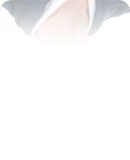


**Great Ideas**

**in**

UC Berkeley 

Teaching Professor Dan Garcia

**Computer Architecture** (a.k.a. Machine Structures)



Garcia, Kao

**cs61c.org **

**From ENIAC (1946) to EDSAC (1949) **

• ENIAC: First Electronic General-Purpose Computer • Needed 2-3 days to setup new program • Programmed with patch cords and switches

• At that time & before, "computer" mostly referred to people who did calculations

• Mostly women! (See *Hidden Figures*, 2016)

• EDSAC: First General Stored-Program Computer

• Programs held as **numbers in memory** • Revolution! Program is also data!

• 35-bit binary **two’s complement** wordsGarcia, Kao

**05 Memory (Mis) Management** (2) 

**What gets printed?**

[Concept Check]

sizeof()**: compile-time operator; gives size in bytes (of type or variable).**

// for this exercise, assume

// shorts are 16b on a 64-bit architecture void mystery(short arr[], int len) { printf("%d ", len);

printf("%d\n", sizeof(arr));

}

int main() {

short nums[] = {1, 2, 3, 99, 100}; printf("%d ", sizeof(nums));

mystery(nums, sizeof(nums)/sizeof(short)); return 0;

}

A. 10 5 10

B. 10 5 8

C. 80 5 80

D. 80 5 40

E. Other

Garcia, Kao

**05 Memory (Mis) Management** (3) 

**What gets printed?**

[Concept Check]

sizeof()**: compile-time operator; gives size in bytes (of type or variable).**

// for this exercise, assume

// shorts are 16b on a 64-bit architecture void mystery(short arr[], int len) { printf("%d ", len);

A. 10 5 10 B. 10 5 8 C. 80 5 80 D. 80 5 40

printf("%d\n", sizeof(arr)); }

Array has decayed to a pointer

E. Other

int main() {

short nums[] = {1, 2, 3, 99, 100}; printf("%d ", sizeof(nums));

mystery(nums, sizeof(nums)/sizeof(short)); return 0;

}

In array’s declared scope, total array size.

In array’s declared scope, # elements in array.

Garcia, Kao

**05 Memory (Mis) Management** (4) 

**Agenda** 

**Memory Locations**

• Memory Locations

• The Stack

• The Heap

• Linked List Example

• When Memory Goes Bad • Implementing Memory

Management

Garcia, Kao

**05 Memory (Mis) Management** (5) 

**C Program Address Space** • A program’s address space contains 4 regions:

0xFFFF FFFF

**stack**

• **Stack**: local variables inside functions, grows downward • **Heap**: space requested via malloc();

resizes dynamically, grows upward

• **Data (Static Data)**: variables declared outside main, does not grow or shrink

• **Text (aka code)**: program executable loaded when program starts, does not change

• 0x00000000 chunk is unwriteable/unreadable

so you that crash on NULL pointer access

0x0000 0000

• Programming in C requires knowing where objects are in memory, otherwise things don’t work as expected. • By contrast, Java hides location of objects.

heap

data

text

For now, OS somehow prevents accesses between stack and heap.

(more later w/virtual memory) Garcia, Kao

**05 Memory (Mis) Management** (6) 

**Where are variables allocated?**

• **Global**: If declared **outside** a function, allocated in **data (static)** storage.

**stack** 0xFFFF FFFF

• **Local**: If declared **inside** function, allocated on the **stack** and

freed when function returns. • NB: main() is also a function.

• For both these memory types, the management is **automatic**. • You don’t need to worry about deallocating when you are no longer using them.

int myGlobal; … main() {

int myTemp; …

}

0x0000 0000

heap

datatext

• But a variable **does not exist anymore** once a function ends!

Garcia, Kao

**05 Memory (Mis) Management** (7) 

**Agenda The Stack**

• Memory Locations

• The Stack

• The Heap

• Linked List Example

• When Memory Goes Bad • Implementing Memory

Management

Garcia, Kao

**05 Memory (Mis) Management** (8) 

**The Stack**

• Every time a function is called,

a new **stack frame** is allocated on the stack.

• A stack frame includes:

• Return “instruction” address (who called me?)

• Arguments\*

• Space for other local variables

• Stack frames contiguous blocks of memory; the **stack pointer** indicates the start of stack frames. **SP**

• When function ends, stack frame is tossed off the stack; frees memory for future stack frames.

• (more later when we cover details for a RISC-V

fooA() { fooB(); } fooB() { fooC(); } fooC() { … }

fooA frame

fooB frame

fooC frame

processor architecture) 

**05 Memory (Mis) Management** (9)

\*see slide notes in pptx for technical elaboration

Garcia, Kao 

**The Stack is Last In, First Out (LIFO)**

int 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 Pointer Stack Pointer Stack Pointer Stack Pointer Stack Pointer Stack Pointer

stack

Stack

grows

down

The stack grows

downward; a()’s

local variables have

lower byte addresses

than main()’s, and so

on.

Garcia, Kao

**05 Memory (Mis) Management** (10) 

**Recall: Array Are Very Primitive** 1. Array bounds are not checked during element access.

[From last time]

• Consequence: We can accidentally access off the end of an array!

2. An array is passed to a function as a pointer.

• Consequence: The array size is lost! Be careful with sizeof()!

3. Declared arrays are only allocated while the scope is valid.

char \*foo() {

char string[32]; ...;

return string;

} **is incorrect** Solution:

Dynamic memory allocation!

(more ~~late~~ now)

Garcia, Kao

**05 Memory (Mis) Management** (11) 

**Passing Pointers into the Stack **

It is fine to pass a pointer to stack space further down.

I.e., pointers to addresses higher in the stack point to data in currently allocated stack frames.

**However, However, it is catastrophically bad to return a pointer to something in the stack!**

Memory will be overwritten when other functions are called!

So your data would no longer exist…and writes can overwrite key pointers, causing crashes!

void load\_buf() { … }; int main() {

… char buf[...]; load\_buf(**buf**, BUFLEN); … }

char \*make\_buf() {

char buf[50]; return buf;

}void foo() {…}

int main(){

char \*stackAddr = \ make\_buf(); foo(stackAddr); … }

stack

buf char array

persistent through load\_buf’s

execution

stack

stackAddr stackAddr points to overwritten

memory

~~buf???~~

Garcia, Kao

**05 Memory (Mis) Management** (12) 

**Agenda The Heap**

• Memory Locations

• The Stack

• The Heap

• Linked List Example

• When Memory Goes Bad • Implementing Memory

Management

Garcia, Kao

**05 Memory (Mis) Management** (13) 

**What is the Heap?**

• The heap is dynamic memory – memory that can be

allocated, resized, and freed during program runtime.

• Useful for persistent memory across function calls.

• But biggest source of pointer bugs, memory leaks, …

• Similar to Java **new** command allocates memory….but with key differences below. • **Huge** pool of mem (usually >> stack), but not allocated in contiguous order. • Back-to-back requests for heap memory *could* result in blocks very far apart • In C, specify number of bytes of memory **explicitly** to allocate/deallocate item. • malloc(): Allocates raw, uninitialized memory from heap

• free(): Frees memory on heap

• realloc(): Resizes previously allocated heap blocks to new size

• Unlike the stack, memory gets reused only when programmer **explicitly** cleans up Garcia, Kao

**05 Memory (Mis) Management** (14) 

void \*malloc(size\_t n) • Allocates a block of uninitialized memory:

• size\_t n is an unsigned integer type big enough to “count” memory bytes. • Returns void \* pointer to block of memory on heap.

• A return of NULL indicates no more memory (**always** check for it!!!) • To allocate a struct:

typedef struct { ... } TreeNode;

TreeNode \*tp = (TreeNode \*) malloc(sizeof(TreeNode));

**Typecast** casts return value

from type (void \*) to (TreeNode \*)

• To allocate an array of 20 ints:

sizeof(type) gives size in bytes.Assuming size of objects

int \*ptr = (int \*) malloc(20\*sizeof(int)); if (ptr != NULL) { … // always check NULL after • Many years ago ints used to be 16b. Now, 32b or 64b…

can lead to misleading, unportable code. Use sizeof()!

Garcia, Kao

**05 Memory (Mis) Management** (15) 

void free(void \*ptr)

• Dynamically frees heap memory

• ptr is a pointer containing an address originally returned by malloc()/realloc().

int \*ptr = (int \*) malloc (sizeof(int)\*20);

...

free(ptr); // implicit typecast to (void \*)

• When you free memory, be sure to pass the original address returned from malloc(). Otherwise, crash (or worse!)



Garcia, Kao

**05 Memory (Mis) Management** (16) 

void \*realloc(void \*ptr, size\_t size) • Resize a previously allocated block at ptr to a new size.

• Returns new address of the memory block.

• In doing so, it may need to copy all data to a new location.

• realloc(NULL, size); // behaves like malloc

• realloc(ptr, 0); // behaves like free, deallocates heap block • Remember: Always check for return NULL,

which would mean you’ve run out of memory!

**int \*ip; ip = (int \*) malloc(10\*sizeof(int));**

**… … … /\* check for NULL \*/**

**ip = (int \*) realloc(ip, 20\*sizeof(int));**

**/\* contents of first 10 elements retained \*/**

**… … … /\* check for NULL \*/**

**realloc(ip,0); /\* equivalent to free(ip); \*/**

Garcia, Kao

**05 Memory (Mis) Management** (17) 

**Agenda** 

**Linked List Example**

• Memory Locations

• The Stack

• The Heap

• Linked List Example

• When Memory Goes Bad • Implementing Memory

Management

Garcia, Kao

**05 Memory (Mis) Management** (18) 

**Linked List Example**

# include <string.h>

int main() {

struct Node \*head = NULL;

1

add\_to\_front(&head, "abc");

2

… // free nodes, strings here… }

struct Node {

char \*data;

struct Node \*next; };

void add\_to\_front(struct Node \*\*head\_ptr, char \*data) {

struct Node \*node = (struct Node \*) malloc(sizeof(struct Node));

3

node->data = (char \*) malloc(strlen(data) + 1); // extra byte!

4

strcpy(node->data, data); // strcpy also copies null terminator

5

node->next = \*head\_ptr;

6

\*head\_ptr = node;

7

}

Garcia, Kao

**05 Memory (Mis) Management** (19) 

**Linked List Example**

# include <string.h>

int main() {

struct Node \*head = NULL;

1

add\_to\_front(&head, "abc");

2

… // free nodes, strings here… }

struct Node {

char \*data;

struct Node \*next; };

void add\_to\_front(struct Node \*\*head\_ptr, char \*data) {

struct Node \*node = (struct Node \*) malloc(sizeof(struct Node));

3

node->data = (char \*) malloc(strlen(data) + 1); // extra byte!

4

strcpy(node->data, data); // strcpy also copies null terminator

5

node->next = \*head\_ptr;

6

\*head\_ptr = node;

7

}

head\_ptr head NULL

data 'a' 'b' 'c' '\0'

Garcia, Kao

**05 Memory (Mis) Management** (20) 

**Linked List Example**

# include <string.h>

int main() {

struct Node \*head = NULL;

1

add\_to\_front(&head, "abc");

2

… // free nodes, strings here… }

struct Node {

char \*data;

struct Node \*next; };

void add\_to\_front(struct Node \*\*head\_ptr, char \*data) { struct Node \*node = (struct Node \*) malloc(sizeof(struct Node));

3

node->data = (char \*) malloc(strlen(data) + 1); // extra byte!

4

strcpy(node->data, data); // strcpy also copies null terminator

5

node->next = \*head\_ptr;

6

\*head\_ptr = node;

7

} 0x300

node 0x300 ??? ???

head\_ptr NULL

pointer stored on stack

malloc’ed sizeof(struct Node) bytes starting at heap address 0x300

data 'a' 'b' 'c' '\0' Garcia, Kao

**05 Memory (Mis) Management** (21) 

**Linked List Example**

# include <string.h>

int main() {

struct Node \*head = NULL;

1

add\_to\_front(&head, "abc");

2

… // free nodes, strings here… }

struct Node {

char \*data;

struct Node \*next; };

void add\_to\_front(struct Node \*\*head\_ptr, char \*data) { struct Node \*node = (struct Node \*) malloc(sizeof(struct Node));

3

node->data = (char \*) malloc(strlen(data) + 1); // extra byte!

4

strcpy(node->data, data); // strcpy also copies null terminator

5

node->next = \*head\_ptr;

6

\*head\_ptr = node;

7

}

0x300

node 0x350 ??? 0x300

head\_ptr NULL

malloc’ed 4 bytes starting at heap address 0x350

0x350

? ? ? ?

data 'a' 'b' 'c' '\0' Garcia, Kao

**05 Memory (Mis) Management** (22) 

**Linked List Example**

# include <string.h>

int main() {

struct Node \*head = NULL;

1

add\_to\_front(&head, "abc");

2

… // free nodes, strings here… }

struct Node {

char \*data;

struct Node \*next; };

void add\_to\_front(struct Node \*\*head\_ptr, char \*data) { struct Node \*node = (struct Node \*) malloc(sizeof(struct Node));

3

node->data = (char \*) malloc(strlen(data) + 1); // extra byte!

4

strcpy(node->data, data); // strcpy also copies null terminator

5

node->next = \*head\_ptr;

6

\*head\_ptr = node;

7

}

0x300

node 0x350 ??? 0x300

0x350

head\_ptr NULL data 'a' 'b' 'c' '\0'

'a' 'b' 'c' '\0'

Garcia, Kao

**05 Memory (Mis) Management** (23) 

**Linked List Example**

# include <string.h>

int main() {

struct Node \*head = NULL;

1

add\_to\_front(&head, "abc");

2

… // free nodes, strings here… }

struct Node {

char \*data;

struct Node \*next; };

void add\_to\_front(struct Node \*\*head\_ptr, char \*data) { struct Node \*node = (struct Node \*) malloc(sizeof(struct Node));

3

node->data = (char \*) malloc(strlen(data) + 1); // extra byte!

4

strcpy(node->data, data); // strcpy also copies null terminator

5

node->next = \*head\_ptr;

6

\*head\_ptr = node;

7

}

0x300

node 0x350 NULL 0x300

0x350

head\_ptr NULL data 'a' 'b' 'c' '\0'

'a' 'b' 'c' '\0'

Garcia, Kao

**05 Memory (Mis) Management** (24) 

**Linked List Example**

# include <string.h>

int main() {

struct Node \*head = NULL;

1

add\_to\_front(&head, "abc");

2

… // free nodes, strings here… }

struct Node {

char \*data;

struct Node \*next; };

void add\_to\_front(struct Node \*\*head\_ptr, char \*data) { struct Node \*node = (struct Node \*) malloc(sizeof(struct Node));

3

node->data = (char \*) malloc(strlen(data) + 1); // extra byte!

4

strcpy(node->data, data); // strcpy also copies null terminator

5

node->next = \*head\_ptr;

6

\*head\_ptr = node;

7

}

0x300

node 0x350 NULL 0x300

0x350

head\_ptr 0x300 data 'a' 'b' 'c' '\0'

'a' 'b' 'c' '\0'

Garcia, Kao

**05 Memory (Mis) Management** (25) 

**Linked List Example**

# include <string.h>

int main() {

struct Node \*head = NULL;

1

add\_to\_front(&head, "abc");

2

… // free nodes, strings here… }

struct Node {

char \*data;

struct Node \*next; };

void add\_to\_front(struct Node \*\*head\_ptr, char \*data) { struct Node \*node = (struct Node \*) malloc(sizeof(struct Node));

3

node->data = (char \*) malloc(strlen(data) + 1); // extra byte!

4

strcpy(node->data, data); // strcpy also copies null terminator

5

node->next = \*head\_ptr;

6

\*head\_ptr = node;

7

}

Check out the

0x300

0x350 NULL

head

0x300

lecture code in Drive!

0x350

'a' 'b' 'c' '\0'

'a' 'b' 'c'

'\0'

Garcia, Kao

**05 Memory (Mis) Management** (26) 

**Agenda** 

**When**

**Memory Goes Bad**

• Memory Locations

• The Stack

• The Heap

• Linked List Example

• When Memory Goes Bad • Implementing Memory

Management

Garcia, Kao

**05 Memory (Mis) Management** (27) 

**Working with Memory** Code, Static storage are easy:

• They never grow or shrink.

Stack space is also easy:

• Stack frames are created and destroyed in LIFO order.

**stack** 0xFFFF FFFF 

• Just avoid “**dangling references**”: pointers to deallocated variables (e.g., from old stack frames).

⚠ Working with the heap is tricky:

• Memory can be allocated / deallocated at any time!

heap data

• **“Memory leak”: If you forget to deallocate memory**

run out of memorytext

Your program will eventually

0x0000 0000

• **“Use after free”: If you use data after calling free** • **“Double free”: If you call free 2x on same memory**

Possible crash or

exploitable vulnerability

Garcia, Kao

**05 Memory (Mis) Management** (28) 

**Failure to** free()

• The runtime does not check for the programmer’s failure to manage memory.

• Memory is so performance-critical that there just isn’t time to do this. • Usual result: you corrupt the memory allocator’s internal structure, and you find out much later in a totally unrelated part of your code!

• Memory leak: Failure to free() allocated memory � Initial symptoms. Nothing…

■ Until you hit a critical point, memory leaks aren’t actually a problem

� …Later symptoms: performance drops off a cliff…

■ Memory hierarchy behavior tends to be great just up until it isn’t, then it hits several cliffs

� …and then your program is killed off!

■ Because the operating system (OS) says “no” when you ask for more memory

Garcia, Kao

**05 Memory (Mis) Management** (29) 

**Use after Free** • “**Dangling reference**”

When you keep using a pointer, even after it has been deallocated

• Reads after the free may be corrupted!

• If something else takes over that memory, your program will probably read the wrong information!

• Writes corrupt other data!

• Uh oh... Your program crashes later!



struct foo \*f;

…

f = malloc(sizeof(struct foo)); …

free(f);

…

bar(f->a); // !!!

Garcia, Kao

**05 Memory (Mis) Management** (30) 

**Double-Free... **

struct foo \*f = (struct foo \*) malloc(10\*sizeof(struct foo)); ...

free(f);

...

free(f); // !!!

• May cause either a use-after-free (because something else called malloc() and got that data) or corrupt heap data (because you are no longer freeing a pointer tracked by malloc)

Garcia, Kao

**05 Memory (Mis) Management** (31) 

**Forgetting** realloc() **Can Move Data **• ”Dangling reference”

• Remember, when you realloc it can copy data to a different part of the

heap.

int \*nums;

nums = malloc(10\*sizeof(int)); …

int \*g = nums;

…

nums = realloc(nums,

20\*sizeof(int));

// g could now point

// to invalid memory

int \*nums;

nums = malloc(10\*sizeof(int)); …

// forget to update nums

// on realloc call

realloc(nums, 20\*sizeof(int));

// nums could now point

// to invalid memory,

// and we could have potentially lost a pointer to a new block

• Reads may be corrupted, and writes may corrupt other pieces of memory. Garcia, Kao

**05 Memory (Mis) Management** (32) 

**Faulty Heap Management**

[Concept Check]

How many memory management errors are in this code?

void free\_mem\_x() {

int fnh[3];

**[for next time]**

...

free(fnh);

}

void free\_mem\_y() {

int \*fum = malloc(4\*sizeof(int)); free(fum+1);

...

free(fum);

...

free(fum);

}

A. 1

B. 2

C. 3

D. 0

E. Other

Garcia, Kao

**05 Memory (Mis) Management** (33) 



Garcia, Kao

**05 Memory (Mis) Management** (34) 

**Faulty Heap Management**

[Concept Check]

How many memory management errors are in this code?

void free\_mem\_x() {

int fnh[3];

**[for next time]**

...

free(fnh);

}

void free\_mem\_y() {

free() on stack-allocated memory

A. 1 B. 2 C. 3 D. 0

int \*fum = malloc(4\*sizeof(int));

E. Other

free(fum+1); ...

free(fum); ...

free(fum); }

free() on memory that isn’t the pointer from malloc

Double free()

Garcia, Kao

**05 Memory (Mis) Management** (35) 

**Valgrind to the rescue!!!**

• Valgrind slows down your program by an order of magnitude, but... • It adds a tons of checks designed to catch most (but not all) memory errors • Memory leaks

• Misuse of free

• Writing over the end of arrays

• Tools like Valgrind are absolutely essential for debugging C code.

Check out Lab 02!

Garcia, Kao

**05 Memory (Mis) Management** (36) 

**And in Conclusion…**

• C has 3 pools of memory for variables:

• Data: **global/static** variable storage, basically permanent

• Stack: **local variable** storage, parameters, return address

• Heap (**dynamic** storage): malloc() grabs space from here, free() returns it. • (4th memory pool: text, for the program executable itself)

• Heap data is biggest source of bugs in C code!

�

Garcia, Kao

**05 Memory (Mis) Management** (37) 

**Agenda** 

**Implementing Memory**

**Management**

Material not tested. Recording:

https://www.youtube.com/watch?v=Sq5tSeWfnGY

• Memory Locations

• The Stack

• The Heap

• Linked List Example

• When Memory Goes Bad • Implementing Memory

Management

Garcia, Kao

**05 Memory (Mis) Management** (38) 

**Heap Management Requirements** • Want malloc() and free() to run quickly

• Want minimal memory overhead

• Want to avoid fragmentation\*,

when most of our free memory is in many small chunks • In this case, we might have many free bytes but not be able to satisfy a large request since the free bytes are not contiguous in memory.

\* This is technically

external fragmentation

Garcia, Kao

**05 Memory (Mis) Management** (39) 

**Heap Management Example** R2 (10 bytes) R2 (10 bytes)

R3?

R2 (10 bytes)

Heap grows up

R1 (100 bytes)

Request R1, 100 bytes

R1 (100 bytes)

Request R2, 10 bytes

Memory from R1 is freed

R3?

Request R3, 50 bytes?

Garcia, Kao

**05 Memory (Mis) Management** (40) 

**K&R Malloc/Free Implementation**• From Section 8.7 of K&R

• Code in the book uses some C language features we haven’t discussed and is written in a very terse style; don’t worry if you can’t decipher the code • Each block of memory is preceded by a header that has two fields: • size of the block, and

• a pointer to the next block

• All free blocks are kept in a circular linked list.

• In an allocated block, the header’s pointer field is unused.

Garcia, Kao

**05 Memory (Mis) Management** (41) 

**K&R Malloc/Free Implementation**• malloc() searches the free list for a block that is big enough. If none is found, more memory is requested from the operating system. If what it gets can’t satisfy the request, it fails.

• free() checks if the blocks adjacent to the freed block are also free. • If so, adjacent free blocks are merged (**coalesced**) into a single, larger free block. • Otherwise, freed block is just added to the free list.

Garcia, Kao

**05 Memory (Mis) Management** (42) 

**Choosing a block in** malloc()

• If there are multiple free blocks of memory that are big enough for some request, how do we choose which one to use? • **best-fit**: choose the smallest block that is big enough for the request. • **first-fit**: choose the first block we see that is big enough.

• **next-fit**: like first-fit, but remember where we finished searching and resume searching from there.

Garcia, Kao

**05 Memory (Mis) Management** (43) 