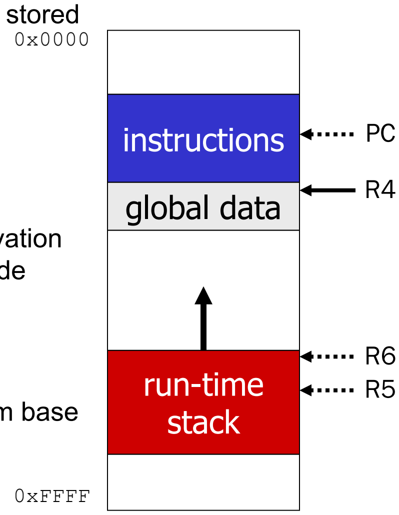
Pointers & Arrays in C & Translation to Assembly

(Chapters 16, 19)

LC3 Memory Allocation & Activation Records

- Global data section: global variables stored here
  - R4 points to beginning
- Run-time stack: for local variables
  - R6 points to top of stack
  - R5 points to top frame on stack
  - Local variables are stored in an activation record, i.e., stack frame, for each code block (function)
  - New frame for each block/function (goes away when block exited)
- symbol table "offset" gives distrance from base of frame (R5 for local var).
  - Address of local var = R5 + offset
  - Address of global var = R4 + offset
- return address from subroutines in R7



Next: Pointers, Arrays, (I/O), Structs . . .

- The real fun stuff in C…..
- Pointers and Arrays
  - Read Chapters 16, 18 of text
- Dynamic data structures
  - Allocating space during run-time
  - Read chapter 19 of text
- C skills...Labs will cover some of these
  - Make files
  - File I/O
  - Debugging – GDB
  - Valgrind
- why do you need to know these ?

3

3

C Review: Pointers and Arrays

- 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
  - Expression `a[4]` refers to the 5th element of the array `a`

4

4

Pointers

- What are pointers?
  - a variable that contains the address of a memory location
- Ex: `int *my_ptr ;   // declaration`
  - Declares a variable called *my_ptr* that contains address of an int.
  - The asterisks: * tells compiler this isn't an integer variable
    - It is a variable that will hold the address of an integer!
  - We know this from Assembly:
    - R0 can hold address of a slot in data memory

5

5

Pointers

- Example of use:

```
int  a=0 ; // declares a regular integer variable
int *b   ; // declares a pointer to an integer var.
b=&a ;     // finds "address" of a, assigns it to b
*b=5 ;     // dereferences b, sets value of a=5
```

Address	Contents
x4000 (a)	5
x4001 (b)	**x4000**

Dereferencing – fancy word for: contents at address

Dereferencing pointer b means:

get contents of memory at the address b is pointing to

6

6

Pointers

- Two language mechanisms for supporting pointers in C
  - * : for dereferencing a pointer
    - called the “Indirection” or “Dereference” operator
  - & : for getting the address of a variable
    - :called the “Address Operator”
  - These “unary” operators are called Pointer Operators
- Note: There is a difference between pointer operators and declaring pointer variables:
    - **int *** my_pointer ;
      - “int *” in this context is a “type” not the use of the operator *
    - Confused? Chapter 16 in Patt/Patel is outstanding!

7

7

Why use pointers….
Passing by value is not enough

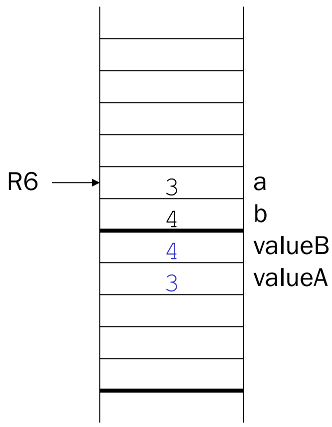
- In C, arguments/parameters are passed by value
  - Arguments pushed onto run-time stack
- Example : you’ve seen this in *swap*:
  - function that's supposed to swap the values of its arguments.

8

8

Executing the Swap Function

before call

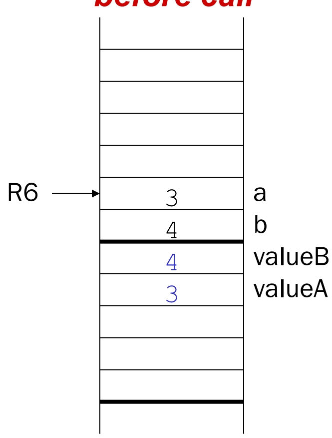



```
void Swap(int a, int b)
{
  int temp = a;
  a = b;
  b = temp;
   return;
}
 /* in main we call….*/
 Swap(valueA, valueB);
```

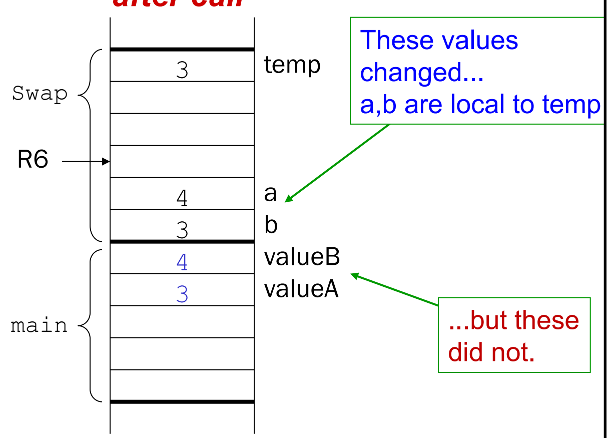
9

Executing the Swap Function

before call



after call



Swap needs <u>addresses</u> of variables outside its own activation record/scope.

10

Pointers as Arguments

- Passing a pointer into a function allows the function to *read/change memory outside its activation record*.
- Let's rewrite the swap function

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

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

We call it like this:

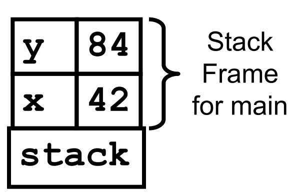
```
   int x = 42;
   int y = 84;
swap(&x, &y);
```

11

Tracing the run-time stack

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

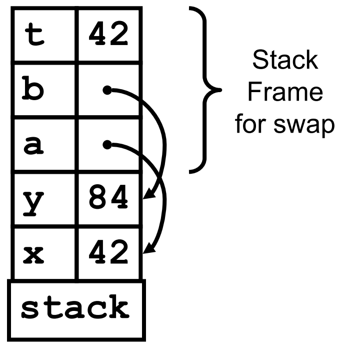
```
void swap(int *a, int *b)
{
   int t;
   t = *a;
   *a = *b;
   *b = t;
}
```



Tracing the call to swap

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


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

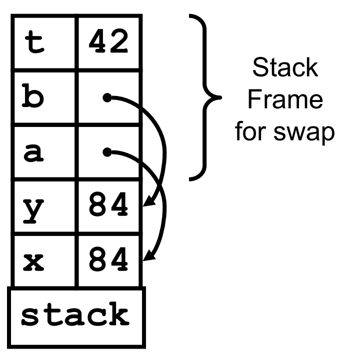


13

Trace

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


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



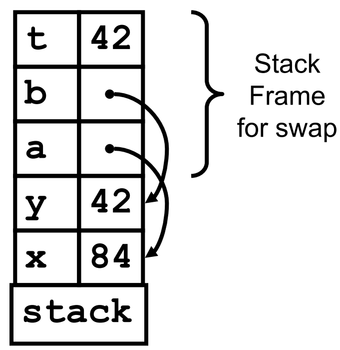
14

Trace

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



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



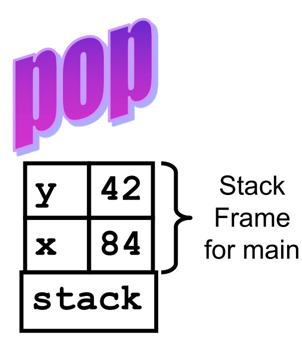
15

Trace

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



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



16

Passing Pointers & LC3 Code generation

•How do you pass pointers in the activation record (in LC3 compiler) ?

•Parameters to the function are the addresses of the arguments!

- Address for a local var is R5 + offset
- Set value of argument = R5+offset

17

17

Passing Pointers to a Function

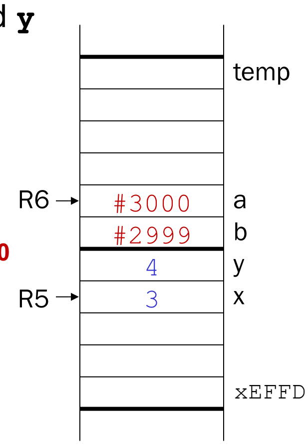
•main() wants to swap the values of **`x`** and **`y`**

•passes the addresses to Swap:

•

```
swap(&x, &y);
```

Address of x,y pushed onto stack



R5= #3000

•Code for passing arguments:

```
ADD R0, R5, #-1 ; addr of x
ADD R6, R6, #-1 ; push
STR R0, R6, #0
ADD R0, R5, #0  ; addr of y
ADD R6, R6, #-1 ; push
STR R0, R6, #0
```

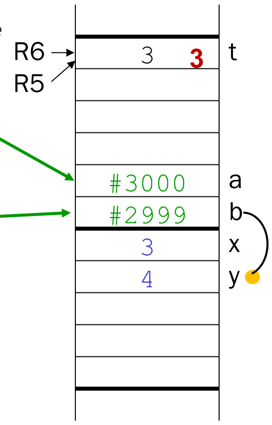
18

18

LC3 Code generation (for swap)

•Inside the Swap routine

```
  ; int t = *a; get a and dereference
  LDR   R0, R5, #4 ; R0=#3000
  LDR   R1, R0, #0 ; R1=M[#3000]=3
  STR   R1, R5, #0 ; t=3
; *a = *b; get b & dereference
  LDR   R1, R5, #5 ; R1=#2999
  LDR   R2, R1, #0 ; R1=M[#2999]=4
  STR   R2, R0, #0 ; M[#3000]=4 4
; *b = t;
  LDR   R2, R5, #0 ; R2=3
  STR   R2, R1, #0 ; M[#2999]=3
```



19

Example and C to LC3 translation

```
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

Symbol Table: 2 local variables `i, ptr`

Offset for `i` =0

Offset for `ptr = -1`

20

Example: LC-3 Code – Questions 1,2 in inclass-nov10.pdf

•**Symbol Table:** *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;
        ???              ; get  addr of i
        ???              ; store in ptr
•; *ptr = *ptr + 1;
       ???               ; get  ptr
       ???               ; dereference/load contents (*ptr)

       ???               ; add one
       ???               ; store result where ptr  points
```

21

21

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

22

22

Pointers & Arrays in C & Translation to Assembly: Part 2 – Arrays

23

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); <u>no limit checking</u> at compile-time or run-time

24

24

Arrays

- What are arrays?
  - a collection of many variables of the same type with an index
- Ex: `int my_array[10] ;  // declaration`
  - LC-3: allocates 10 slots for 16-bit integers in Data Memory
  - These are ***stored in consecutive locations in memory***

my_array → x4000

Just a label for memory location x4000

Address	Contents
x4000	X
x4001	X
x4002	X
…	…
x4008	X
x4009	X

On LC-3: 10 "16-bit" slots

Note: can't assume initialized to 0

25

25

Arrays

- Indexing Arrays
  - C offers "indexing" capability on array variables
- Ex: In this example: `my_array[2]` equals `4`
  - Allocates 10 slots for 16-bit integers in Data Memory
  - What happens when you type: my_array [11] ???

my_array → x4000

Offset of 2 from start: my_array → x4002

Address	Contents
x4000	X
x4001	X
x4002	4
…	…
x4008	X
x4009	X

On LC-3: 10 "16-bit" slots

Note: can't assume initialized to 0

Remember the offset?: `LDR RD, RS, Offset`

Imagine: `LDR R0, my_array, #2`

26

26

Arrays and pointers

- Arrays and pointers are intimately connected in C
  - Array declarations allocate areas of memory for use
  - We are really defining an address (aka – a pointer) to the first element of the array
- Example – mixing arrays and pointers!

```
int my_array[10]; // declares array of 10 ints
int *my_ptr;  // declares a pointer to an int var.
my_ptr = my_array + 2; // points to 3rd row in array
```

my_array → x4000

my_ptr=x4002 → x4002

Address	Contents
x4000	X
x4001	X
x4002	4
…	…
x4008	X
x4009	X

Dereferencing ptr:

`*my_ptr` **equals** 4

27

27

Pointers and Arrays – Assembly code versions

- In terms of assembly we can make a distinction between the address of the start of a block of memory and the values stored in that block of memory

C Code	Assembly Code
`int my_array[10]`	`my_array .BLKW #10`
`int *my_ptr; my_ptr = my_array;`	`LEA R0, my_array ; R0 is equiv to ; "my_ptr"`

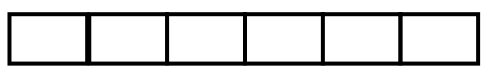
28

28

Arrays: Memory layout

```
int ia[6];
```

- Allocates consecutive spaces for 6 integers
- How much space is allocated?
  - Depends on the type of the array
  - How many bytes for an int ?
  - How many bytes for a char?
  - Ex: if 4 bytes for int, then we need 24 bytes for 6 integers
  - Ex: 1 byte for char, then we need 6 bytes for 6 character array

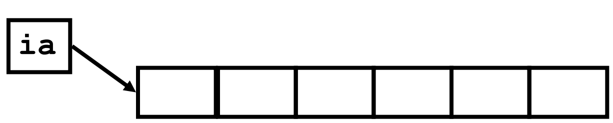


Arrays

```
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?



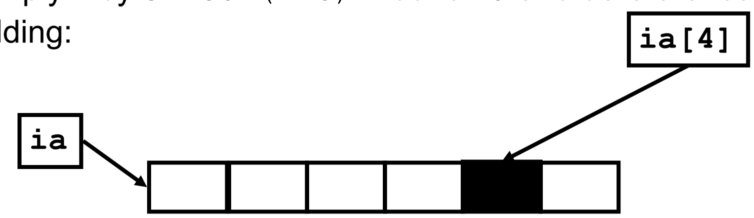
Arrays

```
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
  - Cannot change ia
- What does `ia[4]` mean?
- Multiply 4 by `sizeof(int)`. Add to `ia` and dereference yielding:



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

sizeof

- if sizeof(int) == 4 then sizeof(i) == 4
- On a typical 32 bit machine...

```
sizeof(*ip) → 4
sizeof(ip) → 4
char *cp;
sizeof(char) → 1
sizeof(*cp) → 1
sizeof(cp) → 4

int ia[6];
sizeof(ia) → 24
```

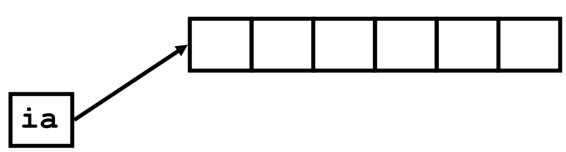
Not the same thing!!!

33

33

Arrays

```
int ia[6];
```



- **ia[4]** means ***(ia + 4)**

34

34

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!
- C does size calculations under the covers, depending on size of item being pointed to:

```
•double x[10];
•double *y = x;
     *(y + 3) = 13;
```

allocates 20 words (2 per element)

same as x[3] -- base address plus 6

35

35

Pointer Arithmetic

•Address calculations depend on size of elements

- In our LC-3 code, we've been assuming one word per element.
  - e.g., to find 4th element, we add 4 to base address
- It's ok, because we've only shown code for int and char, both of which take up one word.
- If double, we'd have to add 8 to find address of 4th element.

36

36

Pointer Arithmetic

- Just as we used arithmetic on address values to iterate through arrays in assembly, we can use arithmetic on pointer values in C

```
•float my_array[10]; // declares array of 10 floats
•float *my_ptr;      // declares a pointer to a float
•my_ptr = my_array + 2; // points to 3rd row in array
•my_ptr = my_ptr + 1;   // points to 4th row in array
```

  - Compiler looks at the type of variable being pointed to and increments by the correct amount to point to the next element
  - In this case ptr may actually be incremented by 4 since each float takes up 4 bytes

37

37

Pointers/Arrays/Strings

- There is no “string” datatype in C
  - But we can use arrays of char’s to mimic behavior
- Simplest Ways to Declare “Strings”:
  - **`char my_string [256] ;`**
    - Works just like any array, each element is character
      `my_string[0]='T' ;`
      `my_string[1]='h' ;`
    - You must “null terminate” this array
    - Note: no way to know length of an array
      - Unless one loops through it entirely and determines ending
    - Pass “my_string” as argument to functions!
      - That’s the 1st address of the string in memory
  - **`char *my_string = "This is a string" ;`**
    - Will be null terminated
    - Cannot be modified

38

38

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.)

39

39

Correspondence between Ptr and Array Notation

•Given the declarations,
each line below gives three equivalent expressions:

•`cptr`	`word`	`&word[0]`
•`(cptr + n)`	`word + n`	`&word[n]`
•`*cptr`	`*word`	`word[0]`
•`*(cptr + n)`	`*(word + n)`	`word[n]`

```
char word[10];
char *cptr;
cptr = word;/* points to word[0] */
```

40

40

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 + inputValues[index];
   return (sum / MAX_NUMS);
}
```

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

41

41

Array as a Local Variable

```
int foo(int myarray[ ] )
{
  int grid[10];
…
}
```

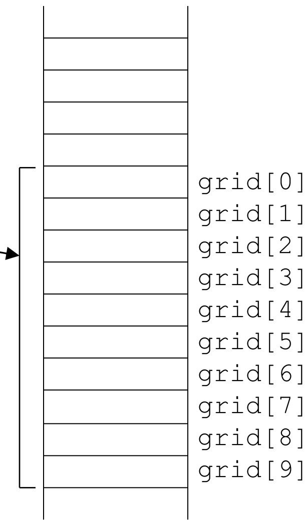
42

42

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]`.

43

43

C to Assembly Translation- Arrays

44

44

Example and C to LC3 translation

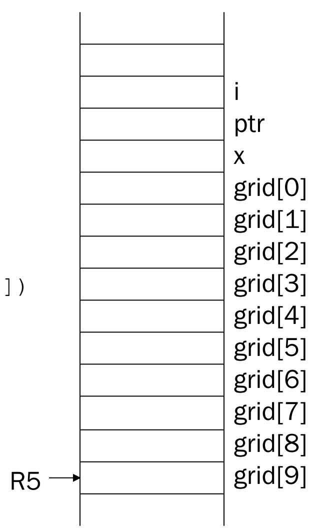
```
int foo(){
int grid[10];
int x,
int *ptr;
int i;
     ….
     grid[6] =5;
     x = grid[3] + 1;
     grid[i] = x;
     ptr = grid;
     grid[x] = grid[x] +2;
     …
}
```

Symbol Table

Identifier	offset
grid	-9
x	-10
ptr	-11
i	-12

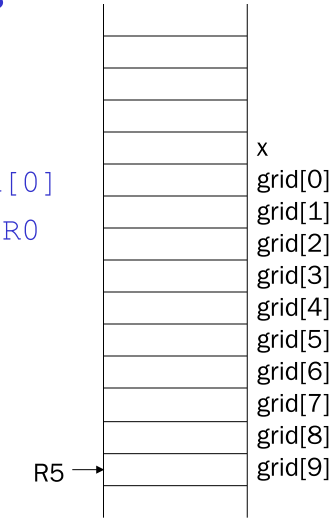
LC-3 Code for Array References

```
grid[6] =5;
x = grid[3] + 1;
grid[i] = x;
ptr=grid;
grid[x+1] = grid[x] +2;
 where is @grid[0] (address grid[0])
 ??? @grid[6] = @grid[0} +6
??? ; plus 1
 STR ??? ; store into x
@grid[i] = @grid[0] + i
```



LC-3 Code for Array References

```
; 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
```

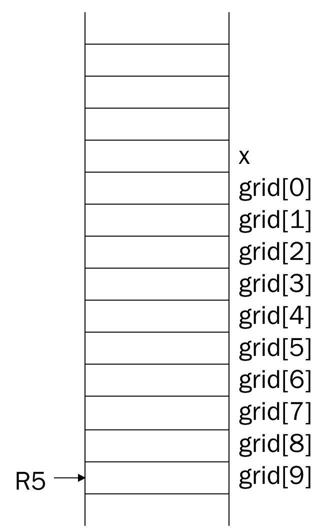


47

47

LC-3 Code for Array References

```
; x = grid[3] + 1
  ; R0 = &grid[0]
  ; R1 = grid[3]
  ; add 1
 ; store into x = R1
```



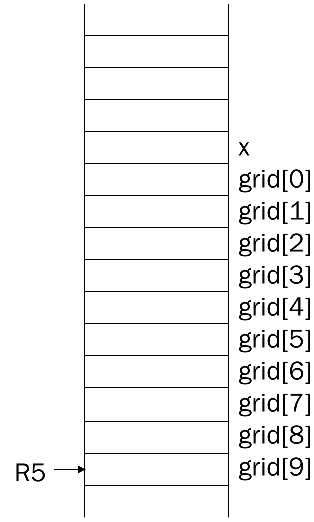
48

48

More LC-3 Code

```
; grid[x+1] = grid[x] + 2
  LDR R0, R5, #-10   ; R0 = x
  ADD R1, R5, #-9    ; R1 = &grid[0]
  ADD R1, R0, R1     ; R1 = &grid[x]
  LDR R2, R1, #0     ; R2 = grid[x]
  ADD R2, R2, #2     ; add 2

  LDR R0, R5, #-10   ; R0 = x
  ADD R0, R0, #1     ; R0 = x+1
  ADD R1, R5, #-9    ; R1 = &grid[0]
  ADD R1, R0, R1     ; R1 = &grix[x+1]
  STR R2, R1, #0     ; grid[x+1] = R2
```



49

49

Common Pitfalls with Arrays in C

- Overrun array limits
  - There is no checking at run-time or compile-time to see whether reference is within array bounds.

```
int array[10];
int i;
for (i = 0; i <= 10; i++) array[i] = 0;
```

- Declaration with variable size
  - Size of array must be known at compile time.

```
void SomeFunction(int num_elements) {
  int temp[num_elements];
  …
}
```

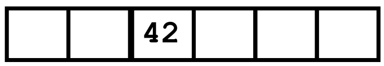
50

50

Recall

```
int ia[6];

ia[2] = 42;
```



Address calculation:

`2 * sizeof(*ia) + ia`

Access is by dereferencing

`*(2 * sizeof(*ia) + ia)`

Remember!
You don't type in
the sizeof part!

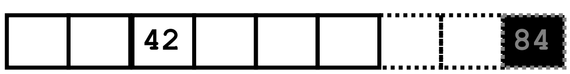
51

51

What happens?

```
int ia[6];

ia[8] = 84;
```



Address calculation:

`8 * sizeof(*ia) + ia`

Remember!
You don't type in
the sizeof part!

52

52

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	

53

53

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	

54

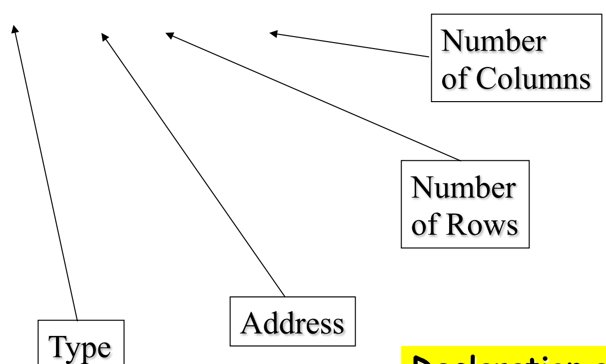
54

Multidimensional Arrays in C

55

Declaration

```
int ia[3][4];
```

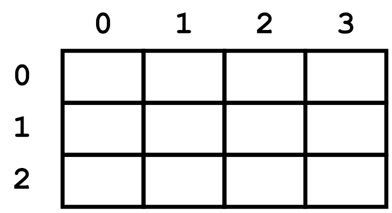


Declaration at compile time
i.e. size must be known

56

56

How does a two dimensional array work?

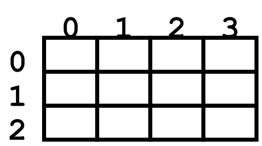


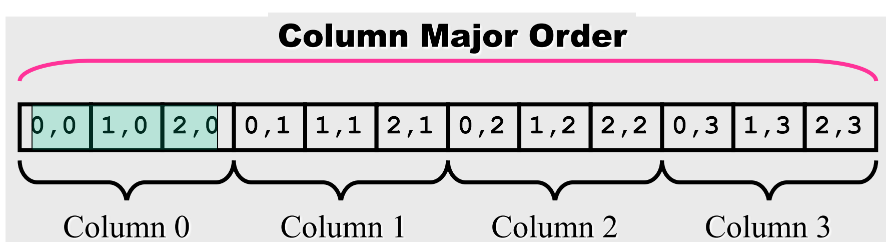
How would you store it?

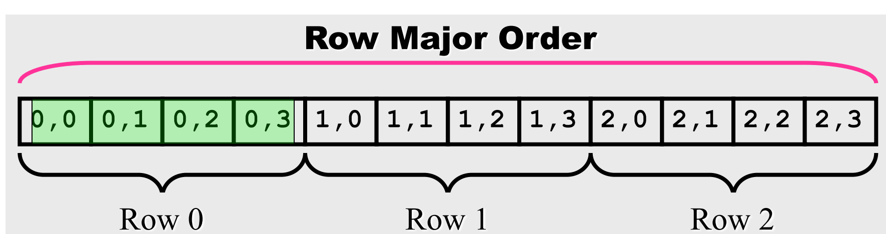
57

57

How would you store it?







58

58

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

59

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

60

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)
```

61

61

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

62

62

How does C store arrays

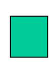
- Row major
  - Pointer arithmetic stays unmodified

- Remember this…..
  - Affects how well your program does when you access memory

63

Now think about

- A 3D array

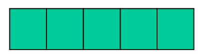


`int a`

64

Now think about

- A 3D array



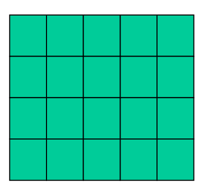
`int a[5]`

65

65

Now think about

- A 3D array



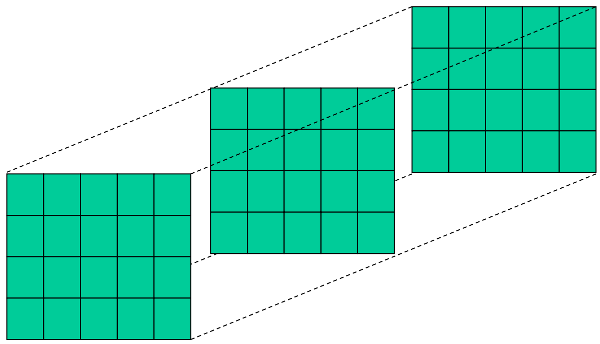
`int a[4][5]`

66

66

Now think about

- A 3D array



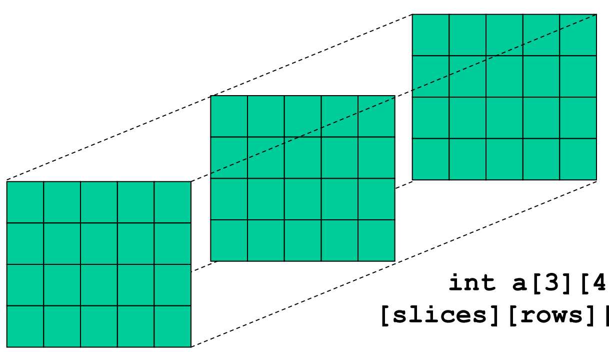
`int a[3][4][5]`

67

67

Offset to a[i][j][k]?

- A 3D array



```
int a[3][4][5]
[slices][rows][columns]
```

```
offset = (i * rows * columns) + (j * columns)
         + k
```

68

68

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]`

69

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

70

70