CIS 330:

Lecture 5:
Memory, Memory Errors, and Debuggers
Announcements

• Matt’s OH: Mon 12-1, Tues 12-2
• Hank’s OH: Weds 3:45-4:45, Fri 12:30-1:30

• Lecture cancelled on April 29
  – YouTube lecture will be posted in its place

• Also: will leave for UK before the end of the quarter ... projects will need to be turned in.
  – (date coming)
Let’s grade

• Grade 2A/2D
Outline

• Review
• More on Memory
• Memory Errors
• Debuggers
Outline

• Review
• More on Memory
• Memory Errors
• Debuggers
Memory Segments

• Von Neumann architecture: one memory space, for both instructions and data
• → so break memory into “segments”
  – ... creates boundaries to prevent confusion
• 4 segments:
  – Code segment
  – Data segment
  – Stack segment
  – Heap segment
C: must manage your own memory

• This is a big change from other programs
• You keep track of memory
  – Allocation
  – How much there is / indexing memory
  – Deallocation
malloc

- malloc: command for allocating memory

```
#include <stdlib.h>

void *
calloc(size_t count, size_t size);

void
free(void *ptr);

void *
malloc(size_t size);

void *
realloc(void *ptr, size_t size);

void *
reallocf(void *ptr, size_t size);

void *
valloc(size_t size);
```

**DESCRIPTION**

The `malloc()`, `calloc()`, `valloc()`, `realloc()`, and `reallocf()` functions allocate memory. The allocated memory is aligned such that it can be used for any data type, including AltiVec- and SSE-related types. The `free()` function frees allocations that were created via the preceding allocation functions.

The `malloc()` function allocates `size` bytes of memory and returns a pointer to the allocated memory.
# Allocation / Deallocation Example

```c
#include <stdlib.h>
int main()
{
    int stack_varA;
    int stack_varB[2];
    int *heap_varA;
    int *heap_varB;
    heap_varA = malloc(sizeof(int));
    heap_varB = malloc(sizeof(int)*2);
    free(heap_varA);
    free(heap_varB);
}
```

Automatic allocation on the stack. (Deallocation occurs when out of scope.)

Explicit allocation from the heap. (Deallocation occurs with “free” call.)
sizeof

• sizeof: gets size of type

• Usually:
  – sizeof(int) == 4
  – sizeof(float) == 4
  – sizeof(double) == 8
  – sizeof(unsigned char) == 1
  – sizeof(char) == 1

• ➔ array of 10 ints ➔ malloc(10*sizeof(int))
Hexadecimal

- Binary: 2 values
- Decimal: 10 values
- Hexadecimal: 16 values
  - 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F
- 0x: prefix for hexadecimal
- 0x10 = 16
- 0x101 = 257
Memory Addresses

• Every location in memory has an address associated with it
• Locations in memory are represented in hexadecimal

Memory addresses descend in the stack, ascend in the heap.
Pointers

• Pointers store locations in memory

• “&”: unary operator that gives the address of a variable.

```c
int x;
int *yp = &x;
```
Pointers

- Pointers store locations in memory

```c
#include <stdio.h>

int main()
{
    int x, y;
    printf("The location of x is %p and the location of y is %p\n", &x, &y);
}
```

The location of x is 0x7fff56d26bcc and the location of y is 0x7fff56d26bc8
NULL pointer

- **NULL**: defined by compiler to be a location that is not valid.
  - Typically 0x00000000
- You can use NULL to initialize pointers, and also to check to see whether a pointer is set already.
‘*’ operator

- Let “ptr” be a pointer
- Then “*ptr” returns value in the address that pts points to.
- * = “dereference operator”

```c
#include <stdio.h>

int main()
{
    int x = 3;
    int *y = &x;
    int z = *y;
    printf("x = %d, z = %d\n", x, z);
}
```
Behavior of dereference

• When you dereference, you get the value at that moment.
  – Whatever happens afterwards won’t have effect.

```c
#include <stdio.h>

int main()
{
    int x = 3;
    int *y = &x;
    int z = *y;
    x = 4;
    printf("x = %d, y = %d, z = %d\n", x, *y, z);
}
```

```
C02LN00GFD58:330 hank$ cat ptr2.c
#include <stdio.h>

int main()
{
    int x = 3;
    int *y = &x;
    int z = *y;
    x = 4;
    printf("x = %d, y = %d, z = %d\n", x, *y, z);
}

C02LN00GFD58:330 hank$ gcc ptr2.c
C02LN00GFD58:330 hank$ ./a.out
x = 4, y = 4, z = 3
```
Pointer Arithmetic

- You can combine pointers and integers to get new pointer locations

```c
#include <stdio.h>

int main()
{
    int x = 3;
    int *y = &x;
    int *z = y + 1;
    char a = 'A';
    char *b = &a;
    char *c = b + 1;
    printf("x = %d, y = %p, z = %p\n", x, y, z);
    printf("a = %c, b = %p, c = %p\n", a, b, c);
}
```

```bash
hank$ cat ptr_arith.c
C02LN00GFD58:330 hank$ gcc ptr_arith.c
C02LN00GFD58:330 hank$ ./a.out
x = 3, y = 0x7fff5d397bcc, z = 0x7fff5d397bd0
a = A, b = 0x7fff5d397bb7, c = 0x7fff5d397bb8
```
Arrays

• Arrays: container that has multiple elements of identical type, all stored in contiguous memory

int A[10];

→ 10 integers, stored in 40 consecutive bytes (assuming sizeof(int) == 4)

Arrays are just pointers. You can use arrays and pointers interchangeably.
[ ] operator

• [ ] is a way of dereferencing memory
  – Recall that ‘*’ is the dereference operator

• A[0] <= => *A
• A[5] <= => *(A+5);
More array relationships

```c
int A[10];
int *B;

B=&(A[0]) → B = A
B=&(A[5]) → B = A+5
```
Pointers to pointers

• Remember: pointer points to a location in memory
  – We’ve been considering cases where locations in memory are arrays of integers
  – But locations in memory could be pointer themselves
Call by reference / call by value

• Refers to how parameters are passed to a function.
  – Call by reference: send a reference (pointer) as a function parameter
    • Side effects in that function affect the variable in the calling function
  – Call by value: send the value of the variable as a function parameter
    • Side effects in that function don’t affect the variable in the calling function
Call by Reference

```c
#include <stdio.h>

void foo(int *x)
{
    *x = *x + 1;
}

int main()
{
    int x = 2;
    foo(&x);
    printf("X is \%d\n", x);
}
```

```bash
hank$ cat cbr.c
hank$ gcc cbr.c
hank$ ./a.out
X is 3
```
Call by Value

```c
#include <stdio.h>

void foo(int x)
{
    x = x+1;
}

int main()
{
    int x = 2;
    foo(x);
    printf("X is %d\n", x);
}
```

```
C02LN00GFD58:330 hank$ gcc cbv.c
C02LN00GFD58:330 hank$ ./a.out
X is 2
```
Outline

• Review
• More on Memory
• Memory Errors
• Debuggers
## Stack vs Heap: Pros and Cons

<table>
<thead>
<tr>
<th></th>
<th>Stack</th>
<th>Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td>Allocation/Deallocation</td>
<td>Automatic</td>
<td>Explicit</td>
</tr>
</tbody>
</table>
How stack memory is allocated into Stack Memory Segment

```c
void foo()
{
    int stack_varA;
    int stack_varB;
}

int main()
{
    int stack_varC;
    int stack_varD;
    foo();
}
```
How stack memory is allocated into Stack Memory Segment

```c
void foo()
{
    int stack_varA;
    int stack_varB;
}

int main()
{
    int stack_varC;
    int stack_varD;
    foo();
}
```
How stack memory is allocated into Stack Memory Segment

```c
void foo()
{
    int stack_varA;
    int stack_varB;
}

int main()
{
    int stack_varC;
    int stack_varD;
    foo();
}
```
How stack memory is allocated into Stack Memory Segment

```c
void foo()
{
    int stack_varA;
    int stack_varB;
}

int main()
{
    int stack_varC;
    int stack_varD;
    foo();
}
```
How stack memory is allocated into Stack Memory Segment

int doubler(int A) {
    int stack_varA;
    stack_varA = 2*A;
    return stack_varA;
}

int main() {
    int stack_varC;
    int stack_varD = 3;
    stack_varC = doubler(stack_varD);
}
How stack memory is allocated into Stack Memory Segment

```c
int doubler(int A)
{
    int stack_varA;
    stack_varA = 2*A;
    return stack_varA;
}

int main()
{
    int stack_varC;
    int stack_varD = 3;
    stack_varC = doubler(stack_varD);
}
```
How stack memory is allocated into Stack Memory Segment

```c
int doubler(int A) {
    int stack_varA;
    stack_varA = 2*A;
    return stack_varA;
}

int main() {
    int stack_varC;
    int stack_varD = 3;
    stack_varC = doubler(stack_varD);
}
```
How stack memory is allocated into Stack Memory Segment

```c
int doubler(int A) {
    int stack_varA;
    stack_varA = 2*A;
    return stack_varA;
}

int main() {
    int stack_varC;
    int stack_varD = 3;
    stack_varC = doubler(stack_varD);
}
```

Return copies into location specified by calling function

Code
Data
Stack
- stack_varC
- stack_varD
- <info for how to get back to main>
  - A (= 3)
  - <Location for RV>
    - stack_varA
Free
Heap
How stack memory is allocated into Stack Memory Segment

```c
int doubler(int A) {
    int stack_varA;
    stack_varA = 2*A;
    return stack_varA;
}

int main() {
    int stack_varC;
    int stack_varD = 3;
    stack_varC = doubler(stack_varD);
}
```
This code is very problematic ... why?

```c
int *foo()
{
    int stack_varC[2] = { 0, 1 };
    return stack_varC;
}

int *bar()
{
    int stack_varD[2] = { 2, 3 };   
    return stack_varD;
}

int main()
{
    int *stack_varA, *stack_varB;
    stack_varA = foo();
    stack_varB = bar();
    stack_varA[0] *= stack_varB[0];
}
```

foo and bar are returning addresses that are on the stack ... they could easily be overwritten (and bar’s `stack_varD` overwrites foo’s `stack_varC` in this program)
int main()
{
    int stack_varA;
    {
        int stack_varB = 3;
    
    }
}

int main()
{
    int stack_varA;
    {
        int stack_varB = 3;
    }
}

Diagram: Nested Scope

- Code
- Data
- Stack
  - stack_varA
  - stack_varB
- Free
- Heap
You can create new scope within a function by adding `{` and `}`.

```c
int main()
{
    int stack_varA;
    {
        int stack_varB = 3;
    }
}
```
# Stack vs Heap: Pros and Cons

<table>
<thead>
<tr>
<th></th>
<th>Stack</th>
<th>Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td>Allocation/Deallocation</td>
<td>Automatic</td>
<td>Explicit</td>
</tr>
<tr>
<td>Access</td>
<td>Fast</td>
<td>Slower</td>
</tr>
</tbody>
</table>

Memory pages associated with stack are almost always immediately available.

Memory pages associated with heap may be located anywhere ... may be caching effects.
Stack vs Heap: Pros and Cons

<table>
<thead>
<tr>
<th></th>
<th>Stack</th>
<th>Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td>Allocation/Deallocation</td>
<td>Automatic</td>
<td>Explicit</td>
</tr>
<tr>
<td>Access</td>
<td>Fast</td>
<td>Slower</td>
</tr>
<tr>
<td>Variable scope</td>
<td>Limited</td>
<td>Unlimited</td>
</tr>
</tbody>
</table>
Variable scope: stack and heap

```c
int *foo()
{
    int stack_varA[2] = { 0, 1 };
    return stack_varA;
}

int *bar()
{
    int *heap_varB;
    heap_varB = malloc(sizeof(int)*2);
    heap_varB[0] = 2;
    heap_varB[1] = 2;
    return heap_varB;
}

int main()
{
    int *stack_varA;
    int *stack_varB;
    stack_varA = foo(); /* problem */
    stack_varB = bar(); /* still good */
}
```

bar returned memory from heap

The calling function – i.e., the function that calls bar – must understand this and take responsibility for calling free.

If it doesn’t, then this is a “memory leak”.

Memory leaks

It is OK that we are using the heap ... that’s what it is there for

The problem is that we lost the references to the 49 allocations on heap

The heap’s memory manager will not be able to re-claim them ... we have effectively limited the memory available to the program.

```c
int i;
int stack_varA;
for (i = 0; i < 50; i++)
    stack_varA = bar();
```
Running out of memory (stack)

```c
int endless_fun()
{
    endless_fun();
}

int main()
{
    endless_fun();
}
```

stack overflow: when the stack runs into the heap.

There is no protection for stack overflows.

(Checking for it would require coordination with the heap’s memory manager on every function calls.)
Running out of memory (heap)

```c
int *heaps_o_fun()
{
    int *heap_A = malloc(sizeof(int)*1000000000); // Allocation too big ...
    return heap_A;
}

int main()
{
    int *stack_A;
    stack_A = heaps_o_fun();
}
```

If the heap memory manager doesn’t have room to make an allocation, then malloc returns NULL .... a more graceful error scenario.
## Stack vs Heap: Pros and Cons

<table>
<thead>
<tr>
<th></th>
<th>Stack</th>
<th>Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td>Allocation/Deallocation</td>
<td>Automatic</td>
<td>Explicit</td>
</tr>
<tr>
<td>Access</td>
<td>Fast</td>
<td>Slower</td>
</tr>
<tr>
<td>Variable scope</td>
<td>Limited</td>
<td>Unlimited</td>
</tr>
<tr>
<td>Fragmentation</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Memory Fragmentation

- Memory fragmentation: the memory allocated on the heap is spread out of the memory space, rather than being concentrated in a certain address space.
Memory Fragmentation

Negative aspects of fragmentation?
(1) can’t make big allocations
(2) losing cache coherency

```c
int *bar()
{
    int *heap_varA;
    heap_varA = malloc(sizeof(int)*2);
    heap_varA[0] = 2;
    heap_varA[1] = 2;
    return heap_varA;
}

int main()
{
    int i;
    int stack_varA[50];
    for (i = 0 ; i < 50 ; i++)
        stack_varA[i] = bar();
    for (i = 0 ; i < 25 ; i++)
        free(stack_varA[i*2]);
}
Fragmentation and Big Allocations

Even if there is lots of memory available, the memory manager can only accept your request if there is a big enough contiguous chunk.
# Stack vs Heap: Pros and Cons

<table>
<thead>
<tr>
<th></th>
<th>Stack</th>
<th>Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Allocation/Deallocation</strong></td>
<td>Automatic</td>
<td>Explicit</td>
</tr>
<tr>
<td><strong>Access</strong></td>
<td>Fast</td>
<td>Slower</td>
</tr>
<tr>
<td><strong>Variable scope</strong></td>
<td>Limited</td>
<td>Unlimited</td>
</tr>
<tr>
<td><strong>Fragmentation</strong></td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Outline

• Review
• More on Memory
• Memory Errors
• Debuggers
Memory Errors

• Array bounds read

```c
int main()
{
    int var;
    int arr[3] = { 0, 1, 2 };
    var=arr[3];
}
```

• Array bounds write

```c
int main()
{
    int var = 2;
    int arr[3];
    arr[3]=var;
}
```
Memory Errors

Vocabulary: “dangling pointer”: pointer that points to memory that has already been freed.

```c
int main()
{
    int *var = malloc(sizeof(int)*2);
    var[0] = 0;
    var[1] = 2;
    free(var);
    var[0] = var[1];
}
```

When does this happen in real-world scenarios?
Memory Errors

• Freeing unallocated memory

```c
int main()
{
    int *var = malloc(sizeof(int)*2);
    var[0] = 0;
    var[1] = 2;
    free(var);
    free(var);
}
```

When does this happen in real-world scenarios?
Memory Errors

• Freeing non-heap memory

```c
int main()
{
    int var[2]
    var[0] = 0;
    var[1] = 2;
    free(var);
}
```

When does this happen in real-world scenarios?
Memory Errors

• NULL pointer read / write
  
  ```c
  int main()
  {
    char *str = NULL;
    printf(str);
    str[0] = 'H';
  }
  ```

• NULL is never a valid location to read from or write to, and accessing them results in a “segmentation fault”
  – .... remember those memory segments?

When does this happen in real-world scenarios?
Memory Errors

• Uninitialized memory read

```c
int main()
{
    int *arr = malloc(sizeof(int)*10);
    int V2=arr[3];
}
```

When does this happen in real-world scenarios?
Memory error in action

```c
#include <stdio.h>

int main()
{
    int *X = NULL;
    printf("X is %d\n", *X);
}
```

```
fawcett:~ error childs$ cat t.c
fawcett:~ error childs$ gcc t.c
fawcett:~ error childs$ ./a.out
Segmentation fault
```

Outline

• Review
• More on Memory
• Memory Errors
• Debuggers
What is a bug?
What is a debugger

• A debugger is a program
• A debugger allows you to examine another program
• Debuggers are paired with compilers
  – gcc compiler → gdb debugger
• Two modes:
  – Debug live program
  – Debug crashed program

Note: printf is my go-to debugger. I usually only use gdb when I am tracking a memory error.
Debug Symbols

• The compiler will not put in “debug symbols” unless you instruct it to.
  – Debug symbols: hints/bread crumbs that allow the debugger to reconstruct your program
    • Includes line numbers, variable names, etc.

• gcc: -g is flag to add debug symbols
  – gcc –g t.c → you can debug a.out
  – gcc t.c → you cannot debug a.out
Debugging live program

• Invoke debugger as: `gdb <executable name>`
• This will bring up an interpreter
• Gameplan:
  – You set break points
  – You tell the program to run
  – You inspect memory when a break point hits
Debugging crashed program: core files

• Core file: file containing state of program
  – Can be very large
  – Name: “core” or “core.#####”

• Unix will generate these when your program crashes
  – Except when “ulimit –c” is set to prevent its creation
    • ... and this is often the case
  – (Unix command “ulimit” limit what resources a program can use)
Debugging crashed program

• Invoke debugger as:
  
gdb <executable name> corefile

• This will bring up an interpreter

• Gameplan:
  
  – You inspect memory from when program crashed and try to figure out what happened
GDB commands

• run: tell debugger to run
  – run argA argB argC
    • (invoke with argA,B,C passed to program on command line)

• break: set a break point

• where: where in the program

• up/down: go up and down the call stack

• <varname>: prints contents of a variable name
GDB

• GDB is always available on Unix
• GDB used to be installed on Mac by default
  – You now have to install it yourself
  – Mac’s actually use CLANG
  – CLANG’s debugger is LLDB
gdb in action

Invoke gdb with our program

(gdb) run
Starting program: /Users/childs/330/error/a.out
Reading symbols for shared libraries ... done

Program received signal EXC_BAD_ACCESS, Could not access memory.
Reason: KERN_INVALID_ADDRESS at address: 0x0000000000000000
0x000000010000f10 in main ()
(gdb) where
#0 0x000000010000f10 in main ()
(gdb) The program is running. Exit anyway? (y or n)
gdb in action

```bash
fawcett:~ error childs$ gcc -g t.c
fawcett:~ error childs$ gdb a.out
```

GNU gdb 6.3.50-20050815 (Apple version gdb-1510) (Wed Sep 22 02:45:02 UTC 2010)
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "x86_64-apple-darwin"...Reading symbols for shared libraries .. done

```
(gdb) run
Starting program: /Users/childs/330/error/a.out
Reading symbols for shared libraries .. done

Program received signal EXC_BAD_ACCESS, Could not access memory.
Reason: KERN_INVALID_ADDRESS at address: 0x0000000000000000
0x000000001000000f10 in main () at t.c:6
  6 printf("X is %d\n", *X);
(gdb) where
#0 0x000000001000000f10 in main () at t.c:6
(gdb) print X
$1 = (int *) 0x0
```

- Tells us line of crash
- We can inspect variables
Project 4A

• Will be posted tonight or tomorrow morning
• You will practice debugging & using a debugger
  – There will be 6-8 programs that you will need to debug
    • Can use gdb or lldb
    • May want to run on ix
• Will be due next week