Functions, Arrays, Pointers

Recall: 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`
Executing the Swap Function

Before call:

<table>
<thead>
<tr>
<th>R6</th>
<th>3</th>
<th>a</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>4</td>
<td>b</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>valueB</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>valueA</td>
</tr>
</tbody>
</table>

void Swap(int a, int b) {
    int temp = a;
    a = b;
    b = temp;
    return;
}

After call:

<table>
<thead>
<tr>
<th>R6</th>
<th>3</th>
<th>a</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>4</td>
<td>b</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>valueB</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>valueA</td>
</tr>
</tbody>
</table>

These values changed...

...but these did not.

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

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
```

Address | Contents
--------|--------
0x4000  | X      
0x4001  | X      
0x4002  | X      
...     | ...    
0x4007  | X      
0x4008  | X      
0x4009  | X      

On LC-3:
- 10 "16-bit" slots
- Note: can't assume initialized to 0

Just a label for memory location 0x4000

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

```
my_array
```

Address | Contents
--------|--------
0x4000  | X      
0x4001  | X      
0x4002  | 4      
...     | ...    
0x4007  | X      
0x4008  | X      
0x4009  | 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
Points

- 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

- 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

- 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 *
  - Confused? Chapter 16 in Patt/Patel is outstanding!

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

- Dereferencing – fancy word for: contents at address
  - Dereferencing pointer b means:
    - get contents of memory at the address b is pointing to
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>my_array</td>
</tr>
<tr>
<td></td>
<td>.BLKW #10</td>
</tr>
<tr>
<td>int *my_ptr;</td>
<td>LEA R0, my_array</td>
</tr>
<tr>
<td>my_ptr = my_array;</td>
<td>; R0 is equiv to</td>
</tr>
<tr>
<td></td>
<td>; “my_ptr”</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`
- `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[3] = '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!

- Download test3.c – do NOT run the code
  - inc exercises
- Read through code and write down answers to the questions
- Next, download test3.c
  - From lectures page, inclass-nov5 link
- Next, run the code on shell and check your answers
  - ssh into shell
  - Compile and run
Next….C to Assembly Translation-
Pointers & Arrays

- code generation by C compiler for pointer and array variables
  - Translating pointer dereferencing and definitions to LC3 (assembly code)
  - Translating array references to assembly code
- Memory allocation (on run-time stack)

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 *ip;
i = 4;
ptr = &i;
*ptr = *ptr + 1;
```

Example: LC-3 Code

```c
Symbol Table: i is 1st local (offset = 0), ptr is 2nd (offset = -1)
```

```c
*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
```
Example: LC-3 Code

- 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;
  - ADD R0, R5, #0; R0 = R5 + 0 (addr of i)
  - STR R0, R5, #1; store in ptr
  - *ptr = *ptr + 1;
  - LDR R0, R5, #0; R0 = ptr
  - LDR R1, R5, #0; load contents (*ptr)
  - ADD R1, R1, #1; add one
  - STR R1, R5, #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.

```c
void NewSwap(int *a, int *b)
{
    int tempVal = *a;
    *a = *b;
    *b = tempVal;
}
```

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

Swap

- a function that will swap two integers

```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);
  ```
```c
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;
}

---

### Passing Pointers

- **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

### 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
- `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?
  
  \[ 6 \times \text{sizeof}(\text{int}) \]
- Also creates \texttt{ia} which is effectively a constant pointer to the first of the six integers
- What does \texttt{ia[4]} mean?

\[
\text{ia} \quad \text{ia[4]}
\]

sizeof

- Compile time operator
- Two forms
  
  \[
  \text{sizeof object} \\
  \text{sizeof ( type name )}
  \]
- 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

\[
\text{sizeof}
\]

- if \text{sizeof}(\text{int}) == 4 then \text{sizeof}() == 4
- On a typical 32 bit machine...
  
  \[
  \begin{align*}
  \text{sizeof(\text{*ip})} & \rightarrow 4 \\
  \text{sizeof(\text{ip})} & \rightarrow 4 \\
  \text{char \text{*cp}} & \\
  \text{sizeof(char)} & \rightarrow 1 \\
  \text{sizeof(\text{*cp})} & \rightarrow 1 \\
  \text{sizeof(\text{cp})} & \rightarrow 4
  \end{align*}
  \]
  
  \[
  \begin{align*}
  \text{int \text{ia}[6];} \\
  \text{sizeof(\text{ia})} & \rightarrow 24
  \end{align*}
  \]
Arrays

int ia[6];

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

Pointer Arithmetic

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

Arrays

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 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:
  - \texttt{double x[10];}
  - \texttt{double \*y = x;}
  - \texttt{\*(y + 3) = 13;}  
    - \texttt{allocates 20 words (2 per element)}
  - \texttt{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.

```
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.)*

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);
    }
}
```

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] */
```

Array as a Local Variable

```
int foo(int myarray[])
{
    int grid[10];
    ...
}
```
Array as a Local Variable

- Array elements are allocated as part of the activation record.
- \texttt{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].

LC-3 Code for Array References

- \texttt{x = grid[3] + 1}
- \texttt{??? ; where is grid[0]}
- \texttt{??? ; R1 = grid[3]}
- \texttt{??? ; plus 1}
- \texttt{STR ??? ; store into x}

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.
  - \texttt{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.
  - \texttt{void SomeFunction(int num_elements) {
    int temp[num_elements];
  }}

Recall

\texttt{int ia[6];
ia[2] = 42;}

Address calculation:
\texttt{2 * sizeof(*ia) + ia}
Access is by dereferencing
\texttt{* (2 * sizeof(*ia) + ia)
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];

    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
Declaration

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

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

**How does a two-dimensional array work?**

![Diagram of a two-dimensional array]

- **Column Major Order**
  - Column 0: 0,0, 1,0, 2,0, 3,0
  - Column 1: 0,1, 1,1, 2,1, 3,1
  - Column 2: 0,2, 1,2, 2,2, 3,2
  - Column 3: 0,3, 1,3, 2,3, 3,3

- **Row Major Order**
  - Row 0: 0,0, 1,0, 2,0, 3,0
  - Row 1: 0,1, 1,1, 2,1, 3,1
  - Row 2: 0,2, 1,2, 2,2, 3,2

**How would you store it?**

**Advantage**

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

```
int 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
```

```
int 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
```
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 column_index \times \text{sizeof(arr_type)} to get actual desired element

**Element Access**

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

What if array is stored in Column Major Order?

\[
\text{Element_Address} = \text{Array_Address} + (\text{Column_Index} \times \text{Num_Rows} + \text{Row_Index}) \times \text{Sizeof(Arr_Type)}
\]

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

int a[4][5]

Now think about

- A 3D array

int a[3][4][5]
Offset to \(a[i][j][k]\)?

- A 3D array

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

Recall

- One Dimensional Array
  - \(\text{int ia}[6]\);
  - Address of beginning of array: \(\text{ia} \equiv \&\text{ia}[0]\)

- Two Dimensional Array
  - \(\text{int ia}[3][6]\);
  - Address of beginning of array: \(\text{ia} \equiv \&\text{ia}[0][0]\)
  - Also
  - Address of row 0: \(\text{ia}[0] \equiv \&\text{ia}[0][0]\)
  - Address of row 1: \(\text{ia}[1] \equiv \&\text{ia}[1][0]\)
  - Address of row 2: \(\text{ia}[2] \equiv \&\text{ia}[2][0]\)

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

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
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.

Defining a Struct

- We first need to define a new type for the compiler and tell it what our struct looks like.

```c
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.

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.flightNum[0]  plane.flightNum[7]  plane.altitude  plane.longitude  plane.latitude  plane.heading  plane.airspeed
```

A struct's members are laid out in the order specified by the definition.

Defining and Declaring at Once

- You can both define and declare a struct at the same time.

```
struct flightType {  /* max 6 characters */
    char flightNum[7];
    int altitude;
    int longitude; /* in tenths of degrees */
    int latitude; /* in tenths of degrees */
    int heading; /* in tenths of degrees */
    double airSpeed; /* in km/hr */
};
```

And you can use the `flightType` name to declare other structs:
```
struct flightType iceMan;
```
**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 runtime 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 Allocation