Memory and Arrays
CSE 333

Instructor: Alex Sanchez-Stern

Teaching Assistants:
Justin Tysdal
Sayuj Shahi
Nicholas Batchelder
Leanna Mi Nguyen
Administrivia (1)

- Exercise 0 was due yesterday
  - Any significant problems getting it done?
    - If unusual situation, please contact the staff with an email message so we can help
  - Sample solution was posted yesterday and linked to calendar
    - Requires CSE login; please do not distribute
      - Non-CSE students should have received guest accounts for the quarter. Let us know (email to cse333-staff) if you’re not set up, but we’ll probably need for you to contact support[at]cs to get it resolved

- Exercise 1 out today, due Monday morning @ 10 am
Administrivia (2)

- Reference system for grading is *current* CSE lab/attu/VM
  - For both exercises and homework (project) code
  - It’s your job to be sure your solution(s) work there
    - Just because it works on ReallyCoolLinuxDistribution® doesn’t mean it necessarily works on other Linux systems, including ours – there are lots of subtle differences between Linux systems that can cause problems
- If you’re having any problems with attu, let us know ASAP! Support has been having some issues with some attu accounts.
Administrivia (3)

❖ Homework 0 out now
  ▪ Due Monday @ 11 pm
  ▪ Logistics and infrastructure for projects – should be quick

❖ Homework 1 will be posted and pushed to repos on wednesday – read and get started as soon as it’s out
  ▪ Linked list and hash table implementations in C
  ▪ Please read the spec and start looking at the code next week
    • For large projects, you must pace yourself so if something baffling happens, you can let it go for the day and come back to it tomorrow
Administrivia (4)

- Class policy changes!
  - Exercises: dropping the lowest score from grade
  - Participation: dropping the lowest three days from grade
Lecture Outline

❖ OS Processes (refresher)
❖ C’s Memory Model (refresher)
❖ Pointers (refresher)
❖ Arrays
OS and Processes

- The OS lets you run multiple applications at once
  - An application runs within an OS “process”
  - The OS timeslices each CPU between runnable processes
    - This happens very quickly: ~100 times per second
Processes and Virtual Memory

- The OS gives each process the illusion of its own private memory
  - Called the process’ address space
  - Contains the process’ virtual memory, visible only to it (via translation)
  - $2^{64}$ bytes on a 64-bit machine
  - Completely unstructured from the OS’s perspective

Virtual Memory

Contains code, data, libraries, stack, etc.
Loading

- When the OS loads a program it:
  1) Creates an address space
  2) Inspects the executable file to see what’s in it
  3) (As needed) copies regions of the file into the right place in the address space
  4) Does any final linking, relocation, or other needed preparation
Lecture Outline

❖ OS Processes (refresher)
❖ C’s Memory Model (refresher)
❖ Pointers (refresher)
❖ Arrays
Memory Management

- **Local** variables on the **Stack**
  - Automatically allocated and freed via calling conventions (push, pop, mov)

- **Global** and **static** variables in **Data**
  - Statically allocated/freed when the process starts/exports

- **Dynamically-allocated** data on the **Heap**
  - `malloc()` to request; must call `free()` to release, otherwise memory leak

![Memory Management Diagram](image)
Review: The Stack

- Used to store data associated with function calls
  - Compiler-inserted code manages stack frames for you

- Stack frame (x86-64) includes:
  - Address to return to
  - Saved registers
    - Based on calling conventions
  - Local variables
  - Argument values
    - Only if > 6 used

Technically, just a convention; the architecture and OS don’t enforce this
# Stack in Action

```c
#include <stdint.h>

int f(int, int);
int g(int);

int main(int argc, char** argv) {
    int n1 = f(3, -5);
    n1 = g(n1);
}

int f(int p1, int p2) {
    int x;
    int a[3];
    ...
    x = g(a[2]);
    return x;
}

int g(int param) {
    return param * 2;
}
```

Note: arrow points to next instruction to be executed (like in `gdb`).
Stack in Action

#include <stdint.h>

int f(int, int);
int g(int);

int main(int argc, char** argv) {
    int n1 = f(3, -5);
    n1 = g(n1);
}

int f(int p1, int p2) {
    int x;
    int a[3];
    ...
    x = g(a[2]);
    return x;
}

int g(int param) {
    return param * 2;
}

Note: arrow points to next instruction to be executed (like in gdb).
#include <stdint.h>

int f(int, int);
int g(int);

int main(int argc, char** argv) {
    int n1 = f(3, -5);
    n1 = g(n1);
}

int f(int p1, int p2) {
    int x;
    int a[3];
    ...
    x = g(a[2]);
    return x;
}

int g(int param) {
    return param * 2;
}

Note: arrow points to next instruction to be executed (like in gdb).
Stack in Action

#include <stdint.h>

int f(int, int);
int g(int);

int main(int argc, char** argv) {
    int n1 = f(3, -5);
    n1 = g(n1);
}

int f(int p1, int p2) {
    int x;
    int a[3];
    ...
    x = g(a[2]);
    return x;
}

int g(int param) {
    return param * 2;
}

Note: arrow points to next instruction to be executed (like in gdb).
Lecture Outline

❖ OS Processes (refresher)
❖ C’s Memory Model (refresher)
❖ **Pointers** (refresher)
❖ Arrays
Pointers

❖ Variables that store addresses
  ▪ It points to somewhere in the process’ virtual address space

❖ Generic definition: type* name; or type *name;
  ▪ Recommended: do not define multiple pointers on same line:
    int *p1, p2;
  ▪ Instead, use:
    int *p1;
    int *p2;

❖ Create a pointer using the unary & operator
  ▪ Example: int* foo_ptr = &foo;

❖ Follow (“dereference”) a pointer using the unary * operator
  ▪ Access the memory referred to by a pointer
  ▪ Example: int bar = 5 + *foo_ptr;
#include <stdio.h>
#include <stdint.h>

int main(int argc, char** argv) {
    int x = 351;
    int* p;    // p is a pointer to a int

    p = &x;    // p now contains the addr of x
    printf("&x is %p\n", &x);
    printf(" p is %p\n",  p);
    printf(" x is %d\n",  x);

    *p = 333;    // change value of x
    printf(" x is %d\n",  x);
    // prints the same value
    printf(" *p is %d\n", *p);

    return 0;
}
Address Space Layout Randomization

- Linux uses *address space layout randomization* (ASLR) for added security
  - Randomizes:
    - Base of stack
    - Shared library *(mmap)* location
  - Makes Stack-based buffer overflow attacks tougher
  - Makes debugging tougher
  - Can be disabled *(gdb* does this by default); Google if curious
Lecture Outline

❖ OS Processes (refresher)
❖ C’s Memory Model (refresher)
❖ Pointers (refresher)
❖ Arrays

But first, let’s take another five minute break to get up and stretch (and drink water)
(Local) Arrays

Definition: \texttt{type name[num elems]}

- Allocates $\text{num elems} \times \text{sizeof(type)}$ bytes of contiguous memory on the stack
- Initially, array values are “garbage” (i.e., uninitialized, unknown)

Size of an array

- Not stored anywhere at runtime – array does not know its own size!
  - $\text{sizeof(array)}$ only works in variable scope of array definition
- Normal usage is a compile-time constant for \textit{size} (\textit{e.g.} \texttt{int scores[175];})
- Recent versions of C (but not C++) allowed variable-length arrays
  - $\texttt{int n = 175;}$
  - $\texttt{int scores[n];}$ // OK in C99
Using Arrays

- **Initialization:**
  
  ```
  type name[size] = {val0,...,valN};
  ```
  
  - `{ }` initialization can only be used at time of definition
  - If no `size` supplied, infers from length of array initializer

  ```
  int fibs[] = {1, 1, 2, 3, 5, 8}; // this is okay
  ```

- Array name used as identifier for “collection of data”
  
  - `name[index]` specifies an element of the array and can be used as an assignment target or as a value in an expression

  ```
  int primes[6] = {2, 3, 5, 6, 11, 13};
  primes[3] = 7;
  primes[100] = 0; // memory smash!
  ```

- Cannot be reassigned or changed
Multi-dimensional Arrays

- **Generic 2D format:**

  ```c
  type name[rows][cols] = {{values},...,{values}};
  ```

  - Still allocates a single, contiguous chunk of memory
  - C stores arrays in *row-major* order

  ```c
  // a 2-row, 3-column array of doubles
  double grid[2][3];

  // a 3-row, 5-column array of ints
  int matrix[3][5] = {
    {0, 1, 2, 3, 4},
    {0, 2, 4, 6, 8},
    {1, 3, 5, 7, 9}
  };
  ```

- 2-D arrays normally only useful if size known statically in advance. Otherwise use dynamically-allocated data and pointers (later)
Arrays as Parameters

- It’s tricky to use arrays as parameters
  - What happens when you use an array name as an argument?
  - Arrays do not know their own size

```c
int sumAll(int a[]); // prototype

int main(int argc, char** argv) {
    int numbers[] = {9, 8, 1, 9, 5};
    int sum = sumAll(numbers);
    return 0;
}

int sumAll(int a[]) {
    int i, sum = 0;
    for (i = 0; i < ...???
}
Solution 1: Declare Array Size

```c
int sumAll(int a[5]); // prototype

int main(int argc, char** argv) {
    int numbers[] = {9, 8, 1, 9, 5};
    int sum = sumAll(numbers);
    printf("sum is: %d\n", sum);
    return 0;
}

int sumAll(int a[5]) {
    int i, sum = 0;
    for (i = 0; i < 5; i++) {
        sum += a[i];
    }
    return sum;
}
```

- Problem: loss of generality/flexibility
Solution 2: Pass Size as Parameter

```c
int sumAll(int a[], int size); // prototype

int main(int argc, char** argv) {
    int numbers[] = {9, 8, 1, 9, 5};
    int sum = sumAll(numbers, 5);
    printf("sum is: %d\n", sum);
    return 0;
}

int sumAll(int a[], int size) {
    int i, sum = 0;
    for (i = 0; i < size; i++) {
        sum += a[i];
    }
    return sum;
}
```

arraysum.c

- Standard idiom in C programs
Returning an Array

- Local variables, including arrays, are allocated on the stack
  - They “disappear” when a function returns!
  - Can’t safely return local arrays from functions
    - Can’t return an array as a return value – why not?

```c
int* copyArray(int src[], int size) {
    int i, dst[size];    // allowed in C99
    for (i = 0; i < size; i++) {
        dst[i] = src[i];
    }
    return dst;
}
```

No compiler error, but wrong! Returns a pointer to abandoned memory
Solution: Output Parameter

- Create the “returned” array in the caller
  - Pass it as an output parameter to `copyarray()`
  - “output parameter”: A pointer parameter that allows the called function to store values that the caller can use
  - Works because arrays are “passed” as pointers

```c
void copyArray(int src[], int dst[], int size) {
    int i;

    for (i = 0; i < size; i++) {
        dst[i] = src[i];
    }
}
```

copyarray.c
Box-and-Arrow Diagrams

```c
int main(int argc, char** argv) {
    int x = 1;
    int arr[3] = {2, 3, 4};
    int* p = &arr[1];

    printf("&x: %p;  x: %d\n", &x, x);
    printf("&arr[0]: %p;  arr[0]: %d\n", &arr[0], arr[0]);
    printf("&arr[2]: %p;  arr[2]: %d\n", &arr[2], arr[2]);
    printf("&p: %p; p: %p; *p: %d\n", &p, p, *p);

    return EXIT_SUCCESS;
}
```

<table>
<thead>
<tr>
<th>address</th>
<th>name</th>
<th>value</th>
</tr>
</thead>
</table>

boxarrow.c
Box-and-Arrow Diagrams

```c
int main(int argc, char** argv) {
    int x = 1;
    int arr[3] = {2, 3, 4};
    int* p = &arr[1];

    printf("&x: %p;  x: %d\n", &x, x);
    printf("&arr[0]: %p;  arr[0]: %d\n", &arr[0], arr[0]);
    printf("&arr[1]: %p;  arr[1]: %d\n", &arr[1], arr[1]);
    printf("&arr[2]: %p;  arr[2]: %d\n", &arr[2], arr[2]);
    printf("&p: %p;  p: %p;  *p: %d\n", &p, p, *p);

    return EXIT_SUCCESS;
}
```

<table>
<thead>
<tr>
<th>address</th>
<th>name</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;arr[0]</td>
<td>arr[0]</td>
<td>value</td>
</tr>
<tr>
<td>&amp;x</td>
<td>x</td>
<td>value</td>
</tr>
<tr>
<td>&amp;p</td>
<td>p</td>
<td>value</td>
</tr>
</tbody>
</table>
Box-and-Arrow Diagrams

```c
int main(int argc, char** argv) {
    int x = 1;
    int arr[3] = {2, 3, 4};
    int* p = &arr[1];

    printf("&x: %p;  x: %d\n", &x, x);
    printf("&arr[0]: %p;  arr[0]: %d\n", &arr[0], arr[0]);
    printf("&arr[1]: %p;  arr[1]: %d\n", &arr[1], arr[1]);
    printf("&arr[2]: %p;  arr[2]: %d\n", &arr[2], arr[2]);
    printf("&p: %p;  p: %p;  *p: %d\n", &p, p, *p);

    return EXIT_SUCCESS;
}
```

<table>
<thead>
<tr>
<th>address</th>
<th>name</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;arr[0]</td>
<td>arr[0]</td>
<td>2</td>
</tr>
<tr>
<td>&amp;p</td>
<td>p</td>
<td>&amp;arr[1]</td>
</tr>
<tr>
<td>&amp;x</td>
<td>x</td>
<td>1</td>
</tr>
</tbody>
</table>
Box-and-Arrow Diagrams

```
int main(int argc, char** argv) {
    int x = 1;
    int arr[3] = {2, 3, 4};
    int* p = &arr[1];

    printf("&x: %p;  x: %d\n", &x, x);
    printf("&arr[0]: %p; arr[0]: %d\n", &arr[0], arr[0]);
    printf("&arr[2]: %p; arr[2]: %d\n", &arr[2], arr[2]);
    printf("&p: %p; p: %p; *p: %d\n", &p, p, *p);

    return EXIT_SUCCESS;
}
```

<table>
<thead>
<tr>
<th>address</th>
<th>name</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x7fff...78</td>
<td>arr[2]</td>
<td>4</td>
</tr>
<tr>
<td>0x7fff...74</td>
<td>arr[1]</td>
<td>3</td>
</tr>
<tr>
<td>0x7fff...70</td>
<td>arr[0]</td>
<td>2</td>
</tr>
<tr>
<td>0x7fff...68</td>
<td>p</td>
<td>0x7fff...74</td>
</tr>
<tr>
<td>0x7fff...64</td>
<td>x</td>
<td>1</td>
</tr>
</tbody>
</table>
Array Memory Diagram

```
int main()
{
    int original[] = {123, 351, 333};
    int copy[3];
    copyArray(original, copy, 3);
}

void copyArray(int src[], int dst[], int size) {
    for (int i = 0; i < size; i++) {
        dst[i] = src[i];
    }
}
```
# Array Memory Diagram

```c
int main() {
    int original[] = {123, 351, 333};
    int copy[3];
    copyArray(original, copy, 3);
}

void copyArray(int src[], int dst[], int size) {
    for (int i = 0; i < size; i++) {
        dst[i] = src[i];
    }
}
```

dst[i] is really *(dst+i)*. We aren’t changing the pointer dst!
int main() {
    int original[] = {123, 351, 333};
    int copy[3];
    copyArray(original, copy, 3);
}

void copyArray(int src[], int dst[], int size) {
    for (int i = 0; i < size; i++) {
        dst[i] = src[i];
    }
}

dst[i] is really *(dst+i). We aren’t changing the pointer dst!
Output Parameters

- Output parameters are common in library functions
  
  - `long int strtol(char* str, char** endptr, int base);`
  - `int sscanf(char* str, char* format, ...);`

```c
int num, i;
char* pEnd;
char* str1 = "333 rocks"; // ptr to read-only const data
char str2[10];

// converts "333 rocks" into long -- pEnd is conversion end
num = (int) strtol(str1, &pEnd, 10);

// reads string into arguments based on format string
num = sscanf("3 blind mice", "%d %s", &i, str2);
```

outparam.c
Parameters: reference vs. value

- There are two fundamental parameter-passing schemes in programming languages

  - **Call-by-value**
    - Parameter is a local variable initialized with a copy of the calling argument when the function is called; manipulating the parameter only changes the copy, *not* the calling argument
    - **C, Java, C++** (most things)

  - **Call-by-reference**
    - Parameter is an alias for the supplied argument; manipulating the parameter manipulates the calling argument
    - **C++ references** (we’ll see these later)
So what’s the story for arrays?

❖ Is it call-by-value or call-by-reference?

❖ Technical answer: a T[ ] array parameter is “promoted” to a pointer of type T*, and the pointer is passed by value
  ▪ So it acts like a call-by-reference array (if callee changes the array parameter elements it changes the caller’s array)
  ▪ But it’s really a call-by-value pointer (the callee can change the pointer parameter to point to something else(!))

```c
void copyArray(int src[], int dst[], int size) {
    int i;
    dst = src;  // evil! dst now points to same array as src
    for (i = 0; i < size; i++) {
        dst[i] = src[i];  // copies source array to itself!
    }
}
```
It’s unchanged!

```c
void copyArray(int src[], int dst[], int size) {
    int i;
    dst = src;  // evil! dst now points to same array as src
    for (i = 0; i < size; i++) {
        dst[i] = src[i];  // copies source array to itself!
    }
}
```
Array Parameters – [ ] or * ?

❖ Array parameters are *actually* pointers to the beginning of the array

▪ The [ ] syntax for parameter types is just for convenience
  • Use whichever best helps the reader

This code:

```c
void f(int a[]);

int main( ... ) {
    int a[5];
    ...
    f(a);
    return EXIT_SUCCESS;
}
```

Equivalent to:

```c
void f(int* a);

int main( ... ) {
    int a[5];
    ...
    f(&a[0]);
    return EXIT_SUCCESS;
}
```
Extra Exercises

❖ Some lectures contain “Extra Exercise” slides
  ▪ Extra practice for you to do on your own without the pressure of being graded
  ▪ You may use libraries and helper functions as needed
    • Early ones may require reviewing 351 material or looking at documentation for things we haven’t discussed in 333 yet
  ▪ Always good to provide test cases in main()

❖ Solutions for these exercises will be posted on the course website
  ▪ You will get the most benefit from implementing your own solution before looking at the provided one
Extra Exercise #1

- Write a function that:
  - Accepts an array of 32-bit unsigned integers and a length
  - Reverses the elements of the array in place
  - Returns nothing (void)
Extra Exercise #2

- Write a function that:
  - Accepts a string as a parameter
  - Returns:
    - The first white-space separated word in the string as a newly-allocated string
    - AND the size of that word
  - (probably need to wait until we look at malloc/free later)