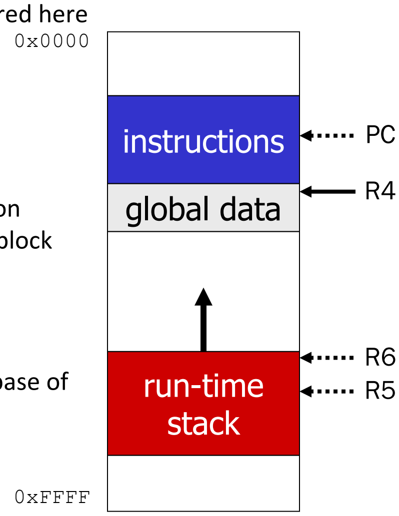
Pointers & Arrays in C & Translation to Assembly

(Chapters 16, 19)

1

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



2

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 … malloc() and free()
  - 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 **`arr[4]`** refers to the 5th element of the array **`arr`**
    - Turns out arr is also pointer to first element in array !

4

4

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!

5

5

Pointers

- Pointer: variable that contains address of a memory location
- Example of use:

```
int  a=0 ; // declares a regular integer variable
int *b   ; // declares a pointer to an integer var.
             // asterisk * tells compiler this is a ptr
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

Why use pointers….
Passing by value is not enough

- In C, arguments/parameters (to function) are passed by value
  - values of Arguments pushed onto run-time stack

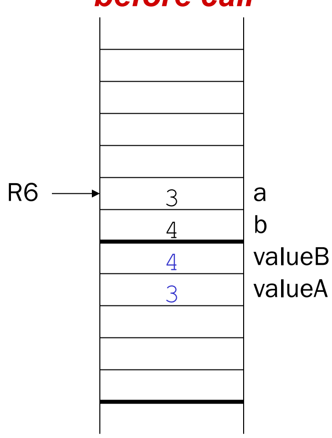
- Example : you've seen this in *swap (quiz)*:
  - function that's supposed to swap the values of its arguments.
  - variables in main remain local to main… foo cannot access them

7

7

Executing the Swap Function

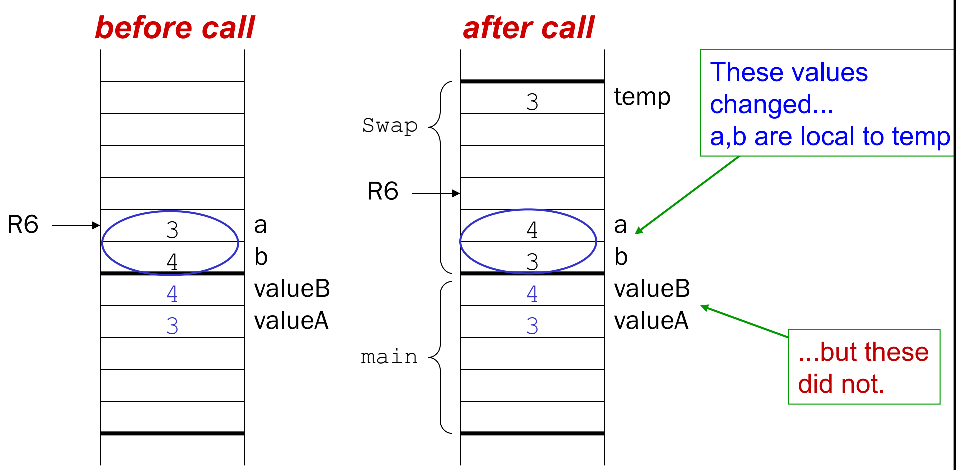
before call




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

8

Executing the Swap Function



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

9

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 <u>pointers</u>. Caller passes <u>addresses</u> of variables that it wants function to change.

We call it like this:

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

10

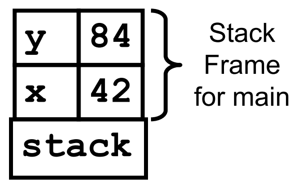
10

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;
}
```



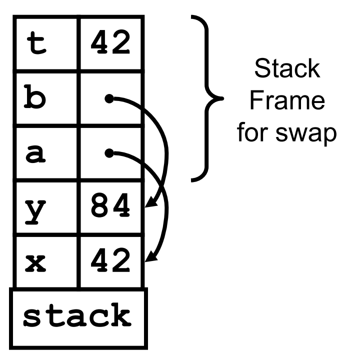
11

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;
}
```

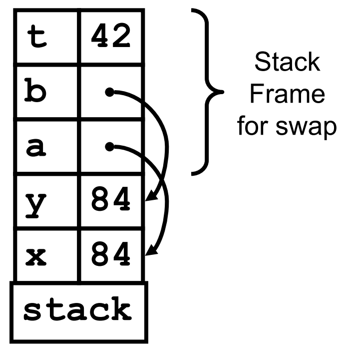


12

Trace

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


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

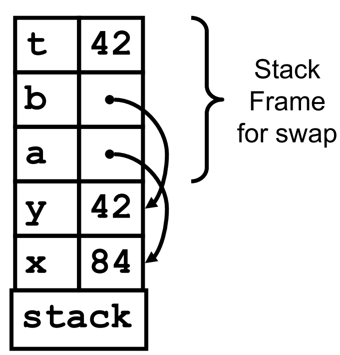


13

Trace

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


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



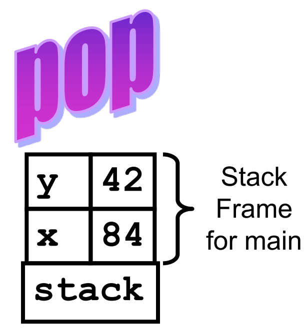
14

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

Passing Pointers & LC3 Code generation

- How to 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

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

Symbol Table offsets

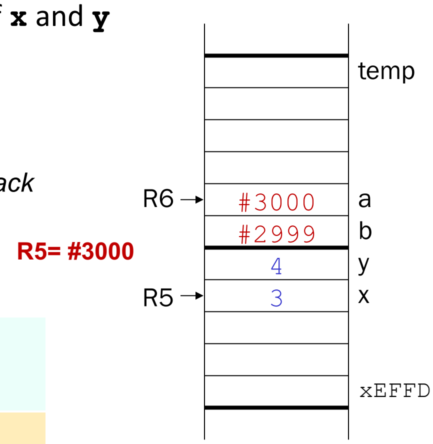
a	4
b	5
t	0

16

16

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



- Code for passing arguments:

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

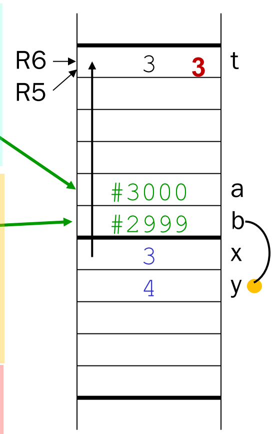
17

17

LC3 Code generation (for swap)

- Inside the Swap routine

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



18

18

Pointers

- Powerful and dangerous
  - What happens with *x if x is pointing to memory outside your user space?
- 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

19

19

Pointers

Name	Contents
i	

Code

```
int i;
```

20

20

Pointers

Name	Contents
i	
ip	

Code

```
int i;
int *ip;
```

21

21

Pointers

Name	Contents
i	42
ip	

Code

```
int i;
int *ip;
i = 42;
```

22

22

Pointers

Name	Contents
i	42
ip	

Code

```
int i;
int *ip;
i = 42;
*ip = 84;
```

ERROR!!!
Core Dump if lucky

23

23

Pointers

Name	Contents
i	42
ip	

Code

```
int i;
int *ip;
i = 42;
ip = &i;
```

Address of Operator

24

24

Pointers

Name	Contents
i	84
ip	

Code

```
int i = 42;
int *ip = &i;
i = 42;
ip = &i;
*ip = 84
```

25

25

Pointers

Name	Contents
i	??????????
ip	

Code

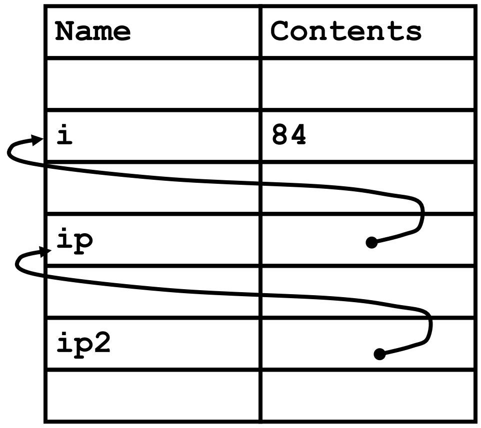
```
int i;
int *ip;
i = 42;
ip = &i;
*ip = &i;
```

NO!!!

26

26

Pointers



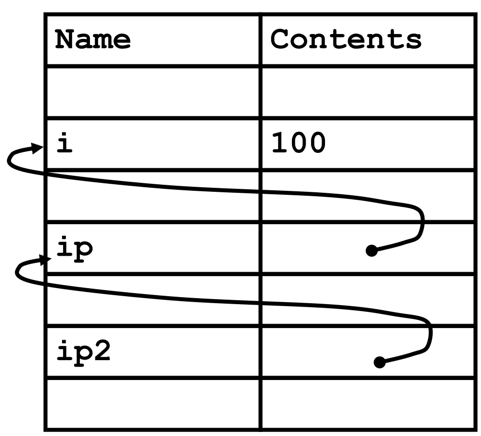
Code

```
int i = 42;
int *ip = &i;
i = 42;
ip = &i;
*ip = 84
int **ip2;
ip2 = &ip;
```

27

27

Pointers



Code

```
int i = 42;
int *ip = &i;
i = 42;
ip = &i;
*ip = 84
int **ip2;
ip2 = &ip;
**ip2=100;
```

28

28

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

29

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

30

30

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 → (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

31

31

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`

32

32

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**

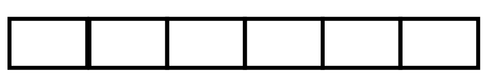
33

33

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



35

35

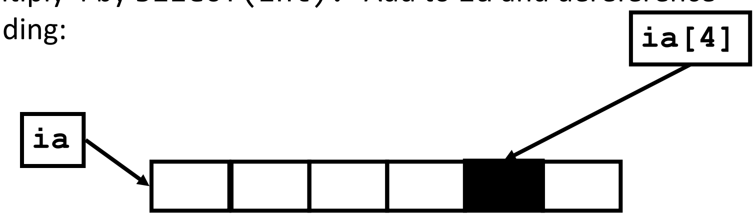
Arrays

```
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:



37

37

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

38

38

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
```

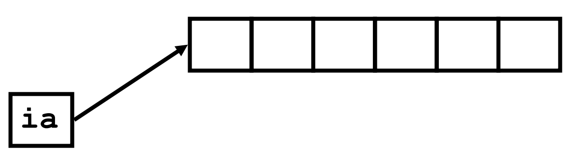
Not the same thing!!! (referring to sizeof(*ip) → 4 and sizeof(ip) → 4)

39

39

Arrays & Pointer Arithmetic

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



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

40

40

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];          <- allocates 20 words (2 per element)
•double *y = x;
      *(y + 3) = 13;    <- same as x[3] -- base address plus 6
```

41

41

Pointers/Arrays/Strings…more in Labs & HW6

- 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

44

44

Summary: Relationship between Arrays and Pointers

- 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.)

45

45

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`

47

47

Array as a Local Variable

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

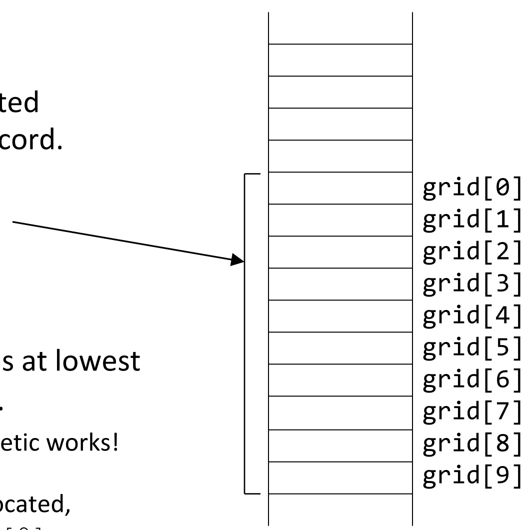
48

48

Array as a Local Variable

- if array is 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.
  - Why ?...so pointer arithmetic works!

  If `grid` is first variable allocated, then R5 will point to `grid[9]`.

49

49

Example and C to LC3 translation

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

Symbol Table

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

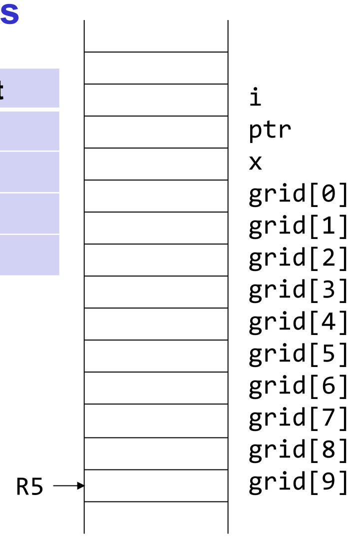
50

50

LC-3 Code for Array References

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

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



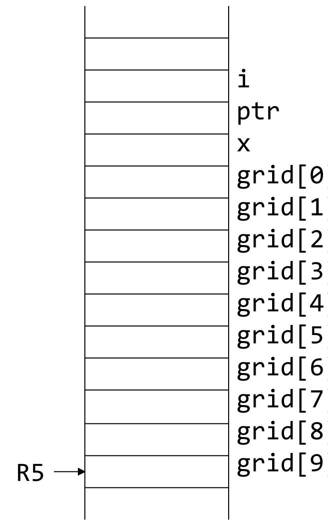
51

51

LC-3 Code for Array References

```
grid[6] = 5;
  where is &grid[0]? (address)?
  &grid[6] = &grid[0] +6


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
```



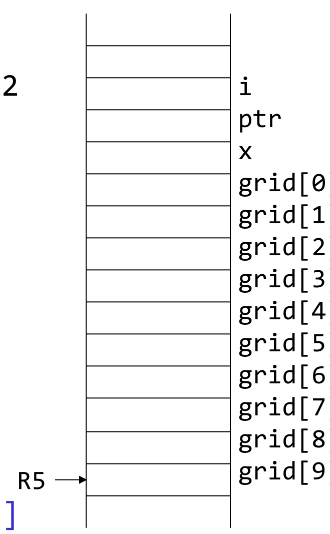
52

52

LC-3 Code for Array References

```
x=grid[i];
get value of i: from address R5-12
get &grid[0]? (address)?
 &grid[i] = &grid[0] +I
Store into x = address R5-10

x =grid[i];
  LDR R0, R5, # -12 ; R0= i
  ADD R1, R5, # -9   ; R1= &grid[0]
  ADD R1, R1, R0 ; R1 = &grid[i]
  LDR R2, R1, #0 ; R2 = grid[i]
  STR R2, R5, # -10   ; x=R2=grid[i]
```



53

53

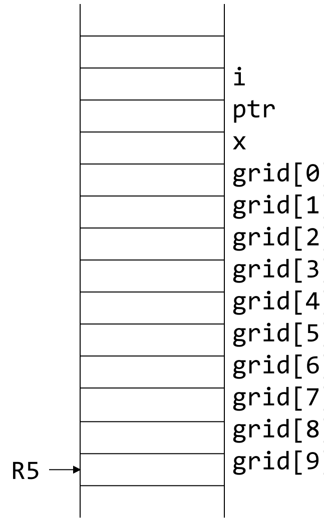
LC-3 Code for Array References

ptr=grid;

get address of grid[0]

set value of ptr to address of grid[0]

```
ptr =grid;
  ADD R1, R5, # -9   ; R1= &grid[0]
  STR R1, R5, # -11   ; ptr= R1
```



54

54

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];
  …
}
```

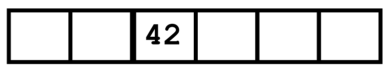
55

55

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!

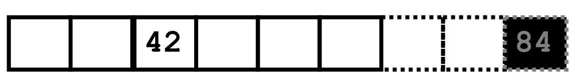
56

56

What happens?

```
int ia[6];

ia[8] = 84;
```



Address calculation:

`8 * sizeof(*ia) + ia`

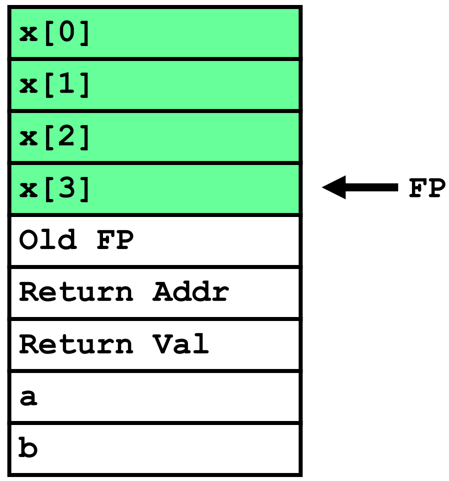
Remember!
You don't type in
the sizeof part!

57

57

Stack Smashing

```
int another(int a, int b) {
    int x[4];
```

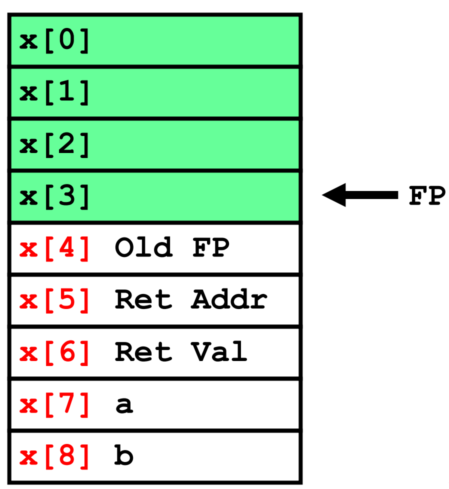


58

58

Stack Smashing

```
int another(int a, int b) {
    int x[4];
```



59

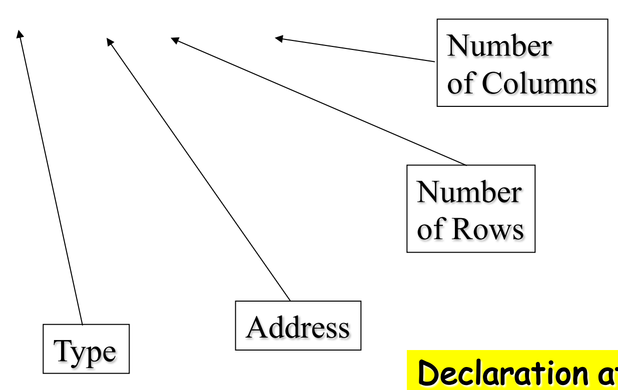
59

Multidimensional Arrays in C

60

Declaration

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

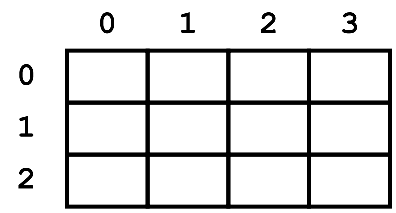


Declaration at compile time
i.e. size must be known

61

61

How does a two dimensional array work?

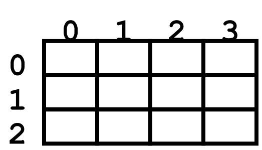


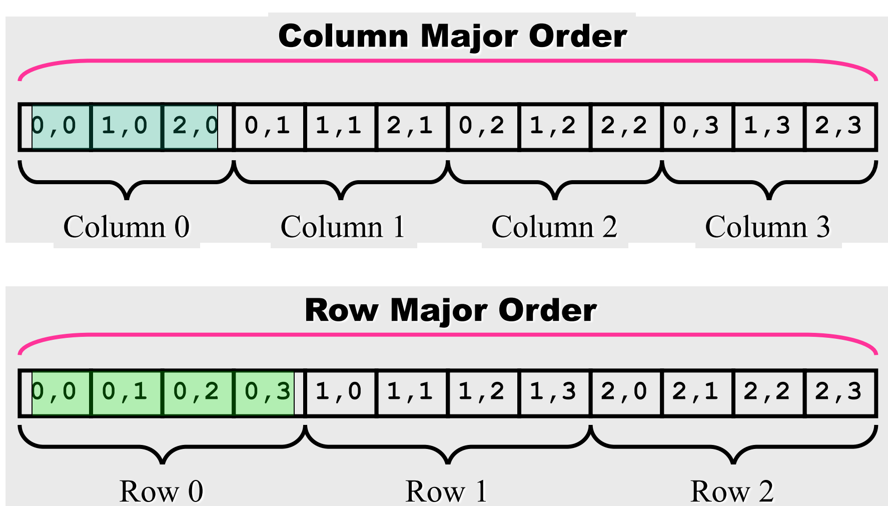
How would you store it?

62

62

How would you store it?





63

63

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

64

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

65

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)
```

66

66

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

67

67

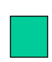
How does C store arrays

- Row major
  - Pointer arithmetic stays unmodified
- Remember this…..
  - Affects how well your program does when you access memory

68

Now think about

- A 3D array

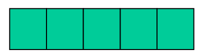


`int a`

69

Now think about

- A 3D array



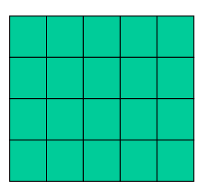
`int a[5]`

70

70

Now think about

- A 3D array



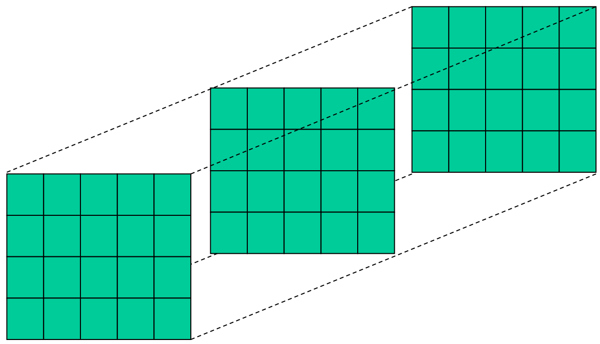
`int a[4][5]`

71

71

Now think about

- A 3D array



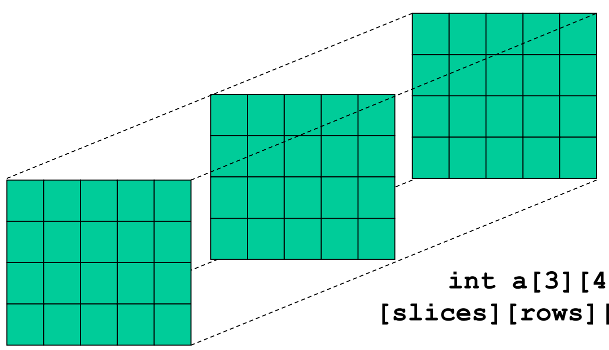
```
int a[3][4][5]
```

72

72

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

- A 3D array



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

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

73

73

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

75