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

<table>
<thead>
<tr>
<th>R6</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>R7</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>R8</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>main</td>
<td>valueA</td>
<td>valueB</td>
</tr>
</tbody>
</table>

after call

<table>
<thead>
<tr>
<th>R6</th>
<th>3</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>R7</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>R8</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>Swap</td>
<td>tempVal</td>
<td>firstVal</td>
</tr>
</tbody>
</table>

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
    - The array variable is a pointer to the base of the array
    - `Base + offset`
    - Thus, the first element is at

Pointers in C

- C lets us talk about and manipulate addresses as "pointer variables"
- But first, let's 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 = 0x%p\n", &x);
    return 0;
}
```

Output: Address of x = 0x0065FDF4

- 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?

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!
  - "p" means "a pointer variable" when used in a declaration
  - "p" means "access the information that this address points to" elsewhere
  - What does "3" mean?
# Check for understanding

```c
#include <stdio.h>

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

    printf("Address of x: \0x%p\n", ptr);
    printf("Address of x: \0x%x\n", &x);
    printf("Address of ptr: \0x%x\n", &ptr);
    printf("Value of x: \0%d\n", *ptr);

    return 0;
}
```

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

# Check for understanding

```c
#include <stdio.h>

int main() {
    int x[10] = {0,1,2,3,4,5,6,7,8,9};

    printf("Address of x[0]: \0x%p\n", &x[0]);
    printf("Address of x: \0x%p\n", x);
    printf("Value of x[0]: \0%d\n", x[0]);
    printf("Value of x[0]: \0%d\n", *x);

    return 0;
}
```

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

## Example

```c
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 = 4;
ADD R0, R0, #4 ; put 4 in R0
STR R0, R5, #0 ; store in i
:ptr = R5;  
ADD R0, R5, #0 ; R0 = R5 + 0 (addr of i)
ADD R6, R6, #0 ; push
STR R0, R6, #0
ADD R1, R1, #1 ; add one
LDR R1, R0, #0 ; load contents (*ptr)
ADD R6, R6, #0 ; store result
```

### Name  Type  Offset  Scope
- i  int  0  main
- ptr  int* -1  main
- R6  R6
- R5  R5
- R4  R4
- R3  R3
- pc  pc

## 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

```
void NewSwap(int *firstVal, int *secondVal) {
    int tempVal = *firstVal;
    *firstVal = *secondVal;
    *secondVal = tempVal;
}
```

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

```
NewSwap(&valueA, &valueB);
```

### First-3 Code for main:

```
ADD R0, R6, #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
```

### x0000

- R6
- R5
- R4
- R3
- pc
- fp
- sp
- x10FD
Inside the NewSwap routine

Pointer arithmetic
It depends on the size of the data pointed to by ptr!!

First element (grid[0]) is at lowest address
or run

Declaration
int* ptr; ptr = ptr + 1; /* ptr increases by 2 */

In other words, we declare a pointer, but have nothing to point to (yet).

Reference
Sometimes we want a pointer that points to nothing.

Offset
Accesses num
useful for multiple results

Scope

- variableName [num]

Type
Pass address of variable where you want result stored

Type
On a x86 (byte addressable) machine

But what about:
If ptr is a pointer, what is ptr = ptr + 1?

Number of elements known at compile time

type  variableName [size];

scanf("%d ", &dataIn);

This solves the "mystery" of why '&' with argument to scanf:

int grid[10];

Numbering starts from 0

LDR  R2, R5, #0 ; R2=3

If grid is only local variable allocated, then R5 will

int* p = NULL; /* p is a null pointer */

Often NULL = 0, because Address 0x00000000 is not a legal address

for most programs on most platforms BUT it depends on the machine!

Array code for LC-3

array code for LC-3

Name Type Offset Scope

grid x int 9 main

x = grid[3] + 1
ADD R0, R5, #9 ; R0 = grid[0]

LDR R1, R0, #3 ; R1 = grid[3]

STR R1, R5, #10 ; x = R1

grid[0]
grid[1]
grid[2]
grid[3]

grid[4]
grid[5]
grid[6]
grid[7]
grid[8]
grid[9]
grid[10]

grid[11]
grid[12]

grid[13]
grid[14]
grid[15]

grid[x+1] = grid[x] + 2
ADD R0, R5, #10 ; R0 = x

ADD R1, R0, #9 ; R1 = grid[0]

LDR R2, R1, #10 ; R2 = grid[x]

ADD R2, R5, #2 ; add 2

ADD R0, R5, #10 ; R0 = x

ADD R5, R0, #5 ; R5 = 5

ADD R0, R5, #9 ; R1 = grid[0]

ADD R1, R5, #10 ; R2 = grid[x]

ADD R2, R0, #3 ; add 3

ADD R2, R5, #10 ; R2 = grid[x]

ADD R1, R5, #9 ; R1 = grid[0]

ADD R2, R0, #3 ; add 3

ADD R2, R5, #10 ; R2 = grid[x]

ADD R1, R5, #9 ; R1 = grid[0]

ADD R2, R0, #3 ; add 3

ADD R2, R5, #10 ; R2 = grid[x]

ADD R1, R5, #9 ; R1 = grid[0]

ADD R2, R0, #3 ; add 3

ADD R2, R5, #10 ; R2 = grid[x]

ADD R1, R5, #9 ; R1 = grid[0]

ADD R2, R0, #3 ; add 3

ADD R2, R5, #10 ; R2 = grid[x]
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 4word[0]
  - (cptr + n) word + n 4word[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.
  - [(0, 2, 3), (4, 5, 6)] is stored 0, 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
main() {
  int numbers[MAX_NUMS];
  mean = Average(numbers);
  …
} int Average(int inputValues[MAX_NUMS]) {
  …
  return (sum / MAX_NUMS);
  …
}

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

- Print and scanf use "%s" format character for string
- Print -- print characters up to terminating zero
  printf("%s", outputString);
- Scanf -- read characters until whitespace, store result in string, and terminate with zero
  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.
  - int array[10];
    int i;
    for (i = 0; i <= 10; 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.
  - 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;`

**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
  - It's ok, because we've only shown code for array[6];

**Consider: Body**

```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: Epilogue**

```c
void read_array() {
    LDR R5, R6, #0 ; Callee return
    ADD R6, R6, #1
    STR R0, R5, #0
    .END
    ; Callee
}
```

**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
}
```
No way to specify limit on number of characters to read
Implementation of C/C++ function
\texttt{hex} (as commonly used in C++)
\begin{verbatim}
&array[0]
\end{verbatim}
When
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?
\begin{verbatim}
void checkAccess()
    ... denyAccess();
    ... giveAccess();
    ... check_password();
\end{verbatim}

\textbf{String Library Code}
- Implementation of C/C++ function \texttt{gets}:
  - No way to specify limit on number of characters to read
  - Similar problems with other C/C++ functions
  - \texttt{scanf}, \texttt{sscanf}, when given valid conversion specification
  - \texttt{dn} (anomaly used in C/C++)

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

\textbf{Vulnerable Buffer Code}

\textbf{Buffer Overflow Executions}
unix>./bufdemo
Type a string:
123
unix>./bufdemo
Type a string:12345
Segmentation Fault
unix>./bufdemo
Type a string:12345678
Segmentation Fault

\textbf{Malicious Use of Buffer Overflow}
\begin{verbatim}
void foo()
{
    char buf[64];
    gets(buf);
    ... }
\end{verbatim}

\begin{verbatim}
bar() {
    ... bar();
    ... }
\end{verbatim}

\begin{verbatim}
void bar()
{
    char buf[64];
    gets(buf);
    ... }
\end{verbatim}

- Input string contains byte representation of executable code
- Overwrite return address with address of buffer
- When \texttt{bar} executes \texttt{int}, will jump to exploit code
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:
    ```
    /etc/fingerd/fingerd
    
    finger "exploit-code padding new-return-address" 
    ```
  - Attack worm attacked fingerd server by sending phony argument:
    ```
    finger droh@cs.cmu.edu
    
    finger "exploit-code padding new-return-address"
    ```
- IM War
  - AOL exploited existing buffer overflow bug in AIM clients
  - `gets()` read argument, exploited code: executed a root shell on the victim machine with a direct TCP connection to the attacker.

Code Red Worm

- History
  - June 20, 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
    ```
    http://<domain>/default.ida?NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN....NNNNNNNNN
    ```
  - HTTP/1.0 400 325 "Not a HTTP request"

- 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

Avoiding Overflow Vulnerability

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

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