Chapter 16

Why do we need Pointers?
Call by Value vs. Call by Reference *in detail*
Implementing Arrays
Buffer Overflow / The “Stack Hack”
A problem with parameter passing via stack

- Consider the following function that's designed to swap the values of its arguments.

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

- int main () {
  int valueA = 3, valueB = 4;
  ...
  Swap (valueA, valueB);
  ...
}
Executing the Swap Function

before call

R6

<table>
<thead>
<tr>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>3</td>
</tr>
</tbody>
</table>

firstVal
secondVal
valueB
valueA

main

after call

Swap

R6

<table>
<thead>
<tr>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>3</td>
</tr>
</tbody>
</table>

tempVal

<table>
<thead>
<tr>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>3</td>
</tr>
</tbody>
</table>

firstVal
secondVal
valueB
valueA

These values changed...

...but these did not.
Pointers and Arrays

- Functions such as the swap example need to be able access variables stored in memory locations outside of their own activation record
  - A function’s activation record defines its “scope”
  - We've seen examples of how to do this in Assembly.

- 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 (still a pointer!)
  - An area of allocated memory with values arranged sequentially
  - Expression `a[4]` refers to the 5th element of the array `a`
    - The array variable is a pointer to the base of the array
    - Base + offset
    - Thus... the first element is 0
Pointers in C

- C lets us talk about and manipulate addresses as “pointer variables”
- But first, lets refresh the somewhat confusing bits.

- &: The “address-of” or “reference” operator
  - This operator does one thing.
  - It returns the address of the variable which follows

```c
#include <stdio.h>
int main() {
    int x = 0;
    printf("Address of x ");
    printf("= 0x%p \n", &x);
    return 0;
}
```

Output: Address of x = 0x0065FDF4
Pointers in C

- How do we store addresses? Pointer variables!
  - Although all pointers in C are exactly the same type (address) they are also typed by the compiler so that the data to which they refer can be appropriately interpreted.
  - A pointer in C is always a pointer to a particular data type: int*, double*, char*, etc.

- Declaration
  - `int *p; /* p is a pointer to an int */`

- Operators
  - `*p` -- returns the value pointed to by p (indirect address / dereference op)
  - `&z` -- returns the address of variable z (address-of operator)

- Important point of common confusion!
  - `*` means “a pointer variable” when used in a declaration
  - `*` means “access the information that this address points to” elsewhere
  - What does *3 mean?
#include <stdio.h>

int main() {
    int x = 12;
    int *ptr = &x;

    printf("Address of x: \t0x%p\n", ptr);
    printf("Address of x: \t0x%x\n", &x);
    printf("Address of ptr: \t0x%x\n", &ptr);
    printf("Value of x: \t%d\n", *ptr);

    return 0;
}

Address of x: 0x0065FDF4
Address of x: 0x65fdf4
Address of ptr: 0x65fdf0
Value of x: 12


```c
#include <stdio.h>

int main() {
    int x[10] = {0,1,2,3,4,5,6,7,8,9};
    printf("Address of x[0]: \t 0x%p\n", &x[0]);
    printf("Address of x: \t 0x%p\n", x);
    printf("Value of x[0]: \t %d\n", x[0]);
    printf("Value of x[0]: \t %d\n", *x);

    return 0;
}
```

Address of x[0]: 0x0065FDD0
Address of x: 0x0065FDD0
Value of x[0]: 0
Value of x[0]: 0
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

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Offset</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>int</td>
<td>0</td>
<td>main</td>
</tr>
<tr>
<td>ptr</td>
<td>int*</td>
<td>-1</td>
<td>main</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>R6</th>
<th>xEFFC</th>
</tr>
</thead>
<tbody>
<tr>
<td>R5</td>
<td>4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ptr</th>
<th>i</th>
<th>fp</th>
<th>pc</th>
<th>ret</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>xxxx</td>
<td>xxxx</td>
<td>xxxx</td>
<td>xxxx</td>
</tr>
</tbody>
</table>
Call by reference

- Passing a pointer as an argument allows the function to read/change memory outside its activation record.
  - But not the pointer itself!
  - If you wanted to change the pointer itself, what would you need to do?

```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.**
Passing Pointers to a Function

main() wants to swap the values of valueA and valueB

NewSwap(&valueA, &valueB);

LC-3 Code for main:

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
Code Using Pointers

- Inside the NewSwap routine
  - \[\text{; int tempVal = *firstVal;}\]
  - \[\text{LDR R0, R5, \#4 ; R0=xEFFA}\]
  - \[\text{LDR R1, R0, \#0 ; R1=M[xEFFA]=3}\]
  - \[\text{STR R1, R5, \#0 ; tempVal=3}\]
  - \[; *firstVal = *secondVal;\]
  - \[\text{LDR R1, R5, \#5 ; R1=xEFF9}\]
  - \[\text{LDR R2, R1, \#0 ; R1=M[xEFF9]=4}\]
  - \[\text{STR R2, R0, \#0 ; M[xEFFA]=4}\]
  - \[; *secondVal = tempVal;\]
  - \[\text{LDR R2, R5, \#0 ; R2=3}\]
  - \[\text{STR R2, R1, \#0 ; M[xEFF9]=3}\]

```assembly
LDR R0, R5, #4 ; R0=xEFFA
LDR R1, R0, #0 ; R1=M[xEFFA]=3
STR R1, R5, #0 ; 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
```
Pointer arithmetic

- If ptr is a pointer, what is ptr = ptr + 1?
  - It depends on the size of the data pointed to by ptr!!

- On a x86 (byte addressable) machine
  - int* ptr;   ptr = ptr + 1;  /* ptr increases by 2 */
  - long* ptr;  ptr = ptr + 1;  /* ptr increases by 4 */

- Sometimes we want a pointer that points to nothing.
  - In other words, we declare a pointer, but have nothing to point to (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 0x00000000 is not a legal address for most programs on most platforms BUT it depends on the machine!
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);

- But what about:
  - scanf("%s", &buffer);  ??

read a decimal integer and store in dataIn
Arrays

- **Declaration**
  - `type  variableName [size];`
  - All elements are the same type/size
  - Number of elements known at compile time

- **Reference**
  - `variableName [num]`
  - Accesses num-th – 1 entry in array
  - Numbering starts from 0
  - No limit checking at compile- or run-time

- Array elements can be allocated as part of the activation record (local) or in global memory

- `int grid[10];`
  - First element (grid[0]) is at lowest address of allocated space
  - If grid is only local variable allocated, then R5 will point to grid[9]
Array code for LC-3

; 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

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Offset</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>grid</td>
<td>int*</td>
<td>-9</td>
<td>main</td>
</tr>
<tr>
<td>x</td>
<td>int</td>
<td>-10</td>
<td>main</td>
</tr>
</tbody>
</table>
More Array code for LC-3

更多的数组代码

; 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 = &grid[x+1]
STR R2, R1, #0 ; grid[x+1] = R2

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Offset</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>grid</td>
<td>int</td>
<td>-9</td>
<td>main</td>
</tr>
<tr>
<td>x</td>
<td>int*</td>
<td>-10</td>
<td>main</td>
</tr>
</tbody>
</table>
Arrays are Pointers

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

- Note that you CAN change cptr, but you CANNOT change word.
  - What is the difference between them?

- 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]
Multi-dimensional arrays

- How do you layout a multi-dimensional array in one-dimensional memory?
  - Array layout is critical for correctly passing arrays between programs written in different languages.
  - It is also important for performance when traversing an array because accessing array elements that are contiguous in memory is usually faster than accessing elements which are not, due to caching.

- Row-major order
  - In row-major storage, a multidimensional array in linear memory is accessed such that rows are stored one after the other.
  - \{ \{1, 2, 3\}, \{4, 5, 6\} \} is stored 1, 2, 3, 4, 5, 6
  - offset = row*NUMCOLS + column

- Column-major order

- Row-major order is used in C, C++, Java, and most modern languages.
- Column-major order is used in Fortran and Matlab.
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);
    ...
}

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

Note: Size of static array must be known at compile time, so MAX_NUMS must be a constant.
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';
  ...
  outputString[9] = '\0';
  ```
I/O with Strings

- Printf and scanf use "%s" format character for string

- **Printf** -- print characters up to terminating zero
  
  ```c
  printf("%s", outputString);
  ```

- **Scanf** -- read characters until whitespace, store result in string, and terminate with zero
  
  ```c
  scanf("%s", inputString);
  ```
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.

  ```c
  int array[10];
  int i;
  for (i = 0; i <= 11; i++) array[i] = 0;
  ```

- **What will happen?**
  - Think about the activation record!

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

  ```c
  void SomeFunction(int num_elements) {
      int temp[num_elements];
      ...
  }
  ```
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:
  - double x[10];
  - double *y = x;
  - *(y + 3) = 13;

allocates 20 words (2 per element)

same as x[3] -- base address plus 6
How important is understanding memory?

- C does not enforce memory safety
  - Many ways to access memory illegally
    - Accessing an array out of bounds
    - Bad pointer arithmetic manipulations
- Some memory bugs come into play only rarely
  - When manipulating large files/strings, etc.
    - Accessing an array out of bounds
    - Bad pointer arithmetic manipulations
- Crash errors: Program accesses illegal memory (SEG Fault)
  - OK for user programs, not so good for OS programs
- Non-crash errors: Strange glitches, “magic” results
- Intentional exploits: These bugs are repeatable and can be exploited to cause great harm
Consider: How can memory be exploited?

```c
void read_array() {
    int array[6];
    int index;
    int hex;

    index = 0;
    do {
        hex = getHexInput();
        array[index] = hex;
        index++;
    } while (hex != 0);
    // code to manipulate data
}
```
Consider: Prologue

```c
void read_array() {
    int array[6];
    int index;
    int hex;

    index = 0;
    do {
        hex = getHexInput();
        array[index] = hex;
        index++;
    } while (hex != 0);
    // code to manipulate data
}
```

```assembly
.ORIG x3000
READ_AR ADD R6, R6, #-1 ; push return
    ADD R6, R6, #-1 ; push ret link
    STR R7, R6, #0
    ADD R6, R6, #-1 ; push frame ptr
    STR R5, R6, #0 ;
    ADD R5, R6, #-1 ; set frame ptr
    ADD R6, R6, #-6 ; int array[6] (#0-#-5)
    ADD R6, R6, #-1 ; int index (#-6)
    ADD R6, R6, #-1 ; int hex (#-7)
    ADD R6, R6, #-1 ; Callee save R0 (#-8)
    STR R0, R5, #-8
    ADD R6, R6, #-1 ; Callee save R1 (#-9)
    STR R1, R5, #-9
    AND R0, R0, #0 ; index = 0
    STR R0, R5, #-1 ;
```
Consider: Body

```c
void read_array() {
    int array[6];
    int index;
    int hex;

    index = 0;
    do {
        hex = getHexInput();
        array[index] = hex;
        index++;
    } until (hex == 0);
    // code to manipulate data
}
```

```assembly
    ... NEXT    TRAP x40 ; hex = getHexInput()
    STR R0, R5, #7 ;
    ; array[index] = hex
    LDR R0, R5, #6 ; R0 <- index
    ADD R1, R5, #5 ; R1 <- &array[0]
    ADD R1, R1, R0 ; R1 <- &array[index]
    LDR R0, R5, #7 ; R0 <- hex
    STR R0, R1, #0 ; array[index] = hex
    LDR R1, R5, #6 ; index++
    ADD R1, R1, #1
    STR R1, R5, #6
    LDR R0, R5, #7 ; until (hex = 0)
    BRnp NEXT
    ...
Consider: Epilogue

```c
void read_array() {
    int array[6];
    int index;
    int hex;

    index = 0;
    do {
        hex = getHexInput();
        array[index] = hex;
        index++;
    } while (hex != 0);
    // code to manipulate data
}
```

```assembly
... LDR R1, R5, #-9 ; Callee restore R1
ADD R6, R6, #1
LDR R0, R5, #-8 ; Callee restore R0
ADD R6, R6, #1

ADD R6, R6, #8 ; Pop locals
LDR R5, R6, #0 ; pop frame ptr
ADD R6, R6, #1
LDR R7, R6, #0 ; pop ret link
ADD R6, R6, #1
RET
```

.END
READ_AR
ADD R6, R6, #-1 ; push return
ADD R6, R6, #-1 ; push ret link
STR R7, R6, #0
ADD R6, R6, #-1 ; push frame ptr
STR R5, R6, #0 ;
ADD R5, R6, #-1 ; set frame ptr
ADD R6, R6, #-6 ; int array[6] (#0-#-5)
ADD R6, R6, #-1 ; int index (#-6)
ADD R6, R6, #-1 ; int hex (#-7)
ADD R6, R6, #-1 ; Callee save R0 (#-8)
STR R0, R5, #-8
ADD R6, R6, #-1 ; Callee save R1 (#-9)
STR R1, R5, #-9
AND R0, R0, #0 ; index = 0
STR R0, R5, #-6 ;

NEXT
TRAP x40 ; hex = getHexInput()
STR R0, R5, #-7 ;
; array[index] = hex
LDR R0, R5, #-6 ; R0 <- index
ADD R1, R5, #-5 ; R1 <- &array[0]
ADD R1, R1, R0 ; R1 <- &array[index]
LDR R0, R5, #-7 ; R0 <- hex
STR R0, R1, #0 ; array[index] = hex
LDR R1, R5, #-6 ; index++
ADD R1, R1, #1 ;
STR R1, R5, #-6 ;

LDR R0, R5, #-7 ; until (hex = 0)
BRnp NEXT
Stack hack (Smashing the Stack)

- Exploit a buffer or array overrun
  - Overwrite return link, inject code (processor specific), and take over machine
- What if the OS can set the stack to be non-executable?

```java
void checkAccess() {
    boolean valid = check_password();
    if (valid) {
        giveAccess();
    } else {
        denyAccess();
    }
} /* end checkAccess */
```
String Library Code

- Implementation of C/C++ function `gets`
  - No way to specify limit on number of characters to read

```c
/* Get string from stdin */
char *gets(char *dest)
{
    int c = getc();
    char *p = dest;
    while (c != EOF && c != '\n') {
        *p++ = c;
        c = getc();
    }
    *p = '\0';
    return dest;
}
```

- Similar problems with other C/C++ functions
  - `strcpy`, `scanf`, `fscanf`, `sscanf`, when given `%s` conversion specification
  - `cin` (as commonly used in C++)
Vulnerable Buffer Code

```c
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    gets(buf);
    puts(buf);
}

int main()
{
    printf("Type a string:");
    echo();
    return 0;
}
```
Buffer Overflow Executions

unix>./bufdemo
Type a string: 123
123

unix>./bufdemo
Type a string: 12345
Segmentation Fault

unix>./bufdemo
Type a string: 12345678
Segmentation Fault
Malicious Use of Buffer Overflow

- Input string contains byte representation of executable code
- Overwrite return address with address of buffer
- When \texttt{bar()} executes \texttt{ret}, will jump to exploit code

```c
void bar() {
    char buf[64];
    gets(buf);
    ...
}

void foo() {
    bar();
    ...
}
```
Exploits Based on Buffer Overflows

- **Buffer overflow bugs allow remote machines to execute arbitrary code on victim machines.**

- Internet worm
  - Early versions of the finger server (fingerd) used `gets()` to read the argument sent by the client:
    - `finger droh@cs.cmu.edu`
  - Worm attacked fingerd server by sending phony argument:
    - `finger "exploit-code padding new-return-address"`
    - Exploit code: executed a root shell on the victim machine with a direct TCP connection to the attacker.

- IM War
  - AOL exploited existing buffer overflow bug in AIM clients
  - Exploit code: returned 4-byte signature (the bytes at some location in the AIM client) to server.
  - When Microsoft changed code to match signature, AOL changed signature location.
Code Red Worm

- **History**
  - June 18, 2001. Microsoft announces buffer overflow vulnerability in IIS Internet server
  - July 19, 2001. Over 250,000 machines infected by new virus in 9 hours
  - White house must change its IP address. Pentagon shut down public WWW servers for day

- **When We Set Up CS:APP Web Site**
  - Received strings of form

```
GET
/default.ida?NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
u9090%u6858%ucbd3%u7801%u9090%u6858%ucbd3%u7801%u9090%u9090%u8190%u00c3%u0003%u8b00%u531b%u53ff%u0078%u0000%u00=a
HTTP/1.0" 400 325 "-" "-
```
Code Red Exploit Code

- Starts 100 threads running
- Spread self
  - Generate random IP addresses & send attack string
  - Between 1st & 19th of month
- Attack www.whitehouse.gov
  - Send 98,304 packets; sleep for 4-1/2 hours; repeat
    - Denial of service attack
  - Between 21st & 27th of month
- Deface server’s home page
  - After waiting 2 hours

Welcome to http://www.worm.com!

Hacked By Chinese!
iphone hack

- Exploits stack overflow of jpeg display dll (which runs as root)
- What does this mean about the safety of viewing images, in general?
- The Safety of using library code, in general?
- What are your ethical responsibilities?
Avoiding Overflow Vulnerability

- Use Library Routines that Limit String Lengths
  - Use "%ns" NOT "%s"
    - Why %3 above?
  - Do NOT use cin in C++

```c
/* Echo Line */
void echo()
{
    char buf[4]; /* Way too small! */
    scanf("%3s",&buf);
    printf("%s",buf);
}
```