CS 61C: Great Ideas in Computer Architecture

C Arrays, Strings and Memory Management

Instructors:
Nicholas Weaver & Vladimir Stojanovic

http://inst.eecs.Berkeley.edu/~cs61c/sp16
Reminder!

C Arrays are Very Primitive

- An array in C does not know its own length, and its bounds are not checked!
  - Consequence: We can accidentally access off the end of an array
  - Consequence: We must pass the array \textit{and its size} to any procedure that is going to manipulate it

- Segmentation faults and bus errors:
  - These are VERY difficult to find; be careful! (You’ll learn how to debug these in lab)
  - But also “fun” to exploit:
    - “Stack overflow exploit”, maliciously write off the end of an array on the stack
    - “Heap overflow exploit”, maliciously write off the end of an array on the heap

- If you write programs in C, you \textit{will} write code that has array-bounds errors!
Use Defined Constants

• Array size $n$; want to access from 0 to $n$-1, so you should use counter AND utilize a variable for declaration & incrementation
  – Bad pattern
    ```c
    int i, ar[10];
    for(i = 0; i < 10; i++){ ... }
    ```
  – Better pattern
    ```c
    const int ARRAY_SIZE = 10;
    int i, a[ARRAY_SIZE];
    for(i = 0; i < ARRAY_SIZE; i++) { ... }
    ```

• SINGLE SOURCE OF TRUTH
  – You’re utilizing indirection and avoiding maintaining two copies of the number 10
  – DRY: “Don’t Repeat Yourself”
  – And don’t forget the < rather than <=:
    When Nick took 60c, he lost a day to a “segfault in a malloc called by printf on large inputs”: Had a <= rather than a < in a single array initialization!
When Arrays Go Bad: Heartbleed

• In TLS encryption, messages have a length...
  – And get copied into memory before being processed

• One message was “Echo Me back the following data, its this long…”
  – But the (different) echo length wasn’t checked to make sure it wasn’t too big...

```
M 5 HB L=5000 107:Ou17;GET / HTTP/1.1
Host: www.mydomain.com
Cookie: login=117kf9012oe
User-Agent: Mozilla....
```

• So you send a small request that says “read back a lot of data”
  – And thus get web requests with auth cookies and other bits of data from random bits of memory...
Pointing to Different Size Objects

• Modern machines are “byte-addressable”
  – Hardware’s memory composed of 8-bit storage cells, each has a unique address
• Type declaration tells compiler how many bytes to fetch on each access through pointer
  – E.g., 32-bit integer stored in 4 consecutive 8-bit bytes
• But we actually want “Byte alignment”
  – Some processors will not allow you to address 32b values without being on 4 byte boundaries
  – Others will just be very slow if you try to access “unaligned” memory.
sizeof() operator

• sizeof(type) returns number of bytes in object
  – But number of bits in a byte is not standardized
    • In olden times, when dragons roamed the earth, bytes could be 5, 6, 7, 9 bits long
• By definition, sizeof(char)==1
  – C does not play well with Unicode (unlike Python), so no
    char c = ‘💩’
• Can take sizeof(arg), or sizeof(structtype)
  – Structure types get padded to ensure structures are also aligned
• We’ll see more of sizeof when we look at dynamic memory management
Pointer Arithmetic

\textit{pointer} + \textit{number} \quad \textit{pointer} - \textit{number}

e.g., \textit{pointer} + 1 \quad \text{adds 1 \underline{something} to a pointer}

\begin{align*}
\text{char} & \quad *p; \\
\text{char} & \quad a; \\
\text{char} & \quad b;
\end{align*}

\begin{align*}
p & = & \&a; \\
p & += & 1;
\end{align*}

\begin{align*}
\text{int} & \quad *p; \\
\text{int} & \quad a; \\
\text{int} & \quad b;
\end{align*}

\begin{align*}
p & = & \&a; \\
p & += & 1;
\end{align*}

In each, \( p \) now points to \( b \)

(Assuming compiler doesn’t reorder variables in memory.
\\textit{Never code like this!!!!!})

\begin{align*}
\text{Adds} & \ 1*\text{sizeof(char)} & \text{to the memory address} \\
\text{Adds} & \ 1*\text{sizeof(int)} & \text{to the memory address}
\end{align*}

\textit{Pointer arithmetic should be used cautiously}

(and by Nick’s standard, “cautious” == Almost NEVER!)
Arrays and Pointers

• Array \approx\text{ pointer to the initial (0th) array element}

\[ a[i] \equiv *(a+i) \]

• An array is passed to a function as a pointer
  – The array size is lost!

• Usually bad style to interchange arrays and pointers
  – Avoid pointer arithmetic!

Passing arrays:

\textbf{Really int *array}

\textbf{Must explicitly pass the size}

\begin{verbatim}
int foo(int array[], unsigned int size)
{
  ... array[size - 1] ...
}

int main(void)
{
  int a[10], b[5];
  ... foo(a, 10) ... foo(b, 5) ...
}
\end{verbatim}
Arrays and Pointers

```c
int foo(int array[],
    unsigned int size)
{
    ...
    printf("%d\n", sizeof(array));
}

int main(void)
{
    int a[10], b[5];
    ... foo(a, 10)... foo(b, 5) ...
    printf("%d\n", sizeof(a));
}
```

What does this print? 4
... because `array` is really a pointer (and a pointer is architecture dependent, but likely to be 4 or 8 on modern 32-64 bit machines!)

What does this print? 40
Arrays and Pointers

These code sequences have the same effect!

But the former is much more readable:
Clickers/Peer Instruction Time

```c
int x[] = { 2, 4, 6, 8, 10 };
int *p = x;
int **pp = &p;
(*pp)++;
(*(*pp))++;
printf("%d\n", *p);
```

Result is:
A: 2
B: 3
C: 4
D: 5
E: None of the above
Clickers/Peer Instruction Time

int x[] = { 2, 4, 6, 8, 10 };
int *p = x;
int **pp = &p;
(*pp)++;  // P points to the start of X (2)
(*(*pp))++;  // PP points to P

printf("%d\n", *p);

Result is:
A: 2
B: 3
C: 4
D: 5
E: None of the above
Administrivia

• hw0 - edX is due on Sunday 1/31,
• hw0 mini bio is due in lab next week, turn in to your lab TA
• MT1 will be Thursday, 2/25 from 6-8PM
• MT2 will be Monday, 4/4 from 7-9 PM
  – Email William and Fred if you have conflicts (except for 16B, that is currently being resolved)
61C Instructor in the News...

• Nick presented at the Enigma conference:
  – “The Golden Age of Bulk Surveillance”
  – https://www.youtube.com/watch?v=zqnKdGnzoh0
int strlen(char *s)
{
    char *p = s;
    while (*p++)
        ; /* Null body of while */
    return (p - s - 1);
}

What happens if there is no zero character at end of string?
Point past end of array?

• Array size $n$; want to access from 0 to $n-1$, but test for exit by comparing to address one element past the array

```c
int ar[10], *p, *q, sum = 0;
...
p = &ar[0]; q = &ar[10];
while (p != q)
    /* sum = sum + *p; p = p + 1; */
    sum += *p++;
```

– Is this legal?

• C defines that one element past end of array must be a valid address, i.e., not cause an error
  
  – **BUT DO NOT DO THIS**: This is ONLY valid for arrays declared in this manner, NOT arrays declared using dynamic allocation (malloc)!
Valid Pointer Arithmetic

• Add an integer to a pointer.
• Subtracting an integer from a pointer
• Subtract 2 pointers (in the same array)
• Compare pointers (<, <=, ==, !=, >, >=)
• Compare pointer to NULL (indicates that the pointer points to nothing)

Everything else illegal since makes no sense:
• adding two pointers
• multiplying pointers
• subtract pointer from integer
Arguments in `main()`

• To get arguments to the main function, use:
  – `int main(int argc, char *argv[])`

• What does this mean?
  – `argc` contains the number of strings on the command line (the executable counts as one, plus one for each argument). Here `argc` is 2:
    ```
    unix% sort myFile
    ```
  – `argv` is a *pointer* to an array containing the arguments as strings
Example

- `foo hello 87 "bar baz"
- `argc = 4 /* number arguments */`
- `argv[0] = "foo",
  argv[1] = "hello",
  argv[2] = "87",
  argv[3] = "bar baz",
  — Array of pointers to strings
C Memory Management

• How does the C compiler determine where to put all the variables in machine’s memory?
• How to create dynamically sized objects?
• *To simplify discussion, we assume one program runs at a time, with access to all of memory.*
• *Later, we’ll discuss virtual memory, which lets multiple programs all run at same time, each thinking they own all of memory.*
C Memory Management

• Program’s *address space* contains 4 regions:
  
  – **stack**: local variables inside functions, grows downward
  
  – **heap**: space requested for dynamic data via `malloc();` resizes dynamically, grows upward
  
  – **static data**: variables declared outside functions, does not grow or shrink. Loaded when program starts, can be modified.
  
  – **code**: loaded when program starts, does not change
Where are Variables Allocated?

• If declared outside a function, allocated in “static” storage
• If declared inside function, allocated on the “stack” and freed when function returns
  – main() is treated like a function

```c
int myGlobal;
main() {
    int myTemp;
}
```
The Stack

- Every time a function is called, a new frame is allocated on the stack.
- Stack frame includes:
  - Return address (who called me?)
  - Arguments
  - Space for local variables
- Stack frames uses contiguous blocks of memory; stack pointer indicates start of stack frame.
- When function ends, stack pointer moves up; frees memory for future stack frames.
- We’ll cover details later for MIPS processor.

```c
fooA() {
  fooB();
}
fooB() {
  fooC();
}
fooC() {
  fooD();
}
```
Last In, First Out (LIFO) data structure

```c
main ()
{ a(0);
}

void a (int m)
{ b(1);
}

void b (int n)
{ c(2);
}

void c (int o)
{ d(3);
}

void d (int p)
{
}
```

Stack Animation

Stack Pointer
C supports functions for heap management:

- **malloc()** allocate a block of uninitialized memory
- **calloc()** allocate a block of zeroed memory
- **free()** free previously allocated block of memory
- **realloc()** change size of previously allocated block
  - careful – it might move!
Malloc()

- **void *malloc(size_t n):**
  
  - Allocate a block of uninitialized memory
  - **NOTE:** Subsequent calls probably will not yield adjacent blocks
  - **n** is an integer, indicating size of requested memory block in bytes
  - **size_t** is an unsigned integer type big enough to “count” memory bytes
  - Returns **void*** pointer to block; **NULL** return indicates no more memory
  - Additional control information (including size) stored in the heap for each allocated block.

  
  
  “Cast” operation, changes type of a variable.  
  Here changes (void *) to (int *)

- **Examples:**
  ```
  int *ip;
  ip = (int *) malloc(sizeof(int));
  
  typedef struct { ... } TreeNode;
  TreeNode *tp = (TreeNode *) malloc(sizeof(TreeNode));
  ```

- **sizeof** returns size of given type in bytes, produces more portable code
Managing the Heap

• **void free(void *p):**
  – Releases memory allocated by `malloc()`
  – p is pointer containing the address *originally* returned by `malloc()`
    ```
    int *ip;
    ip = (int *) malloc(sizeof(int));
    .... ....
    free((void*) ip);  /* Can you free(ip) after ip++ ? */
    ```
  ```
  typedef struct {... } TreeNode;
  TreeNode *tp = (TreeNode *) malloc(sizeof(TreeNode));
  .... ....
  free((void *) tp);
  ```
  – When insufficient free memory, `malloc()` returns **NULL** pointer; **Check for it!**

    ```
    if ((ip = (int *) malloc(sizeof(int))) == NULL){
        printf("\nMemory is FULL\n");
        exit(1); /* Crash and burn! */
    }
    ```
  – When you free memory, you must be sure that you pass the **original address** returned from `malloc()` to `free()`; Otherwise, system exception (or worse)!
    – And never use that memory again “use after free”
Using Dynamic Memory

typedef struct node {
    int key;
    struct node *left;
    struct node *right;
} Node;

Node *root = NULL;

Node *create_node(int key, Node *left, Node *right) {
    Node *np;
    if ( (np = (Node*) malloc(sizeof(Node))) == NULL) {
        printf("Memory exhausted!
"); exit(1); }
    else {
        np->key = key;
        np->left = left;
        np->right = right;
        return np;
    }
}

void insert(int key, Node **tree) {
    if ( (*tree) == NULL) {
        (*tree) = create_node(key, NULL, NULL); return; }
    if (key <= (*tree)->key) {
        insert(key, &(*tree)->left);
    } else {
        insert(key, &(*tree)->right);}
}

insert(10, &root);
insert(16, &root);
insert(5, &root);
insert(11, &root);
Observations

• Code, Static storage are easy: they never grow or shrink
• Stack space is relatively easy: stack frames are created and destroyed in last-in, first-out (LIFO) order
• *Managing the heap is tricky*: memory can be allocated / deallocated at any time
  – If you forget to deallocate memory: “Memory Leak”
    • Your program will eventually run out of memory
  – If you call free twice on the same memory: “Double Free”
    • Possible crash or exploitable vulnerability
  – If you use data after calling free: “Use after free”
    • Possible crash or exploitable vulnerability
int x = 2;
int result;

int foo(int n)
{
    int y;
    if (n <= 0) {
        printf("End case!\n"); return 0; }
    else
    {
        y = n + foo(n-x);
        return y;
    }
}
result = foo(10);

Right after the `printf` executes but before the `return 0`, how many copies of `x` and `y` are there allocated in memory?

A: #x = 1, #y = 1
B: #x = 1, #y = 5
C: #x = 5, #y = 1
D: #x = 1, #y = 6
E: #x = 6, #y = 6
int x = 2;
int result;

int foo(int n)
{
    int y;
    if (n <= 0) { printf("End case!\n"); return 0; }
    else
    {
        y = n + foo(n-x);
        return y;
    }
}
result = foo(10);

Right after the printf executes but before the return 0, how many copies of x and y are there allocated in memory?

A: #x = 1, #y = 1
B: #x = 1, #y = 5
C: #x = 5, #y = 1
D: #x = 1, #y = 6
E: #x = 6, #y = 6

Stack:
foo(10)
foo(8)
foo(6)
foo(4)
foo(2)
foo(0)
And In Conclusion, ...

• C has three main memory segments in which to allocate data:
  – Static Data: Variables outside functions
  – Stack: Variables local to function
  – Heap: Objects explicitly malloc-ed/free-d.

• Heap data is biggest source of bugs in C code