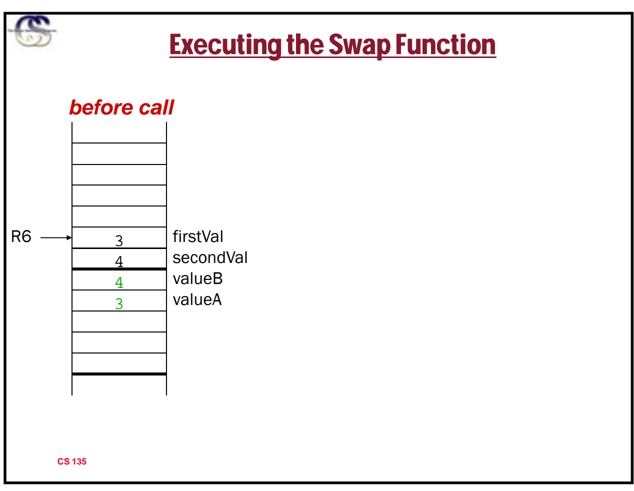

Course Trivia… After Exam 2
Project 3
C programming project
Project 4
Code performance optimization
Compiler optimizations
Due today
Team homeworks 7,8
Quiz 7, Quiz 8
…?
CS 135

Passing by value is not enough

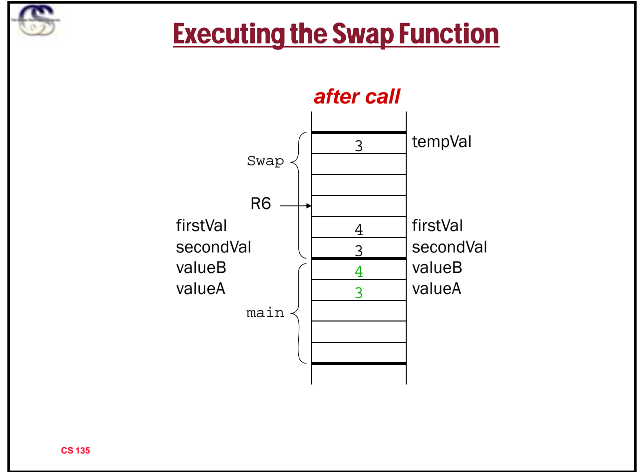
- Parameters are passed by value
  - Arguments pushed onto run-time stack
- Consider the following function that's supposed to swap the values of its arguments.

```
void Swap(int firstVal, int
secondVal)
{
  int tempVal = firstVal;
  firstVal = secondVal;
  secondVal = tempVal;
}
```

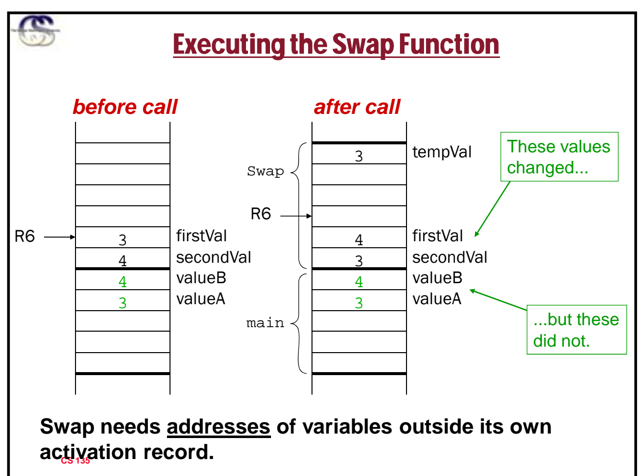

Executing the Swap Function



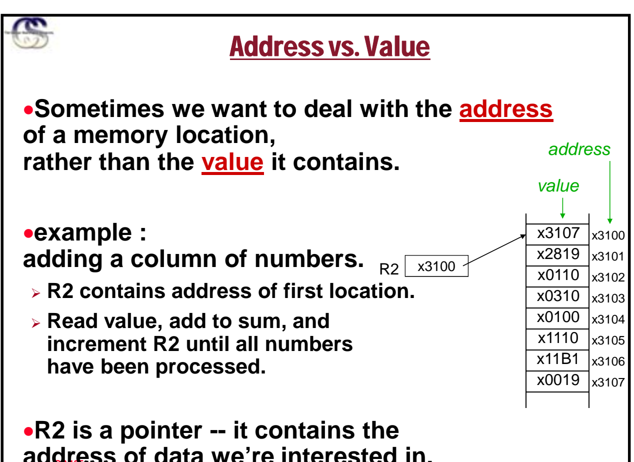
Executing the Swap Function

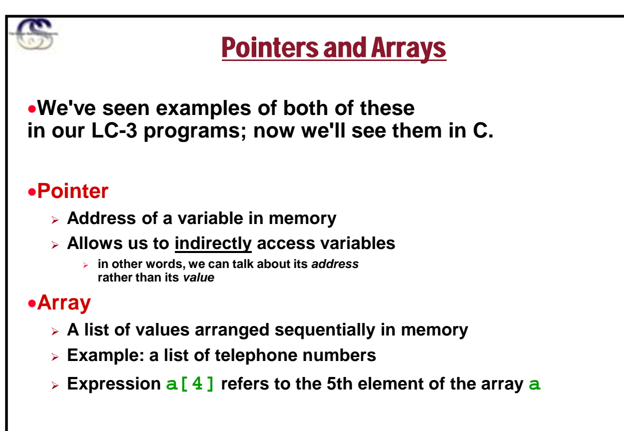


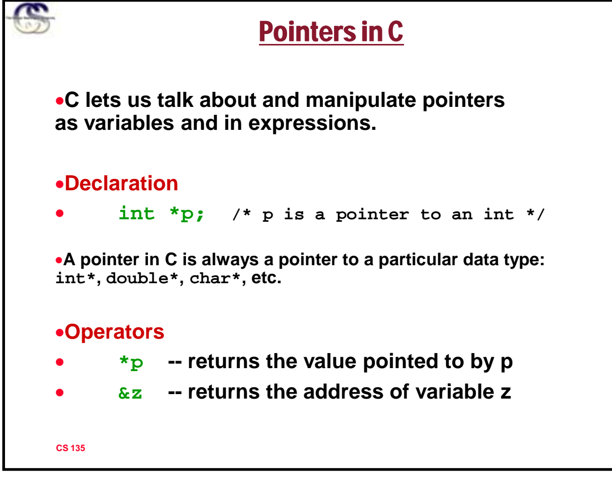
Executing the Swap Function

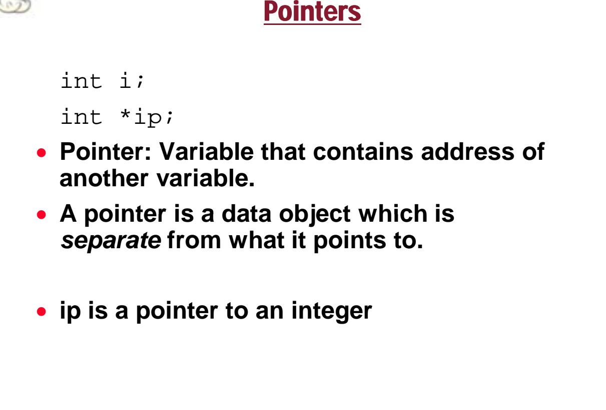


Swap needs addresses of variables outside its own activation record.


Address vs. Value
•Sometimes we want to deal with the address of a memory location, rather than the value it contains.
•example : adding a column of numbers.
R2 contains address of first location.
Read value, add to sum, and increment R2 until all numbers have been processed.
•R2 is a pointer -- it contains the address of data we're interested in.
address
value
R2
x3100
x3107
x2819
x0110
x0310
x0100
x1110
x11B1
x0019
x3100
x3101
x3102
x3103
x3104
x3105
x3106
x3107
CS 135


Pointers and Arrays
•We've seen examples of both of these in our LC-3 programs; now we'll see them in C.
•Pointer
Address of a variable in memory
Allows us to indirectly access variables
in other words, we can talk about its address rather than its value
•Array
A list of values arranged sequentially in memory
Example: a list of telephone numbers
Expression a[4] refers to the 5th element of the array a
CS 135


Pointers in C
•C lets us talk about and manipulate pointers as variables and in expressions.
•Declaration
int *p; /* p is a pointer to an int */
•A pointer in C is always a pointer to a particular data type: int*, double*, char*, etc.
•Operators
*p -- returns the value pointed to by p
&z -- returns the address of variable z
CS 135


Pointers
int i;
int *ip;
Pointer: Variable that contains address of another variable.
A pointer is a data object which is separate from what it points to.
ip is a pointer to an integer
CS 135

Example

- `int i;`
- `int *ptr;`
- `i = 4;`
- `ptr = &i;`
- `*ptr = *ptr + 1;`

store the value 4 into the memory location associated with i

store the address of i into the memory location associated with ptr

read the contents of memory at the address stored in ptr

store the result into memory at the address stored in ptr

Example: LC-3 Code

- *; i is 1st local (offset 0), ptr is 2nd (offset -1)*
- ; i = 4;

```
        AND   R0, R0, #0    ; clear R0
        ADD   R0, R0, #4    ; put 4 in R0
        STR   R0, R5, #0    ; store in i
; ptr = &i;
        ADD   R0, R5, #0    ; R0 = R5 + 0 (addr of i)
        STR   R0, R5, #-1   ; store in ptr
; *ptr = *ptr + 1;
        LDR   R0, R5, #-1   ; R0 = ptr
        LDR   R1, R0, #0    ; load contents (*ptr)
        ADD   R1, R1, #1    ; add one
        STR   R1, R0, #0    ; store result where R0 points
```


Pointers as Arguments

- Passing a pointer into a function allows the function to read/change memory outside its activation record.

```
void NewSwap(int *firstVal, int
*secondVal)
{
  int tempVal = *firstVal;
  *firstVal = *secondVal;
  *secondVal = tempVal;
}
```

Arguments are integer pointers. Caller passes addresses of variables that it wants function to change.

Swap

- **a function that will swap two integers**
- **Last try:**

```
void swap(int *a, int *b)
{
   int t;
   t = *a;
   *a = *b;
   *b = t;
}
```

Now it works...

- **We call it like this**

```
int x = 42;
int y = 84;
swap(&x, &y);
```

CS 135

Trace

```
int x = 42;
int y = 84;
swap(&x, &y);


void swap(int *a, int *b)
{
   int t;
   t = *a;
   *a = *b;
   *b = t;
}
```

y	84
x	42
stack	

Stack Frame for main

CS 135

Trace

```
int x = 42;
int y = 84;
swap(&x, &y);


void swap(int *a, int *b)
{
   int t;
   t = *a;
   *a = *b;
   *b = t;
}
```

t	
b	•
a	•
y	84
x	42
stack	

Stack Frame for swap

CS 135

Trace

```
int x = 42;
int y = 84;
swap(&x, &y);


void swap(int *a, int *b)
{
   int t;
   t = *a;
   *a = *b;
   *b = t;
}
```

t	42
b	•
a	•
y	84
x	42
stack	

Stack Frame for swap

CS 135

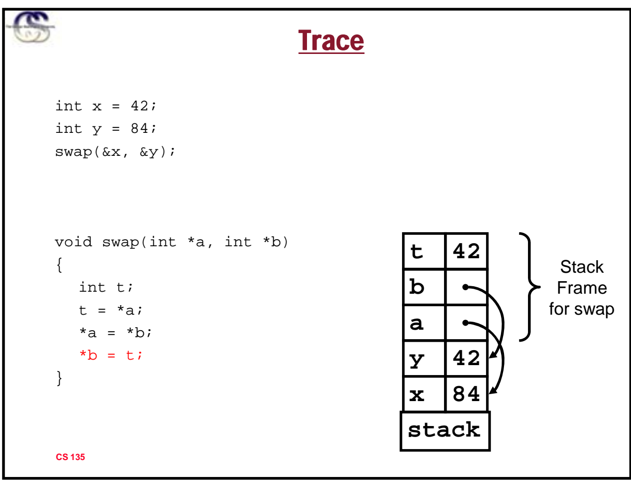
Trace

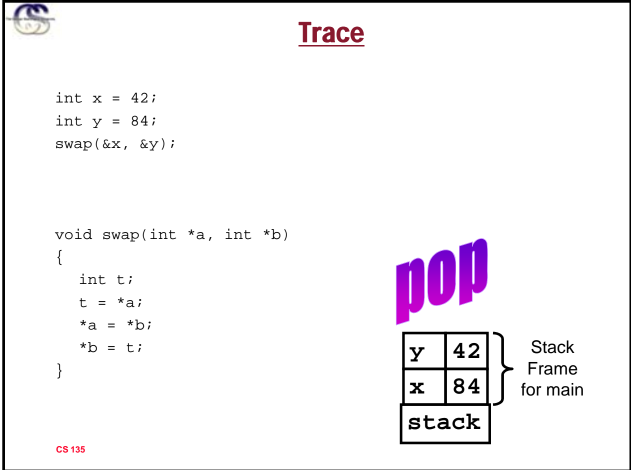
```
int x = 42;
int y = 84;
swap(&x, &y);


void swap(int *a, int *b)
{
   int t;
   t = *a;
   *a = *b;
   *b = t;
}
```

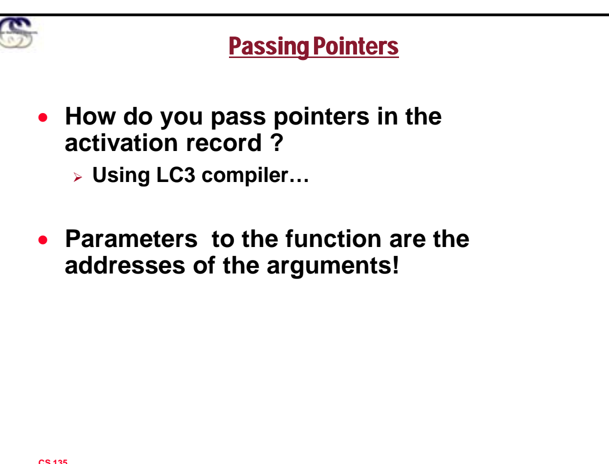
t	42
b	•
a	•
y	84
x	84
stack	

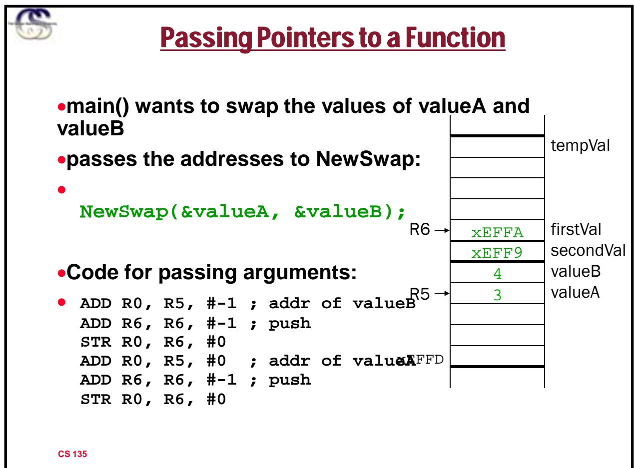
Stack Frame for swap

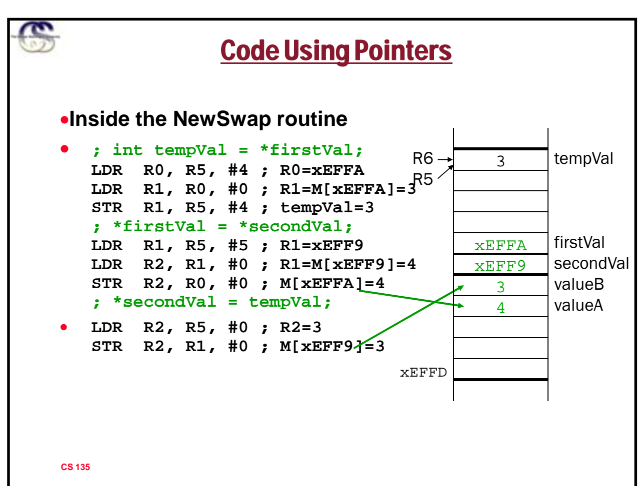
CS 135

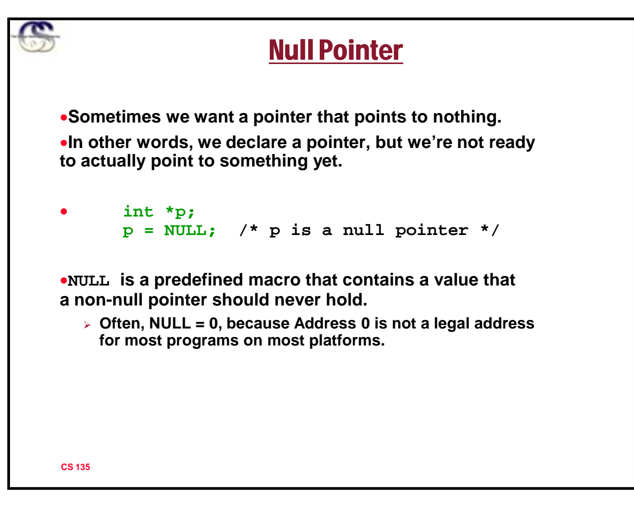


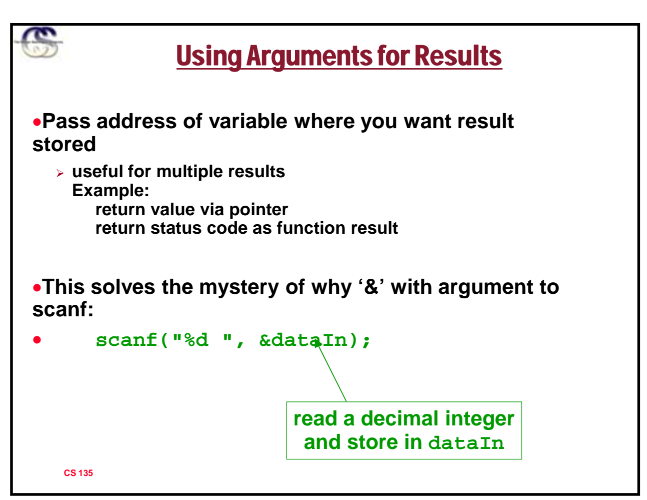


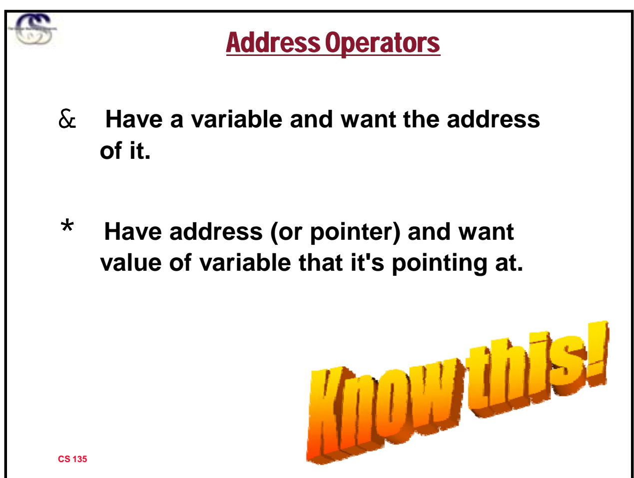


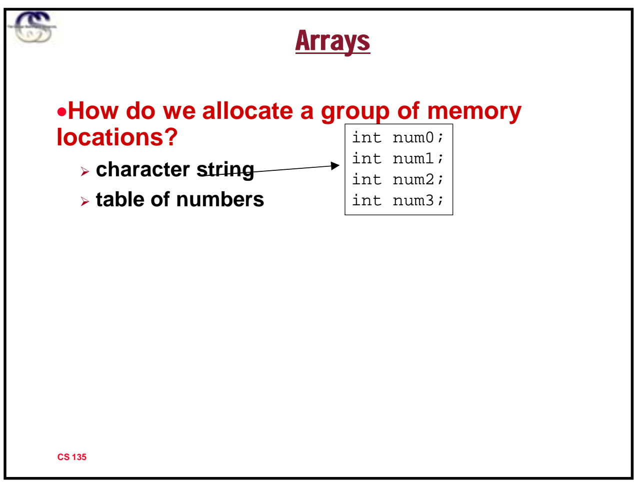

Passing Pointers to a Function
•main() wants to swap the values of valueA and valueB
•passes the addresses to NewSwap:
NewSwap(&valueA, &valueB);
•Code for passing arguments:
ADD R0, R5, #-1 ; addr of valueB
ADD R6, R6, #-1 ; push
STR R0, R6, #0
ADD R0, R5, #0 ; addr of valueA
ADD R6, R6, #-1 ; push
STR R0, R6, #0
tempVal
R6
xEFFA
firstVal
xEFF9
secondVal
4
valueB
R5
3
valueA
xEFFD
CS 135

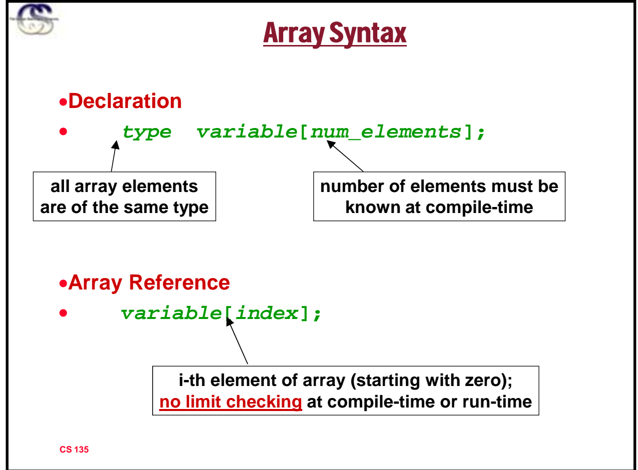

Code Using Pointers
•Inside the NewSwap routine
; int tempVal = *firstVal;
LDR R0, R5, #4 ; R0=xEFFA
LDR R1, R0, #0 ; R1=M[xEFFA]=3
STR R1, R5, #4 ; tempVal=3
; *firstVal = *secondVal;
LDR R1, R5, #5 ; R1=xEFF9
LDR R2, R1, #0 ; R1=M[xEFF9]=4
STR R2, R0, #0 ; M[xEFFA]=4
; *secondVal = tempVal;
LDR R2, R5, #0 ; R2=3
STR R2, R1, #0 ; M[xEFF9]=3
R6
R5
3
tempVal
xEFFA
firstVal
xEFF9
secondVal
3
valueB
4
valueA
xEFFD
CS 135

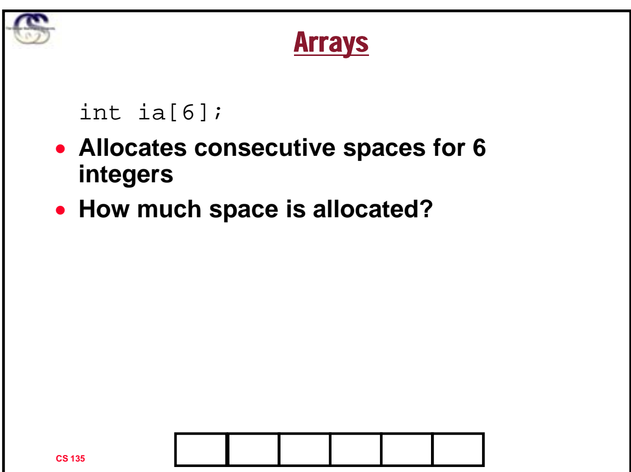

Null Pointer
•Sometimes we want a pointer that points to nothing.
•In other words, we declare a pointer, but we're not ready to actually point to something yet.
int *p;
p = NULL; /* p is a null pointer */
•NULL is a predefined macro that contains a value that a non-null pointer should never hold.
Often, NULL = 0, because Address 0 is not a legal address for most programs on most platforms.
CS 135

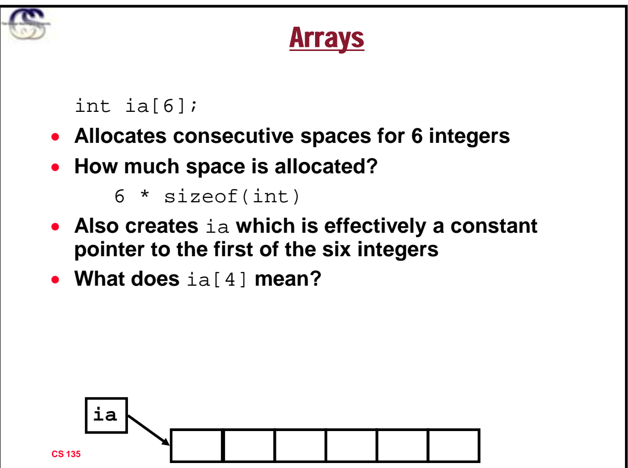

Using Arguments for Results
•Pass address of variable where you want result stored
useful for multiple results
Example:
return value via pointer
return status code as function result
•This solves the mystery of why '&' with argument to scanf:
scanf("%d ", &dataIn);
read a decimal integer and store in dataIn
CS 135

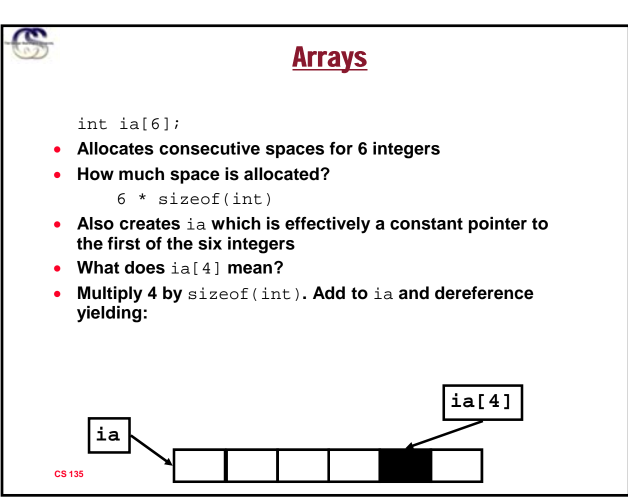

Address Operators
& Have a variable and want the address of it.
* Have address (or pointer) and want value of variable that it's pointing at.
Know this!
CS 135

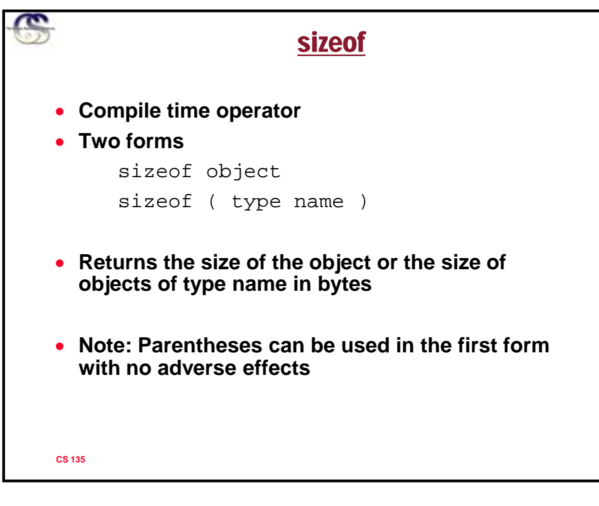

Arrays
•How do we allocate a group of memory locations?
character string
table of numbers
int num0;
int num1;
int num2;
int num3;
CS 135

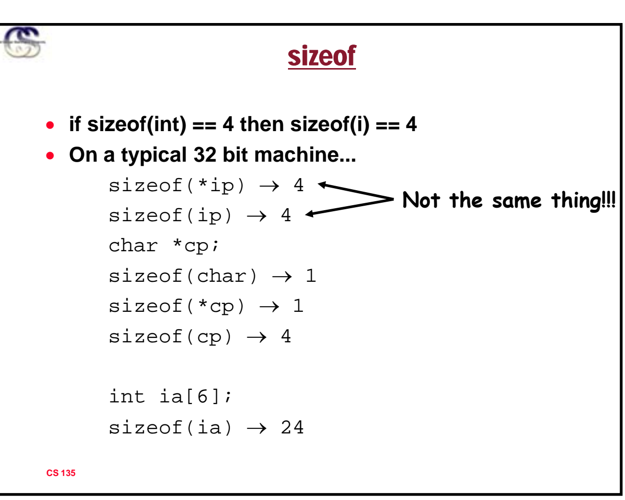

Array Syntax
•Declaration
type variable[num_elements];
all array elements are of the same type
number of elements must be known at compile-time
•Array Reference
variable[index];
i-th element of array (starting with zero);
no limit checking at compile-time or run-time
CS 135

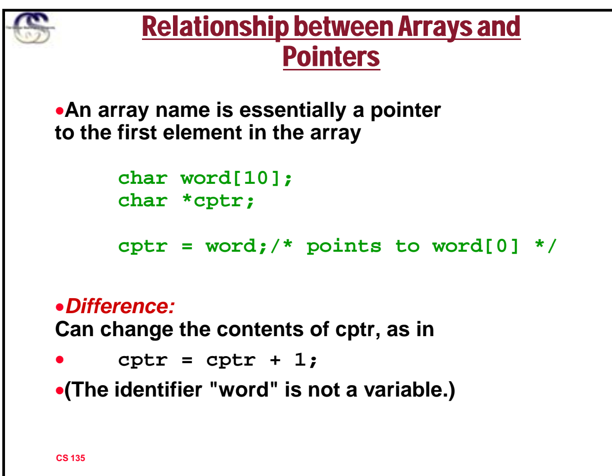

Arrays
int ia[6];
Allocates consecutive spaces for 6 integers
How much space is allocated?
CS 135

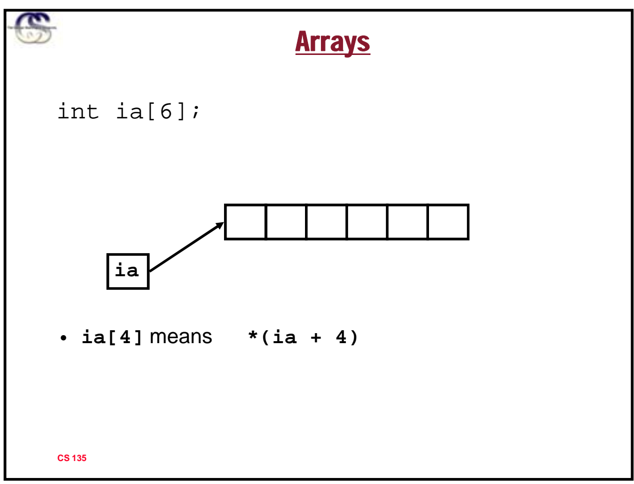

Arrays
int ia[6];
Allocates consecutive spaces for 6 integers
How much space is allocated?
6 * sizeof(int)
Also creates ia which is effectively a constant pointer to the first of the six integers
What does ia[4] mean?
ia
CS 135

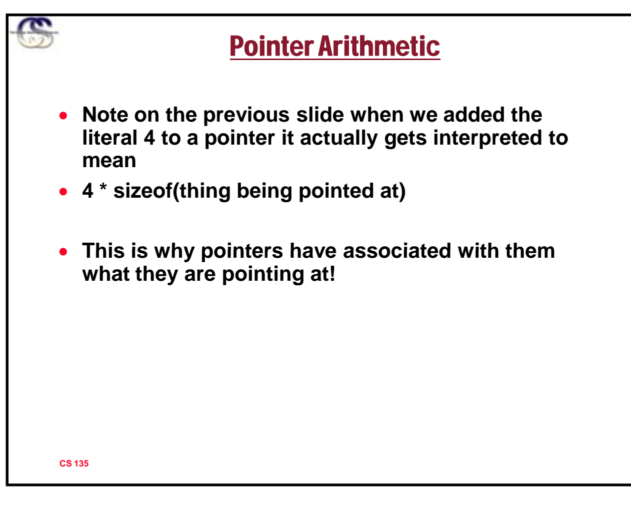

Arrays
int ia[6];
Allocates consecutive spaces for 6 integers
How much space is allocated?
6 * sizeof(int)
Also creates ia which is effectively a constant pointer to the first of the six integers
What does ia[4] mean?
Multiply 4 by sizeof(int). Add to ia and dereference yielding:
ia[4]
ia
CS 135

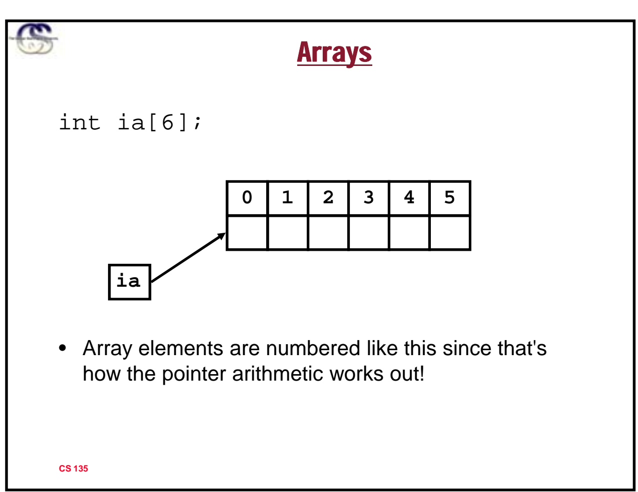

sizeof
Compile time operator
Two forms
sizeof object
sizeof (type name)
Returns the size of the object or the size of objects of type name in bytes
Note: Parentheses can be used in the first form with no adverse effects
CS 135

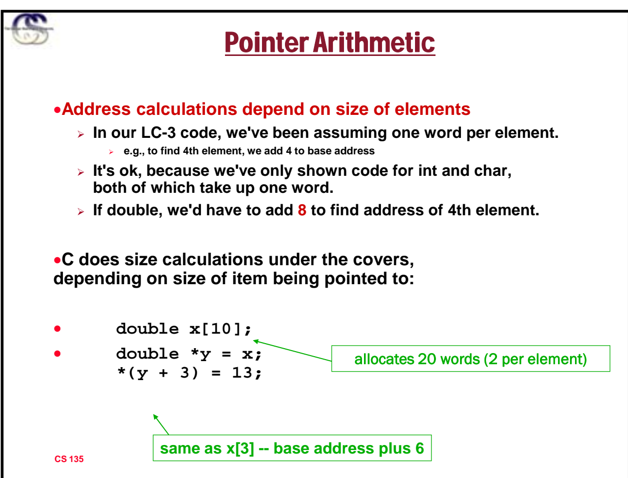

sizeof
if sizeof(int) == 4 then sizeof(i) == 4
On a typical 32 bit machine...
sizeof(*ip) → 4
sizeof(ip) → 4
Not the same thing!!!
char *cp;
sizeof(char) → 1
sizeof(*cp) → 1
sizeof(cp) → 4
int ia[6];
sizeof(ia) → 24
CS 135

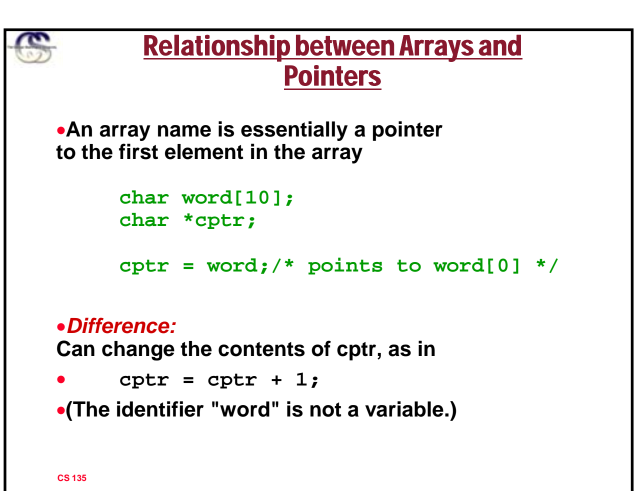

Relationship between Arrays and Pointers
•An array name is essentially a pointer to the first element in the array
char word[10];
char *cptr;
cptr = word;/* points to word[0] */
•Difference:
Can change the contents of cptr, as in
• cptr = cptr + 1;
•(The identifier "word" is not a variable.)
CS 135

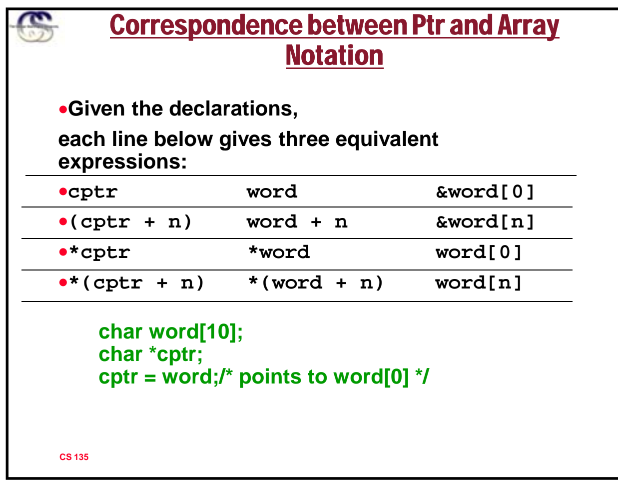

Arrays
int ia[6];
ia
• ia[4] means *(ia + 4)
CS 135


Pointer Arithmetic
• Note on the previous slide when we added the literal 4 to a pointer it actually gets interpreted to mean
• 4 * sizeof(thing being pointed at)
• This is why pointers have associated with them what they are pointing at!
CS 135


Arrays
int ia[6];
0 1 2 3 4 5
ia
• Array elements are numbered like this since that's how the pointer arithmetic works out!
CS 135







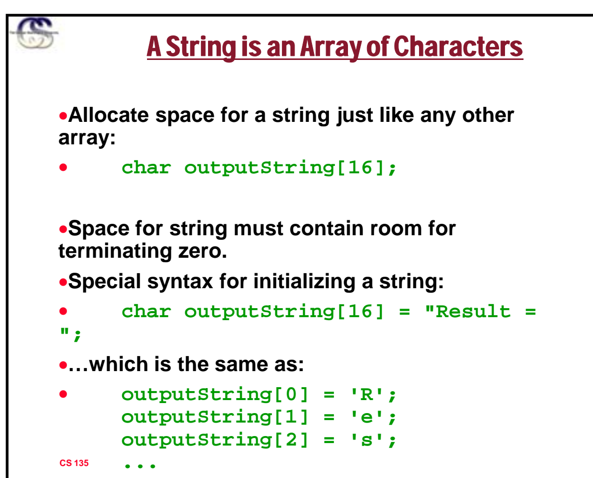
Passing Arrays as Arguments

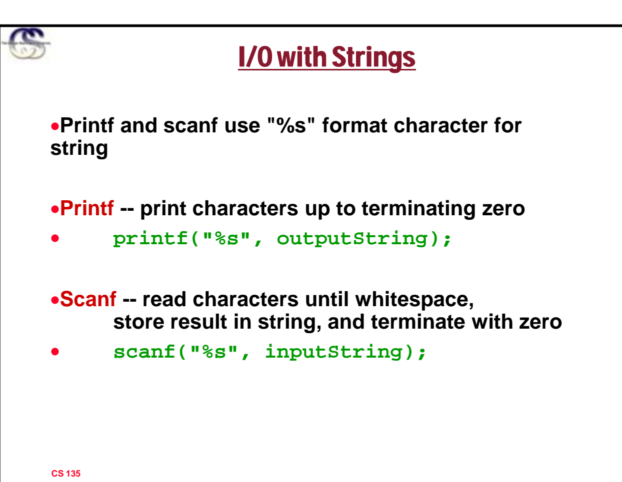
- C passes arrays by reference
  - the address of the array (i.e., of the first element) is written to the function's activation record
  - otherwise, would have to copy each element

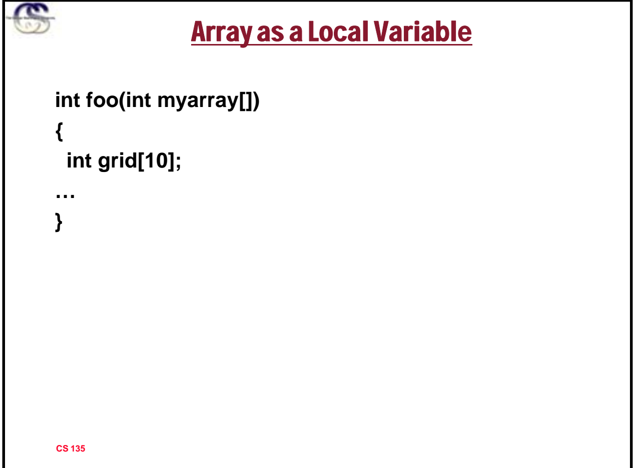
```
•main() {
•  int numbers[MAX_NUMS];
•  …
   mean = Average(numbers);
•  …
}
•int Average(int inputValues[MAX_NUMS]) {
•  …
•  for (index = 0; index < MAX_NUMS; index++)
       sum = sum + indexValues[index];
   return (sum / MAX_NUMS);
•}
```

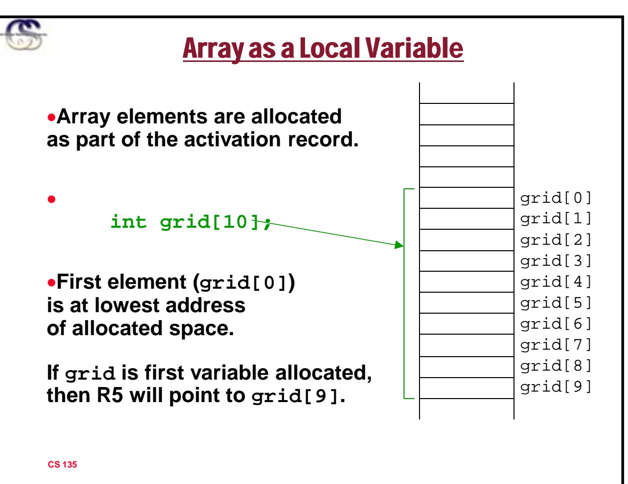
This must be a constant, e.g.,
`#define MAX_NUMS 10`

CS 135


A String is an Array of Characters
•Allocate space for a string just like any other array:
• char outputString[16];
•Space for string must contain room for terminating zero.
•Special syntax for initializing a string:
• char outputString[16] = "Result = ";
•...which is the same as:
• outputString[0] = 'R';
outputString[1] = 'e';
outputString[2] = 's';
...
CS 135


I/O with Strings
•Printf and scanf use "%s" format character for string
•Printf -- print characters up to terminating zero
• printf("%s", outputString);
•Scanf -- read characters until whitespace, store result in string, and terminate with zero
• scanf("%s", inputString);
CS 135


Array as a Local Variable
int foo(int myarray[])
{
int grid[10];
…
}
CS 135

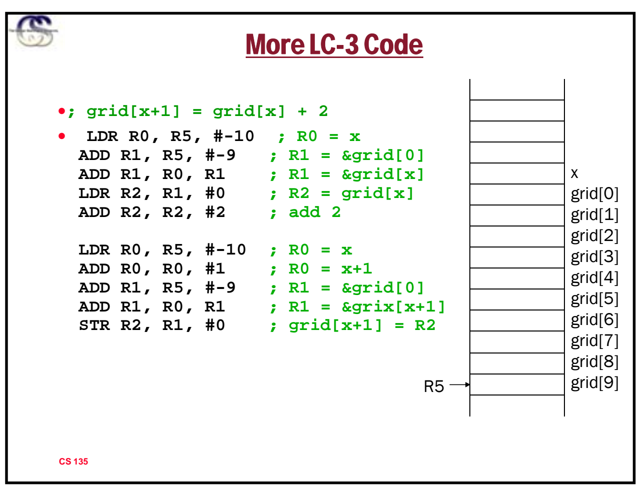

Array as a Local Variable
•Array elements are allocated as part of the activation record.
• int grid[10];
•First element (grid[0]) is at lowest address of allocated space.
If grid is first variable allocated, then R5 will point to grid[9].
grid[0]
grid[1]
grid[2]
grid[3]
grid[4]
grid[5]
grid[6]
grid[7]
grid[8]
grid[9]
CS 135

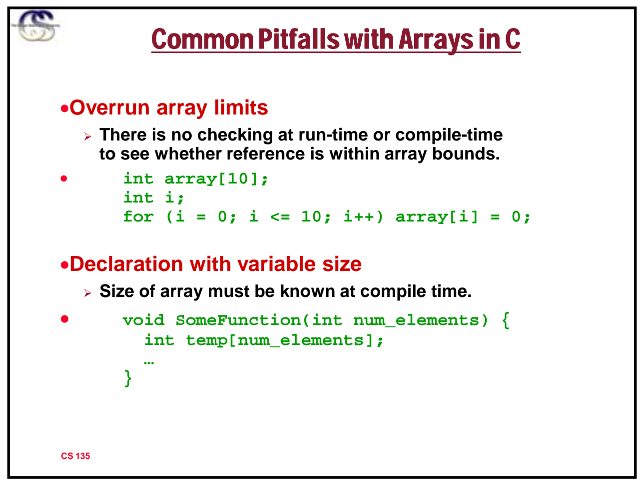
LC-3 Code for Array References

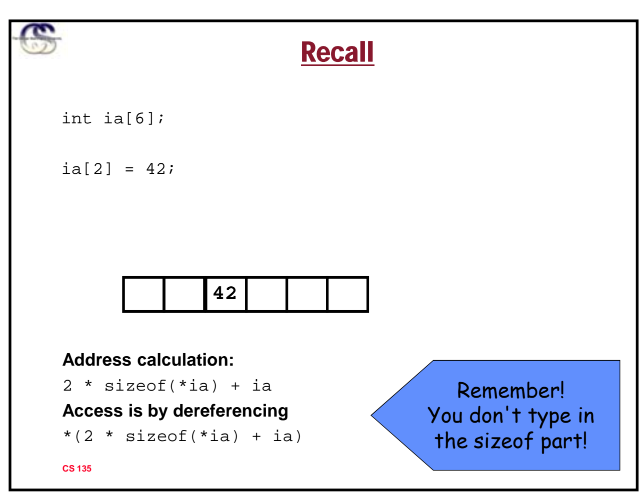
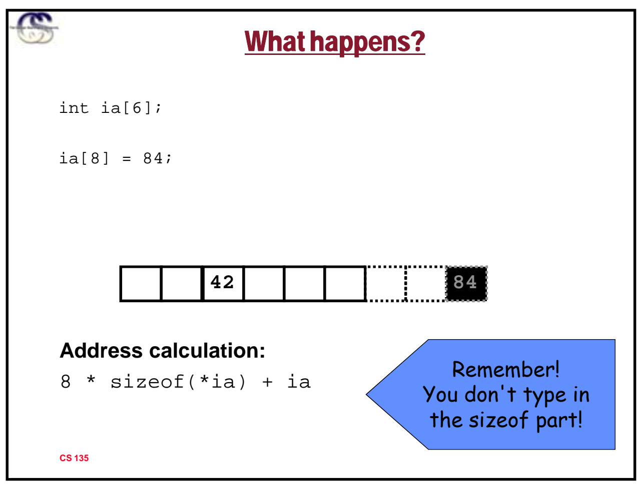
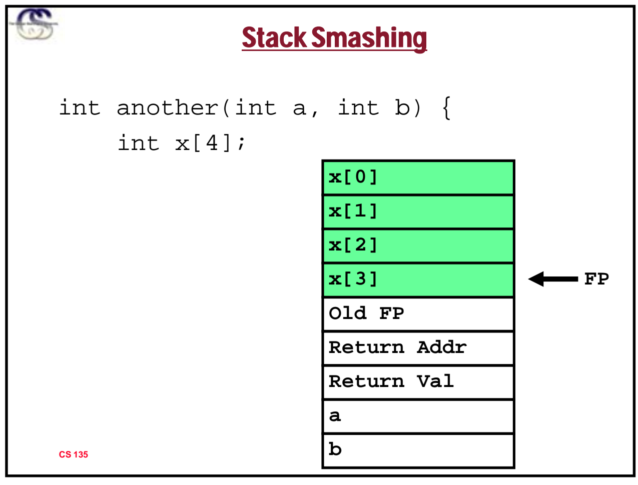
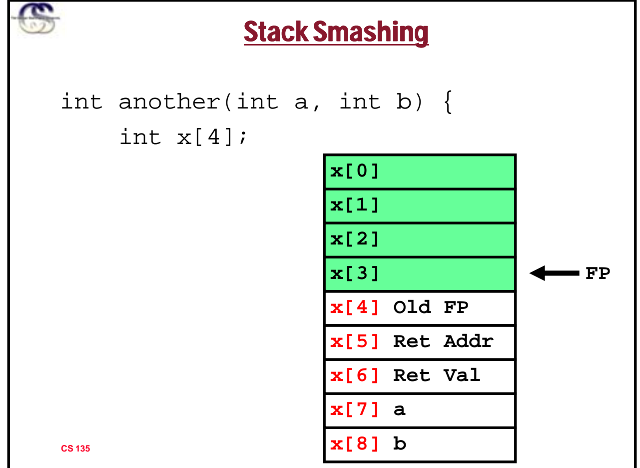
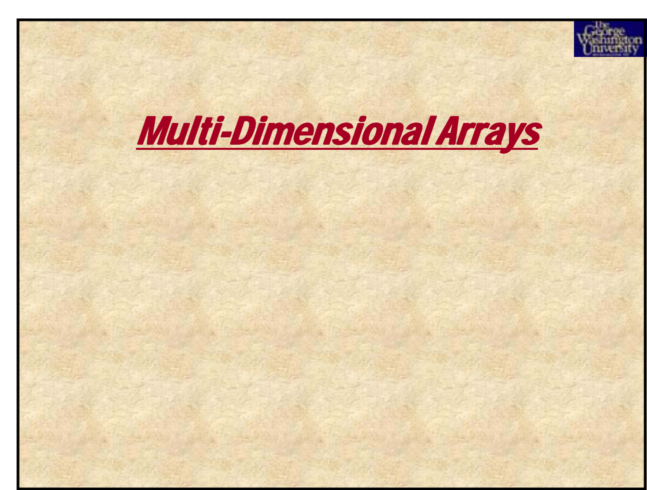
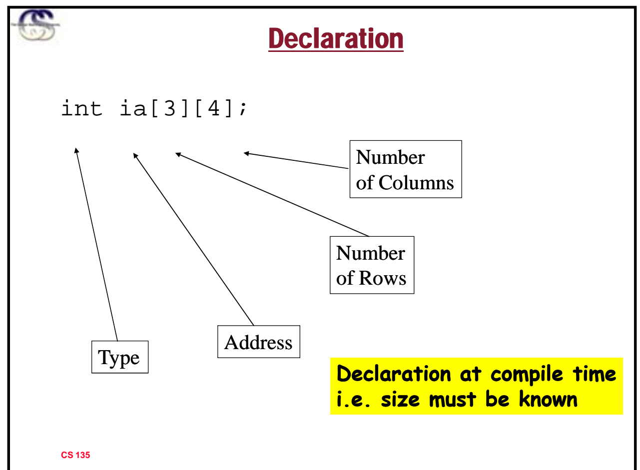
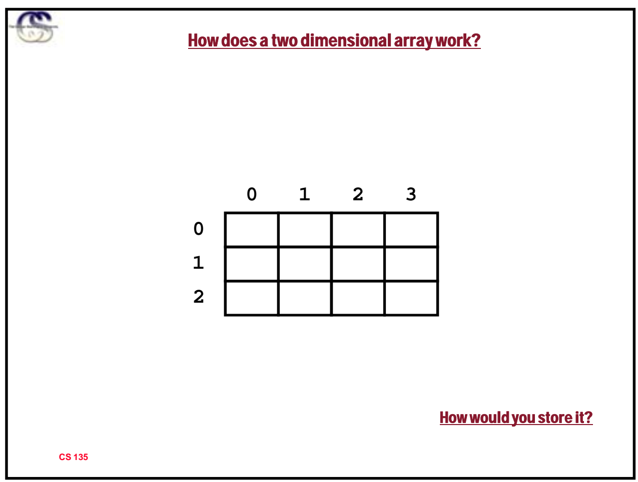
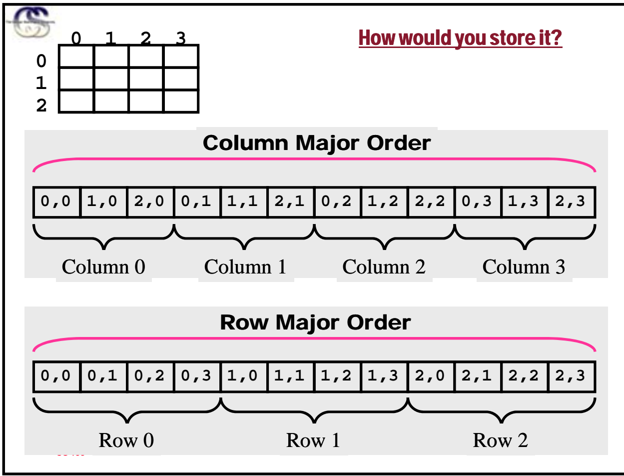
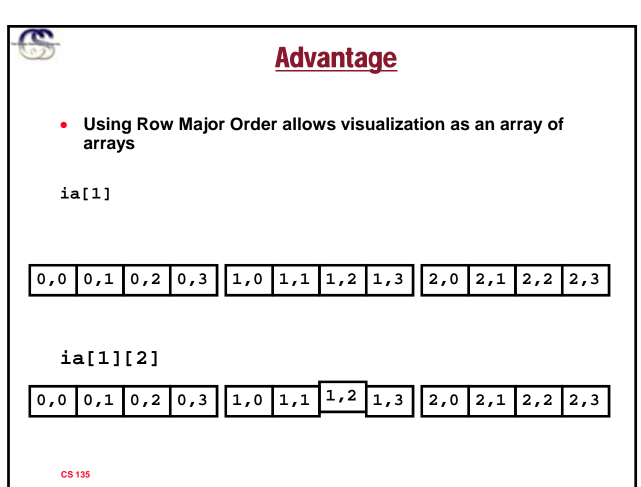
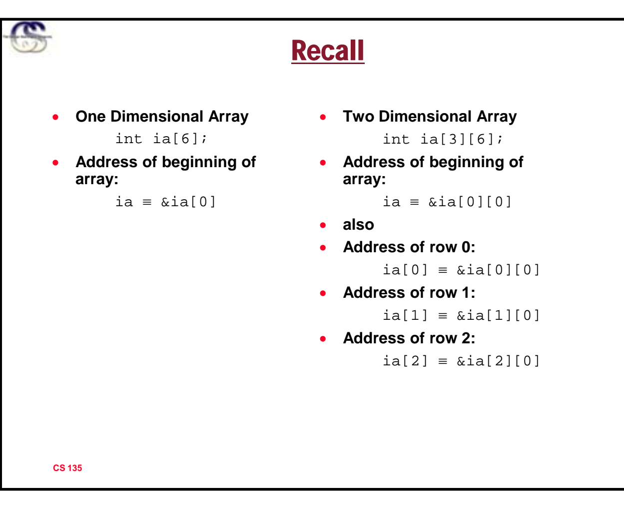
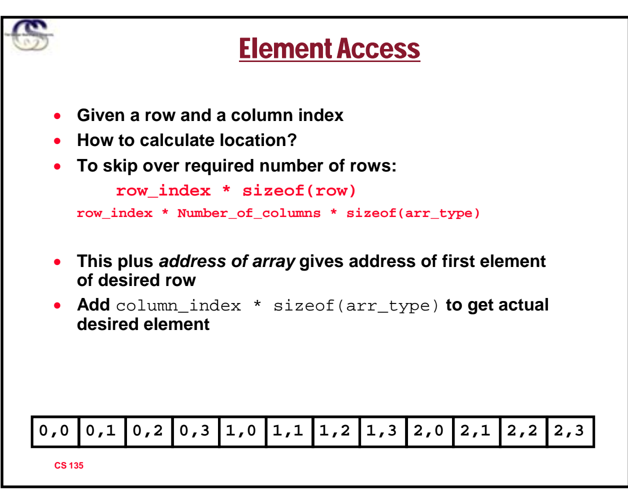
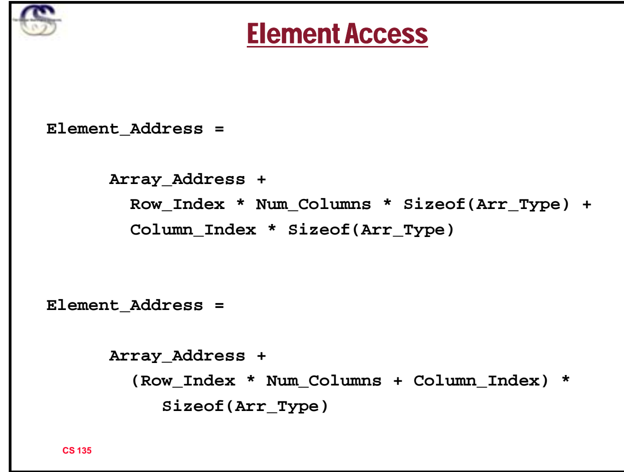
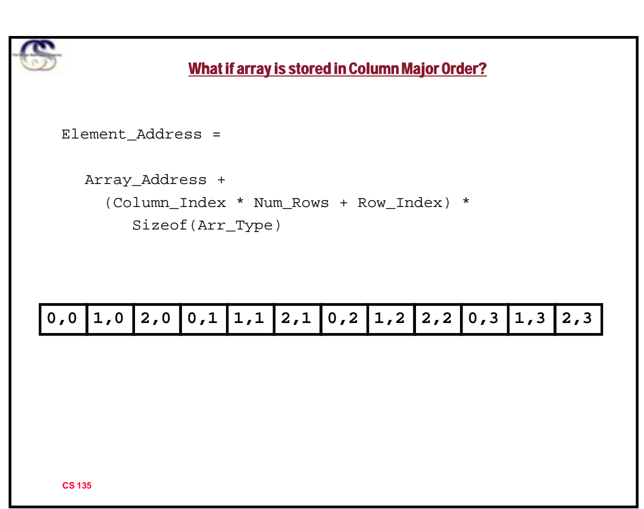
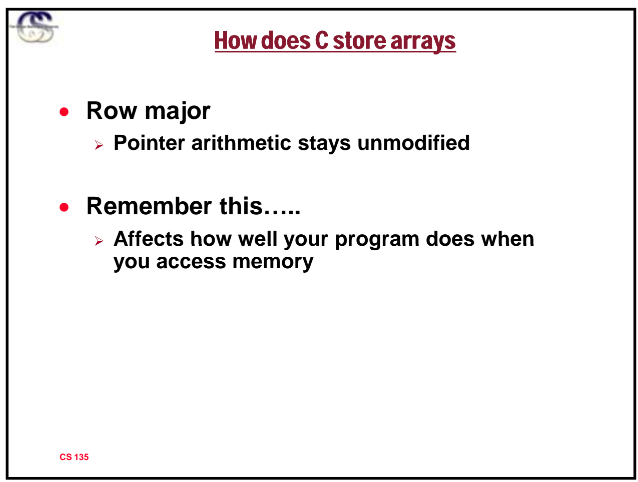
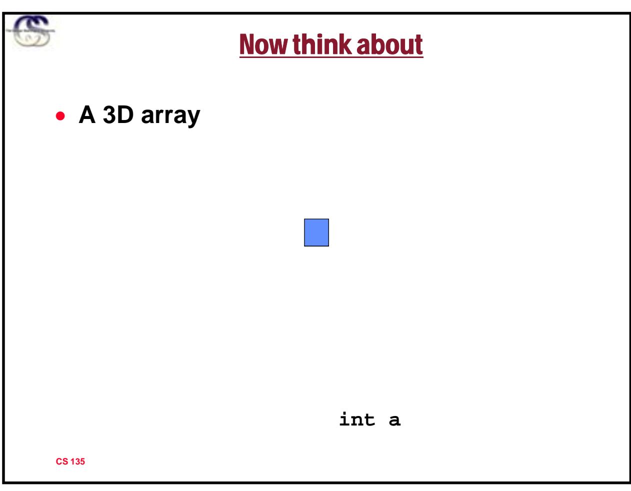
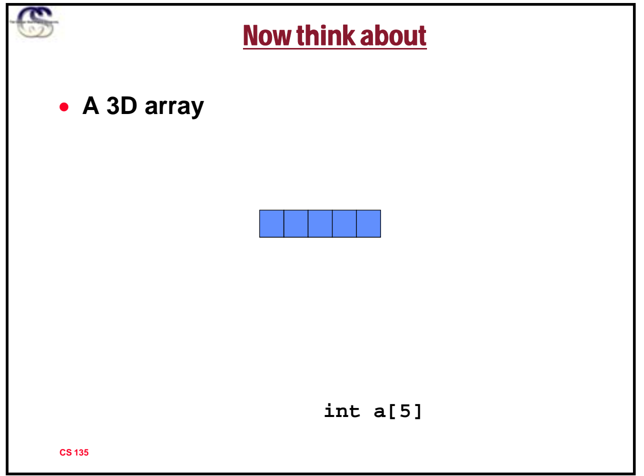
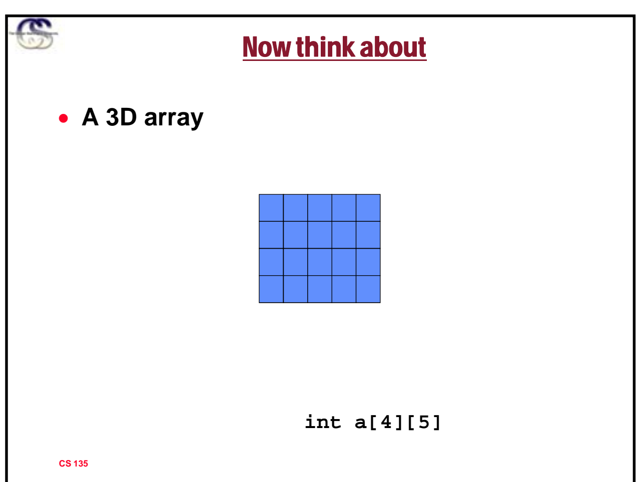
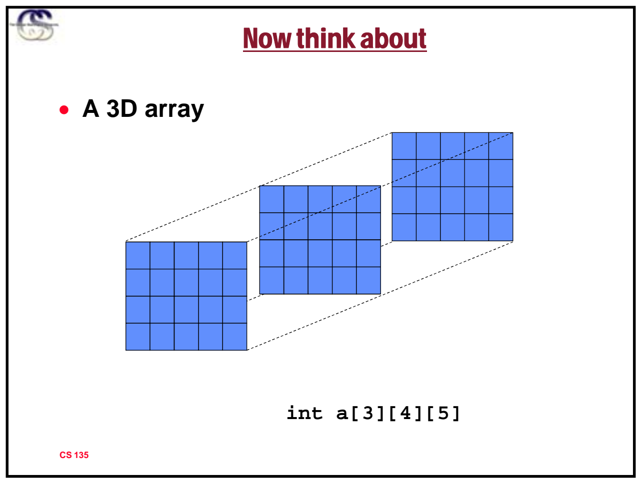
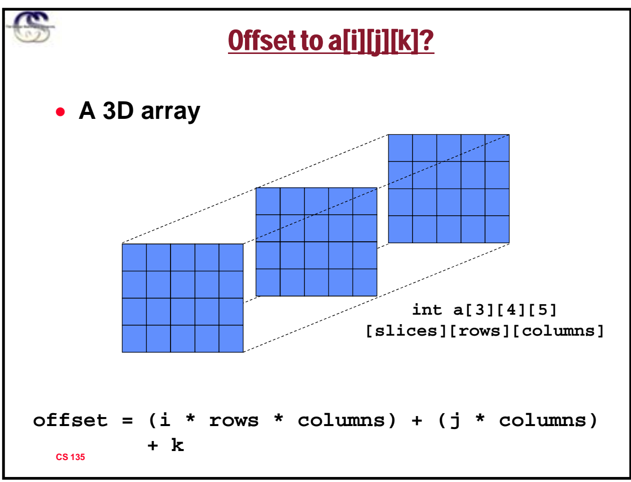
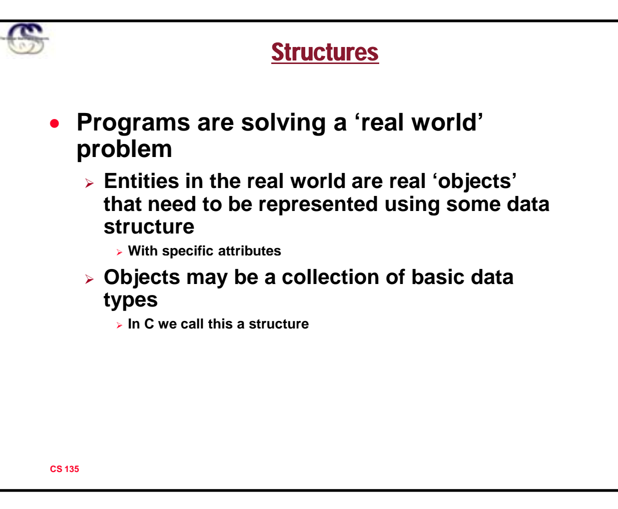
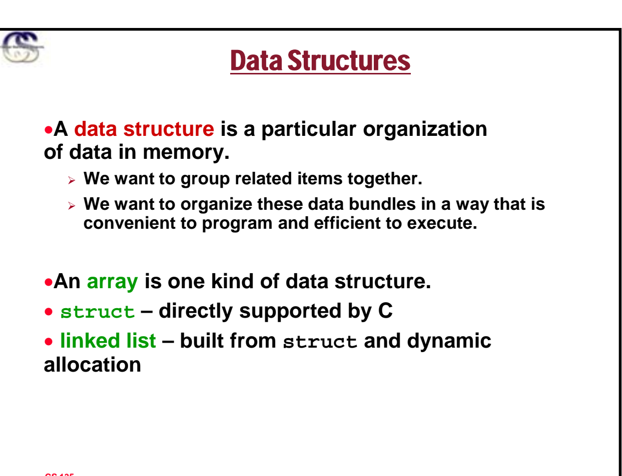
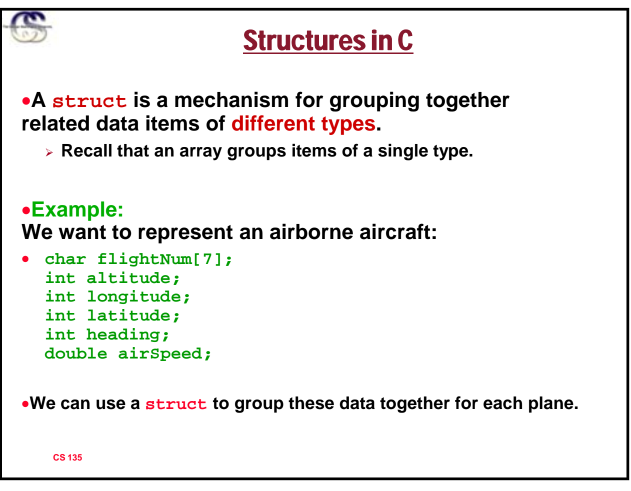
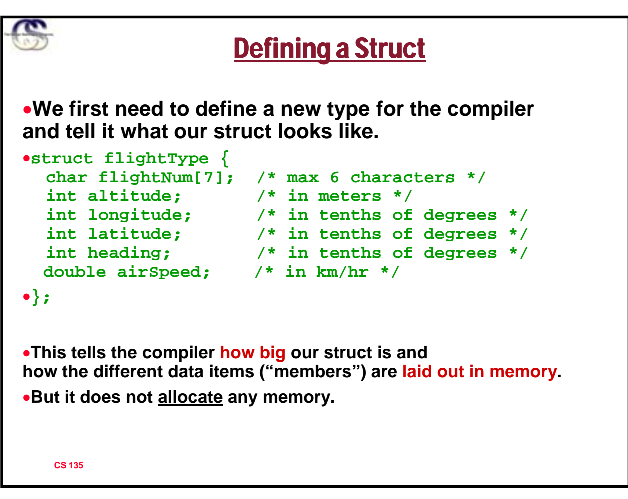
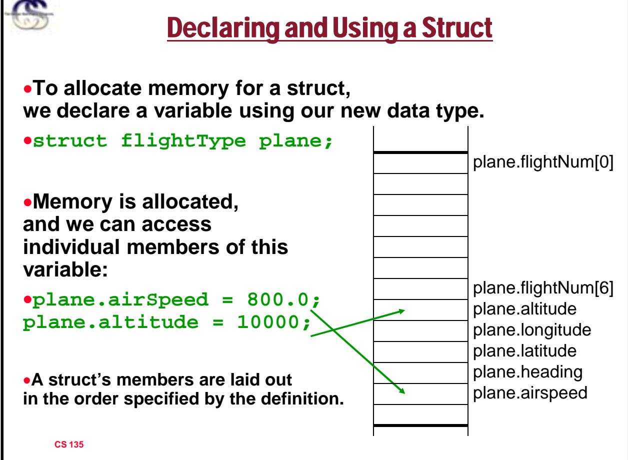
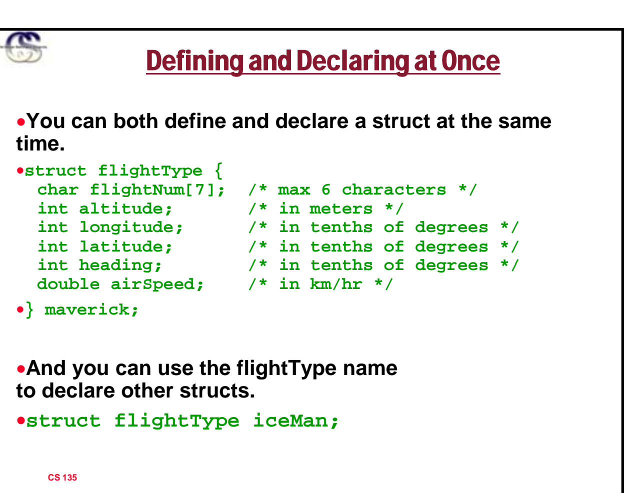
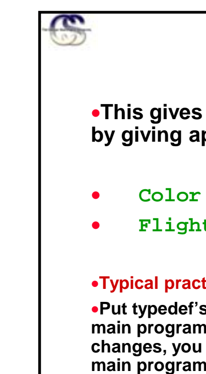
- ; x = grid[3] + 1
- ```
  ADD R0, R5, #-9   ; R0 = &grid[0]
  LDR R1, R0, #3    ; R1 = grid[3]
  ADD R1, R1, #1    ; plus 1
  STR R1, R5, #-10  ; x = R1
  ```
- ; grid[6] = 5;
  ```
  AND R0, R0, #0
  ADD R0, R0, #5   ; R0 = 5
  ADD R1, R5, #-9  ; R1 = &grid[0]
  STR R0, R1, #6   ; grid[6] = R0
  ```

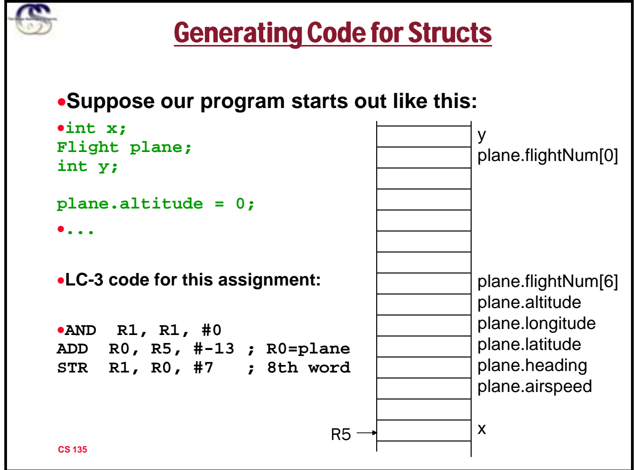
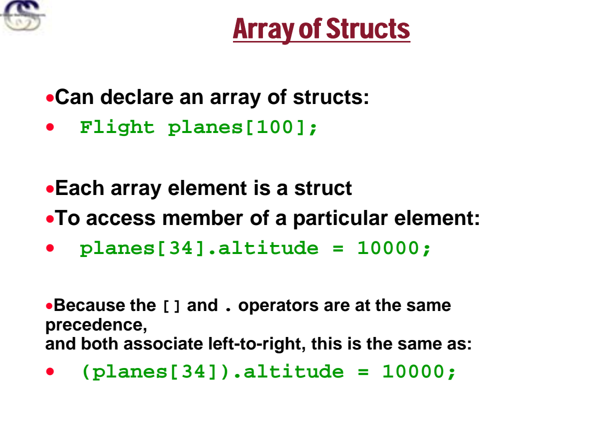
x
grid[0]
grid[1]
grid[2]
grid[3]
grid[4]
grid[5]
grid[6]
grid[7]
grid[8]
grid[9]
R5

CS 135






  ```


What happens?
int ia[6];
ia[8] = 84;
42
84
Address calculation:
8 * sizeof(*ia) + ia
Remember!
You don't type in
the sizeof part!
CS 135


Stack Smashing
int another(int a, int b) {
int x[4];
x[0]
x[1]
x[2]
x[3]
FP
Old FP
Return Addr
Return Val
a
b
CS 135


Stack Smashing
int another(int a, int b) {
int x[4];
x[0]
x[1]
x[2]
x[3]
FP
x[4] Old FP
x[5] Ret Addr
x[6] Ret Val
x[7] a
x[8] b
CS 135


Multi-Dimensional Arrays


Declaration
int ia[3][4];
Number of Columns
Number of Rows
Address
Type
Declaration at compile time i.e. size must be known
CS 135


How does a two dimensional array work?
0 1 2 3
0
1
2
How would you store it?
CS 135


How would you store it?
0 1 2 3
0
1
2
Column Major Order
0,0 1,0 2,0 0,1 1,1 2,1 0,2 1,2 2,2 0,3 1,3 2,3
Column 0 Column 1 Column 2 Column 3
Row Major Order
0,0 0,1 0,2 0,3 1,0 1,1 1,2 1,3 2,0 2,1 2,2 2,3
Row 0 Row 1 Row 2


Advantage
Using Row Major Order allows visualization as an array of arrays
ia[1]
0,0 0,1 0,2 0,3 1,0 1,1 1,2 1,3 2,0 2,1 2,2 2,3
ia[1][2]
0,0 0,1 0,2 0,3 1,0 1,1 1,2 1,3 2,0 2,1 2,2 2,3
CS 135


Recall
One Dimensional Array
int ia[6];
Address of beginning of array:
ia ≡ &ia[0]
Two Dimensional Array
int ia[3][6];
Address of beginning of array:
ia ≡ &ia[0][0]
also
Address of row 0:
ia[0] ≡ &ia[0][0]
Address of row 1:
ia[1] ≡ &ia[1][0]
Address of row 2:
ia[2] ≡ &ia[2][0]
CS 135


Element Access
Given a row and a column index
How to calculate location?
To skip over required number of rows:
row_index * sizeof(row)
row_index * Number_of_columns * sizeof(arr_type)
This plus address of array gives address of first element of desired row
Add column_index * sizeof(arr_type) to get actual desired element
0,0 0,1 0,2 0,3 1,0 1,1 1,2 1,3 2,0 2,1 2,2 2,3
CS 135


Element Access
Element_Address =
Array_Address +
Row_Index * Num_Columns * Sizeof(Arr_Type) +
Column_Index * Sizeof(Arr_Type)
Element_Address =
Array_Address +
(Row_Index * Num_Columns + Column_Index) *
Sizeof(Arr_Type)
CS 135


What if array is stored in Column Major Order?
Element_Address =
Array_Address +
(Column_Index * Num_Rows + Row_Index) *
Sizeof(Arr_Type)
0,0 1,0 2,0 0,1 1,1 2,1 0,2 1,2 2,2 0,3 1,3 2,3
CS 135


How does C store arrays
Row major
Pointer arithmetic stays unmodified
Remember this…..
Affects how well your program does when you access memory
CS 135


Now think about
A 3D array
int a
CS 135


Now think about
A 3D array
int a[5]
CS 135


Now think about
A 3D array
int a[4][5]
CS 135


Now think about
A 3D array
int a[3][4][5]
CS 135


Offset to a[i][j][k]?
A 3D array
int a[3][4][5]
[slices][rows][columns]
offset = (i * rows * columns) + (j * columns) + k
CS 135


Structures
Programs are solving a 'real world' problem
Entities in the real world are real 'objects' that need to be represented using some data structure
With specific attributes
Objects may be a collection of basic data types
In C we call this a structure
CS 135


Data Structures
A data structure is a particular organization of data in memory.
We want to group related items together.
We want to organize these data bundles in a way that is convenient to program and efficient to execute.
An array is one kind of data structure.
struct – directly supported by C
linked list – built from struct and dynamic allocation
CS 135


Structures in C
•A struct is a mechanism for grouping together related data items of different types.
Recall that an array groups items of a single type.
•Example:
We want to represent an airborne aircraft:
• char flightNum[7];
int altitude;
int longitude;
int latitude;
int heading;
double airSpeed;
•We can use a struct to group these data together for each plane.
CS 135


Defining a Struct
•We first need to define a new type for the compiler and tell it what our struct looks like.
•struct flightType {
char flightNum[7]; /* max 6 characters */
int altitude; /* in meters */
int longitude; /* in tenths of degrees */
int latitude; /* in tenths of degrees */
int heading; /* in tenths of degrees */
double airSpeed; /* in km/hr */
•};
•This tells the compiler how big our struct is and how the different data items ("members") are laid out in memory.
•But it does not allocate any memory.
CS 135


Declaring and Using a Struct
•To allocate memory for a struct, we declare a variable using our new data type.
•struct flightType plane;
•Memory is allocated, and we can access individual members of this variable:
•plane.airSpeed = 800.0;
plane.altitude = 10000;
•A struct's members are laid out in the order specified by the definition.
plane.flightNum[0]
plane.flightNum[6]
plane.altitude
plane.longitude
plane.latitude
plane.heading
plane.airspeed
CS 135


Defining and Declaring at Once
•You can both define and declare a struct at the same time.
•struct flightType {
char flightNum[7]; /* max 6 characters */
int altitude; /* in meters */
int longitude; /* in tenths of degrees */
int latitude; /* in tenths of degrees */
int heading; /* in tenths of degrees */
double airSpeed; /* in km/hr */
•} maverick;
•And you can use the flightType name to declare other structs.
•struct flightType iceMan;
CS 135

typedef

- C provides a way to define a data type by giving a new name to a predefined type.
- Syntax:
  - `typedef <type> <name>;`
- Examples:
  - `typedef int Color;`
  - `typedef struct flightType Flight;`
  - ```
    typedef struct ab_type {
      int a;
      double b;
    } ABGroup;
    ```




Using typedef

- This gives us a way to make code more readable by giving application-specific names to types.

```
Color pixels[500];
Flight plane1, plane2;
```

- Typical practice:
- Put typedef's into a header file, and use type names in main program. If the definition of Color/Flight changes, you might not need to change the code in your main program file.
  - Pay attention…..need this in your Project 3,4

Generating Code for Structs

- Suppose our program starts out like this:

```
int x;
Flight plane;
int y;

plane.altitude = 0;
...
```

- LC-3 code for this assignment:

```
AND  R1, R1, #0
ADD  R0, R5, #-13 ; R0=plane
STR  R1, R0, #7   ; 8th word
```



Array of Structs

- Can declare an array of structs:
  - `Flight planes[100];`
- Each array element is a struct
- To access member of a particular element:
  - `planes[34].altitude = 10000;`
- Because the `[]` and `.` operators are at the same precedence, and both associate left-to-right, this is the same as:
  - `(planes[34]).altitude = 10000;`


```

Pointer to Struct

- We can declare and create a pointer to a struct:

```
Flight *planePtr;
planePtr = &planes[34];
```

- To access a member of the struct addressed by Ptr:

```
(*planePtr).altitude = 10000;
```

- Because the . operator has higher precedence than *, this is NOT the same as:

```
*planePtr.altitude = 10000;
```

- C provides special syntax for accessing a struct member through a pointer:

```
planePtr->altitude = 10000;
```

Passing Structs as Arguments

- Unlike an array, a struct is always passed by value into a function.
  - This means the struct members are copied to the function's activation record, and changes inside the function are not reflected in the calling routine's copy.
- Most of the time, you'll want to pass a pointer to a struct.

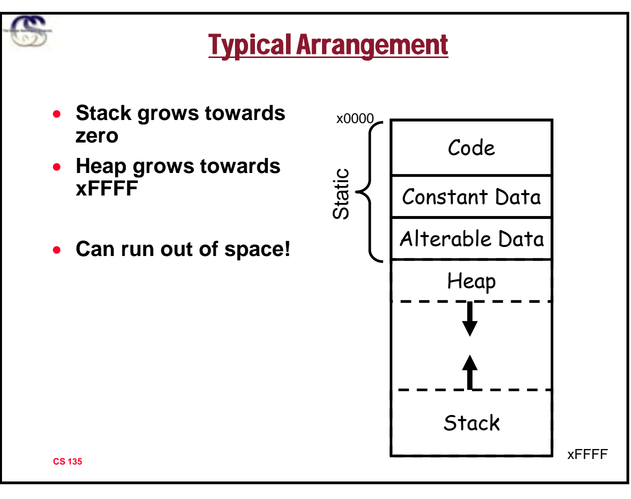
```
int Collide(Flight *planeA, Flight *planeB)
{
  if (planeA->altitude == planeB->altitude) {
     ...
  }
  else
    return 0;
}
```

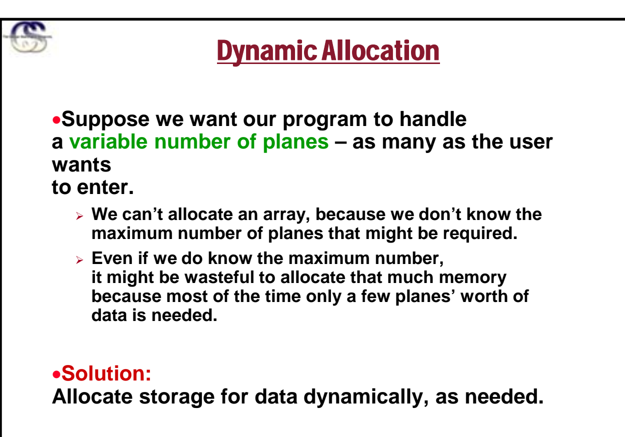
Static vs. Dynamic Allocation

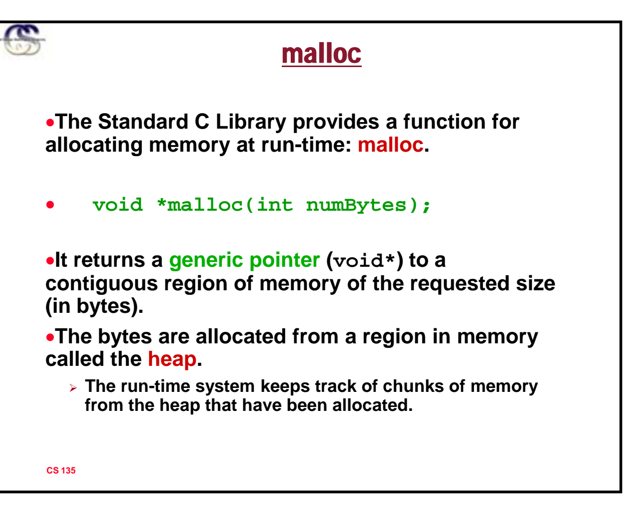
- There are two different ways that multidimensional arrays could be implemented in C.
- Static: When you know the size at compile time
  - A Static implementation which is more efficient in terms of space and probably more efficient in terms of time.
- Dynamic: what if you don't know the size at compile time?
  - More flexible in terms of run time definition but more complicated to understand and build
  - Dynamic data structures
- Need to allocate memory at run-time – malloc
  - Once you are done using this, then release this memory – free
- Next: Dynamic Memory Alloction

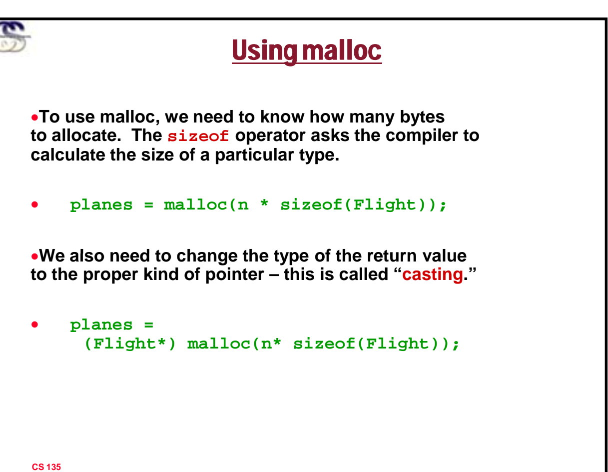
Dynamic Allocation

- Size of all of our data structures have been defined statically
  - int myarray[100] reserves 100 locations
- What if size is only known at run-time ?
  - Guess max size and allocate statically ?
    - int myarray[max_size]
- Dynamic allocation
  - Ask for space at run-time
  - Need run-time support – call system to do this allocation
  - Provide a library call in C for users
- Where do you allocate this space – heap


Typical Arrangement
Stack grows towards zero
Heap grows towards xFFFF
Can run out of space!
x0000
Static
Code
Constant Data
Alterable Data
Heap
Stack
xFFFF
CS 135


Dynamic Allocation
•Suppose we want our program to handle a variable number of planes – as many as the user wants to enter.
We can't allocate an array, because we don't know the maximum number of planes that might be required.
Even if we do know the maximum number, it might be wasteful to allocate that much memory because most of the time only a few planes' worth of data is needed.
•Solution:
Allocate storage for data dynamically, as needed.
CS 135


malloc
•The Standard C Library provides a function for allocating memory at run-time: malloc.
void *malloc(int numBytes);
•It returns a generic pointer (void*) to a contiguous region of memory of the requested size (in bytes).
•The bytes are allocated from a region in memory called the heap.
The run-time system keeps track of chunks of memory from the heap that have been allocated.
CS 135


Using malloc
•To use malloc, we need to know how many bytes to allocate. The sizeof operator asks the compiler to calculate the size of a particular type.
planes = malloc(n * sizeof(Flight));
•We also need to change the type of the return value to the proper kind of pointer – this is called "casting."
planes = (Flight*) malloc(n* sizeof(Flight));
CS 135

Example

```
•int airbornePlanes;
Flight *planes;

printf("How many planes are in the air?");
scanf("%d", &airbornePlanes);

planes =
  (Flight*) malloc(sizeof(Flight) *
airbornePlanes);
if (planes == NULL) {
  printf("Error in allocating the data
array.\n");
  ...
}
planes[0].altitude = ...
```

If allocation fails, malloc returns NULL.

Note: Can use array notation or pointer notation.

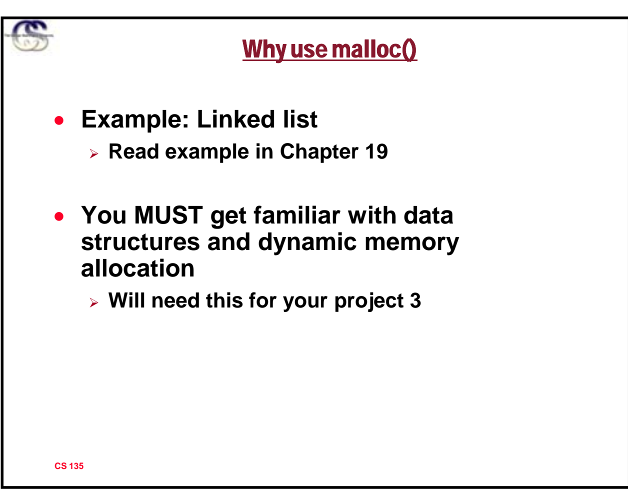
free

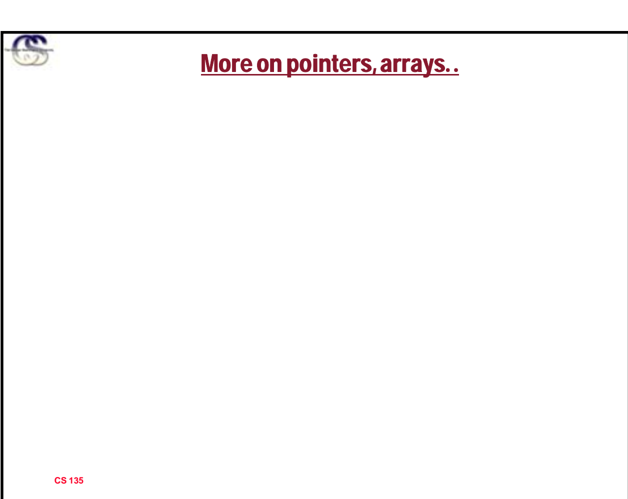
- Once the data is no longer needed, it should be released back into the heap for later use.
- This is done using the free function, passing it the same address that was returned by malloc.

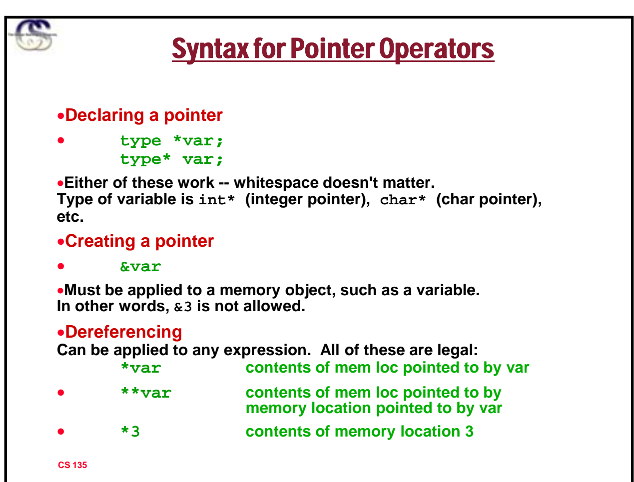
```
void free(void*);
```

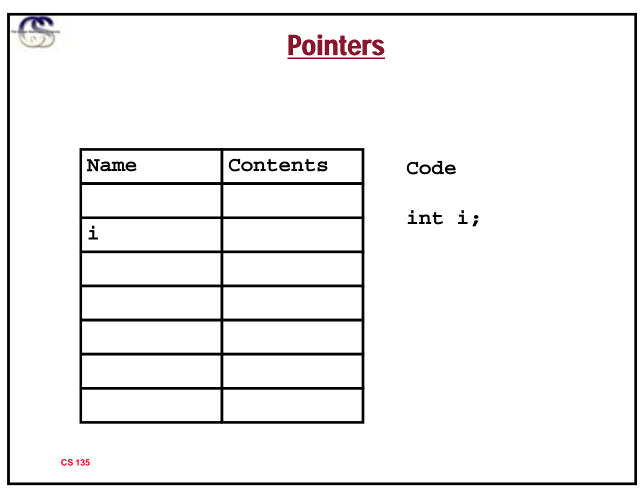
  - `Free(planes[0]);`

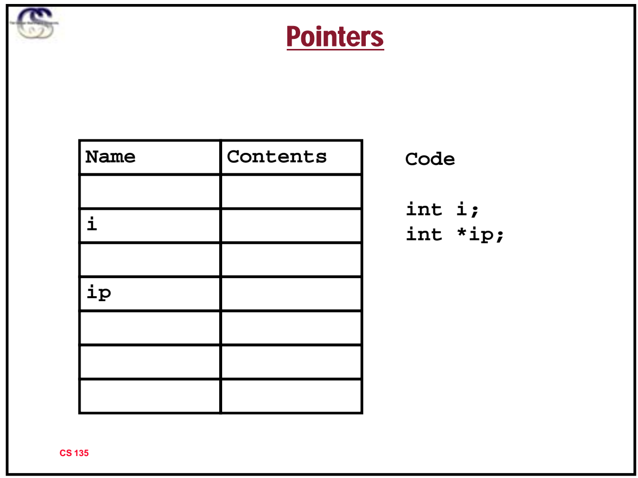
- If allocated data is not freed, the program might run out of heap memory and be unable to continue.
  - Even though it is a local variable, and the values are 'destroyed', the allocator assumes the memory is still in use!

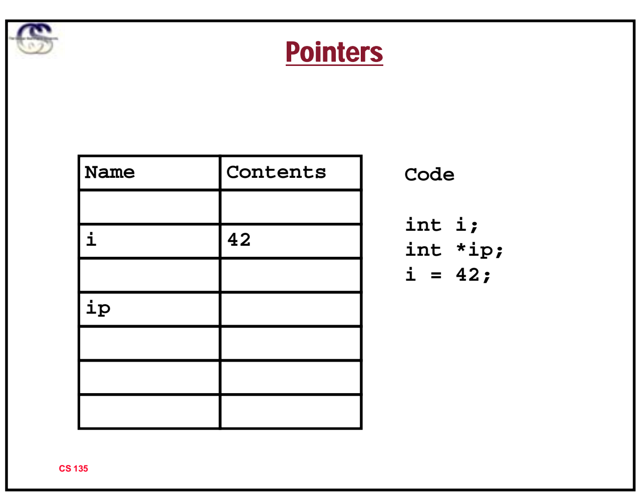


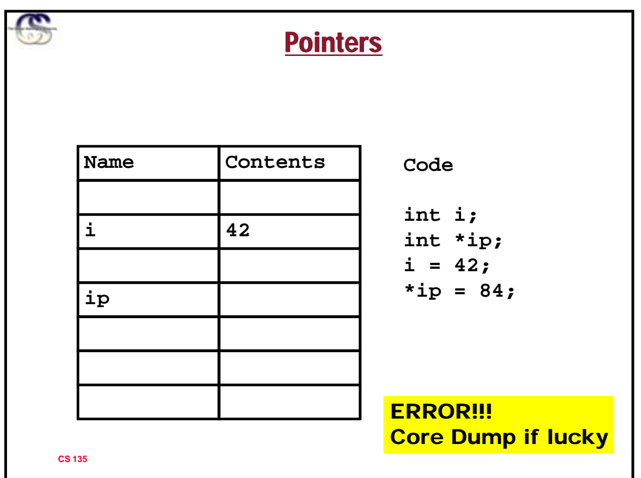


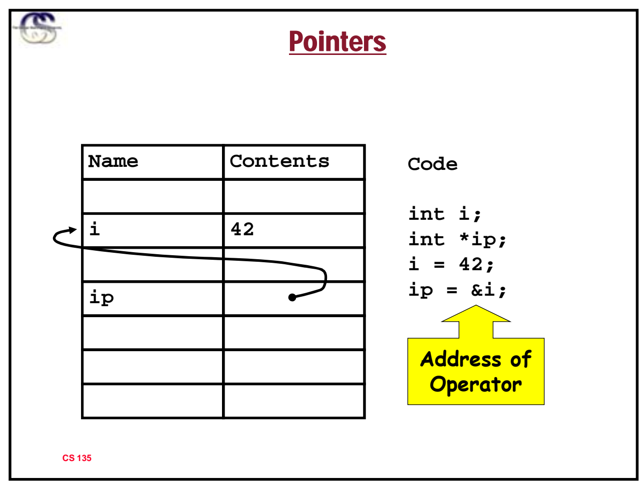

Syntax for Pointer Operators
•Declaring a pointer
type *var;
type* var;
•Either of these work -- whitespace doesn't matter.
Type of variable is int* (integer pointer), char* (char pointer), etc.
•Creating a pointer
&var
•Must be applied to a memory object, such as a variable.
In other words, &3 is not allowed.
•Dereferencing
Can be applied to any expression. All of these are legal:
*var contents of mem loc pointed to by var
**var contents of mem loc pointed to by memory location pointed to by var
*3 contents of memory location 3
CS 135

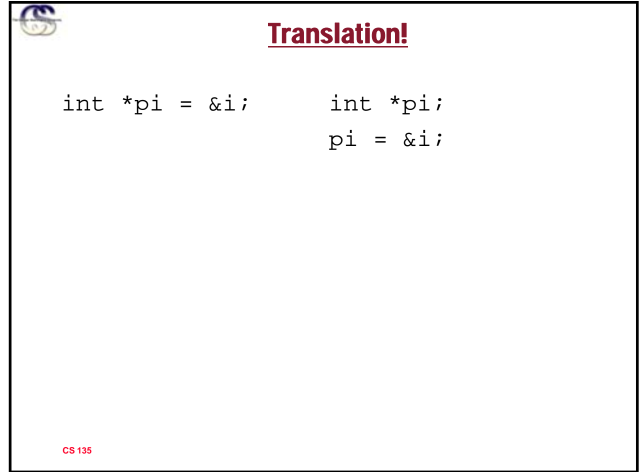

Pointers
Name
Contents
i
Code
int i;
CS 135

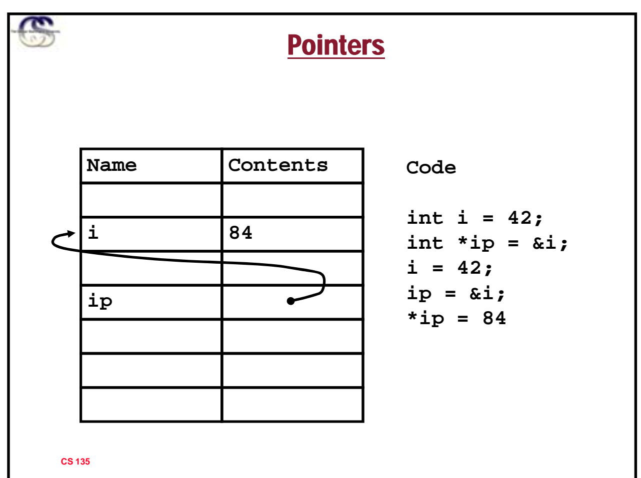

Pointers
Name
Contents
i
ip
Code
int i;
int *ip;
CS 135

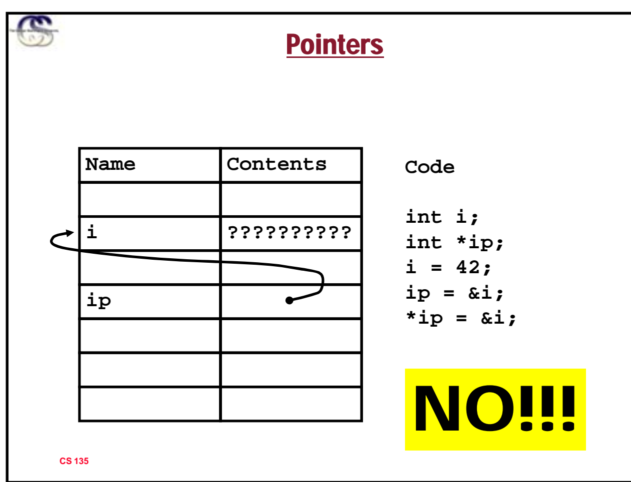

Pointers
Name
Contents
i
42
ip
Code
int i;
int *ip;
i = 42;
CS 135

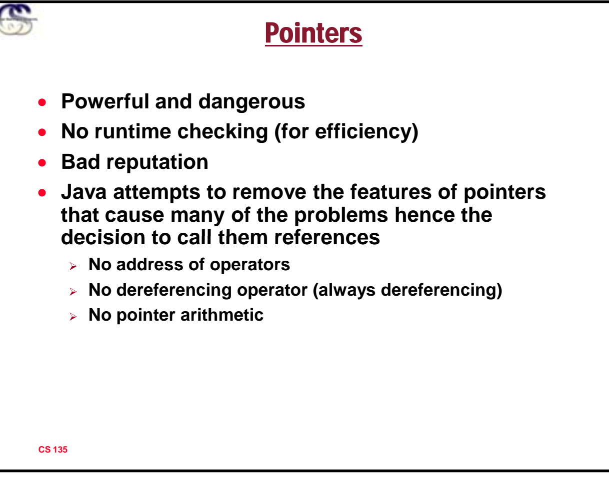

Pointers
Name
Contents
i
42
ip
Code
int i;
int *ip;
i = 42;
*ip = 84;
ERROR!!!
Core Dump if lucky
CS 135

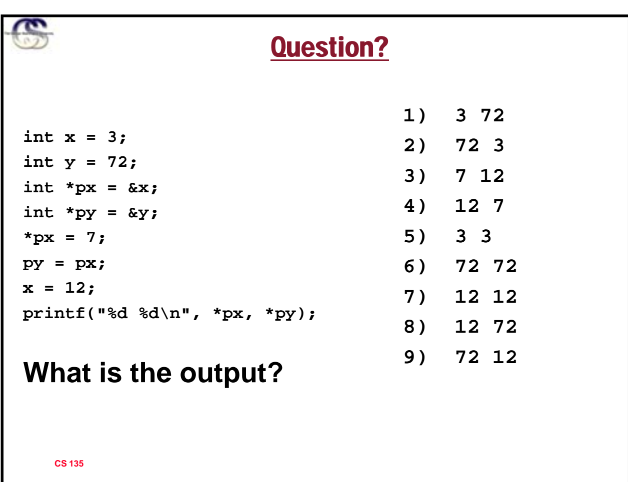

Pointers
Name
Contents
i
42
ip
Code
int i;
int *ip;
i = 42;
ip = &i;
Address of
Operator
CS 135


Translation!
int *pi = &i;
int *pi;
pi = &i;
CS 135


Pointers
Name
Contents
i
84
ip
Code
int i = 42;
int *ip = &i;
i = 42;
ip = &i;
*ip = 84
CS 135


Pointers
Name
Contents
i
??????????
ip
Code
int i;
int *ip;
i = 42;
ip = &i;
*ip = &i;
NO!!!
CS 135


Pointers
Powerful and dangerous
No runtime checking (for efficiency)
Bad reputation
Java attempts to remove the features of pointers that cause many of the problems hence the decision to call them references
No address of operators
No dereferencing operator (always dereferencing)
No pointer arithmetic
CS 135


Question?
int x = 3;
int y = 72;
int *px = &x;
int *py = &y;
*px = 7;
py = px;
x = 12;
printf("%d %d\n", *px, *py);
What is the output?
1) 3 72
2) 72 3
3) 7 12
4) 12 7
5) 3 3
6) 72 72
7) 12 12
8) 12 72
9) 72 12
CS 135