**CS 2461: Computer Architecture I**

**Functions, Arrays, Pointers**

- 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

---

**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
  - Example: a list of telephone numbers
  - Expression `a[4]` refers to the 5th element of the array `a`

---

**Passing by value is not enough**

- **Parameters are passed by value**
  - Arguments pushed onto run-time stack

- **Example 2 (test2.c): you've seen this:**
  - function that's supposed to swap the values of its arguments.
Executing the Swap Function

before call

\[
\begin{align*}
R6 & \rightarrow 3 \\
\text{firstVal} & \rightarrow 4 \\
\text{secondVal} & \rightarrow 4 \\
\text{valueB} & \rightarrow 3 \\
\text{valueA} & \rightarrow 3 \\
\end{align*}
\]

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

after call

\[
\begin{align*}
R6 & \rightarrow 3 \\
\text{tempVal} & \rightarrow 4 \\
\text{firstVal} & \rightarrow 3 \\
\text{secondVal} & \rightarrow 4 \\
\text{valueB} & \rightarrow 3 \\
\text{valueA} & \rightarrow 3 \\
\end{align*}
\]

These values changed...

...but these did not.

Swap needs addresses of variables outside its own activation record.

Executing the Swap Function

before call

\[
\begin{align*}
R6 & \rightarrow 3 \\
\text{firstVal} & \rightarrow 4 \\
\text{secondVal} & \rightarrow 4 \\
\text{valueB} & \rightarrow 3 \\
\text{valueA} & \rightarrow 3 \\
\end{align*}
\]

main

Swap

\[
\begin{align*}
\text{tempVal} & \rightarrow 3 \\
\text{firstVal} & \rightarrow 4 \\
\text{secondVal} & \rightarrow 4 \\
\text{valueB} & \rightarrow 3 \\
\text{valueA} & \rightarrow 3 \\
\end{align*}
\]

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

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

Note: can't assume initialized to 0!

\[
\begin{array}{|c|c|}
\hline
\text{Address} & \text{Contents} \\
\hline
x4000 & X \\
x4001 & X \\
x4002 & X \\
\ldots & \ldots \\
x4008 & X \\
x4009 & X \\
\hline
\end{array}
\]
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

<table>
<thead>
<tr>
<th>Address</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>x4000</td>
<td>X</td>
</tr>
<tr>
<td>x4001</td>
<td>X</td>
</tr>
<tr>
<td>x4002</td>
<td>4</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>x4008</td>
<td>X</td>
</tr>
<tr>
<td>x4009</td>
<td>X</td>
</tr>
</tbody>
</table>

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**

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:
    - RS can hold address of a slot in data memory

- 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
  ```

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
  ```

Pointers

- Two language mechanisms for supporting pointers in C
  - `*` : for dereferencing a pointer
  - `&` : for getting the address of a variable

- For your future knowledge:
  - `*` : called the “Indirection” or “Dereference” operator
  - `&` : 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 `*`

Dereferencing – fancy word for: contents at address
Dereferencing pointer b means:
- get contents of memory at the address b is pointing to

<table>
<thead>
<tr>
<th>Address</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>x4000(a)</td>
<td>5</td>
</tr>
<tr>
<td>x4001(b)</td>
<td>x4000</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>Address</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>x4000</td>
<td>X</td>
</tr>
<tr>
<td>x4001</td>
<td>X</td>
</tr>
<tr>
<td>x4002</td>
<td>4</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>x4008</td>
<td>X</td>
</tr>
<tr>
<td>x4009</td>
<td>X</td>
</tr>
</tbody>
</table>

Dereferencing ptr: *my_ptr equals 4

Pointers and Arrays

- Pointers and arrays are intimately related
- 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

<table>
<thead>
<tr>
<th>C Code</th>
<th>Assembly Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>int my_array[10]</td>
<td></td>
</tr>
<tr>
<td>int *my_ptr;</td>
<td></td>
</tr>
<tr>
<td>my_ptr = my_array;</td>
<td></td>
</tr>
<tr>
<td>;</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>C Code</th>
<th>Assembly Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>; R0 is equiv to</td>
<td></td>
</tr>
<tr>
<td>“my_ptr”</td>
<td></td>
</tr>
</tbody>
</table>

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 w
  - 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

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
Exercise 1....test your pointer skills!

- Go to lectures page, download
- test3.c – do NOT run the code
- Read through code and write down (and submit along with Exercise 2 in a while) answers to the questions
- Next, run the code and check your answers

Pointers

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

- Pointer: Variable that contains address of another variable.
- Operators:
  - *p returns value pointed to by p
  - &z returns address of variable z
- A pointer is a data object which is separate from what it points to.
  - Pointer to a data type
  - ip is a pointer to an integer

Example and C to LC3 translation

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

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;
?? ; get addr of i
?? ; store in ptr
: *ptr = *ptr + 1;
?? ; get ptr
dereference/load contents (*ptr)
?? ; add one
?? ; store result where ptr points
```
Passing a pointer into a function allows the function to read/change memory outside its activation record.

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

Now it works...

- We call it like this
  ```c
  int x = 42;
  int y = 84;
  swap(&x, &y);
  ```

Trace

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

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

Stack Frame for main
int x = 42;
int y = 84;
swap(x, y);

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

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

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

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

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

How do you pass pointers in the activation record?
• Using LC3 compiler...

Parameters to the function are the addresses of the arguments!

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

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
Arrays

- How do we allocate a group of memory locations?

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

Arrays: Memory layout

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

- Allocates consecutive spaces for 6 integers
- How much space is allocated?

Arrays

```c
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?
  
  \[ 6 \times \text{sizeof}(\text{int}) \]
- 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(\text{int})`. Add to `ia` and dereference yielding:

```
ia[4]
```

---

**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

```
if \text{sizeof}(\text{int}) == 4 \text{ then } \text{sizeof}(i) == 4
```

On a typical 32 bit machine...

```
\text{sizeof}(*ip) \rightarrow 4
\text{sizeof}(ip) \rightarrow 4
\text{char } *cp;
\text{sizeof}(\text{char}) \rightarrow 1
\text{sizeof}(*cp) \rightarrow 1
\text{sizeof}(cp) \rightarrow 4
```

```
\text{sizeof}(\text{ia}) \rightarrow 24
```

- `ia[4]` means `*(\text{ia} + 4)`

---

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

- `ia[4]` means `*(\text{ia} + 4)`
Pointer Arithmetic

- Note on the previous slide when we added the literal 4 to a pointer it actually gets interpreted to mean
- \( 4 \times \text{sizeof(thing being pointed at)} \)
- This is why pointers have associated with them what they are pointing at!

Arrays

\[
\text{int ia[6];}
\]

- Array elements are numbered like this since that's how the pointer arithmetic works out!

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.

- C does size calculations under the covers, depending on size of item being pointed to:
  - \( \text{double x[10];} \)
  - \( \text{double *y = x; \quad *(y + 3) = 13;} \)
    - allocates 20 words (2 per element)
    - same as \( x[3] \) -- base address plus 6

Relationship between Arrays and Pointers

- An array name is essentially a pointer to the first element in the array
  \[
  \begin{align*}
  \text{char word[10];} \\
  \text{char *cptr;}
  \end{align*}
  \]
  \[
  \text{cptr = word; /* points to word[0] */}
  \]
- Difference:
  Can change the contents of cptr, as in
  - \( \text{cptr = cptr + 1;} \)
  - (The identifier "word" is not a variable.)
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

```c
main() {
    int numbers[MAX_NUMS];
    mean = Average(numbers);
    ...
}
```

```c
int Average(int inputValues[MAX_NUMS]) {
    ...
    for (index = 0; index < MAX_NUMS; index++)
        sum = sum + inputValues[index];
    return (sum / MAX_NUMS);
}
```

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

Array as a Local Variable

- Array elements are allocated as part of the activation record.
- First element (grid[0]) is at lowest address of allocated space.

```
int grid[10];
```

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

LC-3 Code for Array References

```
; x = grid[3] + 1
???: where is grid[0]
???: R1 = grid[3]
???: plus 1
STR ??? ; store into x
```

```
Reference
```

```
R5
```
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];
  }`

Recall

```c
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!

What happens?

```c
int ia[6];
ia[8] = 84;
```

Address calculation:
`8 * sizeof(*ia) + ia`

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

Stack Smashing

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

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

Multi-Dimensional Arrays

How does a two dimensional array work?

<table>
<thead>
<tr>
<th>Type</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Number of Rows
Number of Columns

Declaration

int ia[3][4];

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

How would you store it?
How would you store it?

Column Major Order

<table>
<thead>
<tr>
<th></th>
<th>Column 0</th>
<th>Column 1</th>
<th>Column 2</th>
<th>Column 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0,0</td>
<td>0,1</td>
<td>0,2</td>
<td>0,3</td>
<td></td>
</tr>
<tr>
<td>1,0</td>
<td>1,1</td>
<td>1,2</td>
<td>1,3</td>
<td></td>
</tr>
<tr>
<td>2,0</td>
<td>2,1</td>
<td>2,2</td>
<td>2,3</td>
<td></td>
</tr>
</tbody>
</table>

Row Major Order

<table>
<thead>
<tr>
<th></th>
<th>Row 0</th>
<th>Row 1</th>
<th>Row 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0,0</td>
<td>0,1</td>
<td>0,2</td>
<td></td>
</tr>
<tr>
<td>1,0</td>
<td>1,1</td>
<td>1,2</td>
<td></td>
</tr>
<tr>
<td>2,0</td>
<td>2,1</td>
<td>2,2</td>
<td></td>
</tr>
</tbody>
</table>

Advantage

- Using Row Major Order allows visualization as an array of arrays

```
ia[1]

0,0 0,1 1,0 1,1 1,2 1,3 2,0 2,1 2,2 2,3
```

Element Access

- Given a row and a column index
- How to calculate location?
- To skip over required number of rows:
  \[
  \text{row_index} \times \text{sizeof(row)}
  \]
  \[
  \text{row_index} \times \text{Number_of_columns} \times \text{sizeof(arr_type)}
  \]

- This plus address of array gives address of first element of desired row
- Add \text{column_index} \times \text{sizeof(arr_type)} to get actual desired element

```
Element_Address = 
Array_Address +
Row_Index \times \text{Num_Columns} \times \text{sizeof(Arr_Type)} +
Column_Index \times \text{sizeof(Arr_Type)}
```

```
Element_Address = 
Array_Address +
(Row_Index \times \text{Num_Columns} + Column_Index) \times \text{sizeof(Arr_Type)}
```
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

How does C store arrays

• Row major
  ▶ Pointer arithmetic stays unmodified

• Remember this.....
  ▶ Affects how well your program does when you access memory

Now think about

• A 3D array

int a

Now think about

• A 3D array

int a[5]
Now think about

- A 3D array

\[
\text{int } a[4][5]
\]

Offset to \(a[i][j][k]\)?

- A 3D array

\[
\text{int } a[3][4][5]
\]

\[
\text{offset} = (i \times \text{rows} \times \text{columns}) + (j \times \text{columns}) + k
\]

Next: Data 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