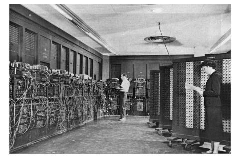
CS61C

UC Berkeley
Teaching Professor
Dan Garcia

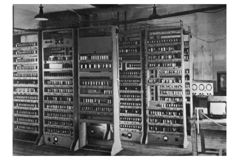
Great Ideas in Computer Architecture
(a.k.a. Machine Structures)

Memory (Mis)Management

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** words

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

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;
}
```

Array has decayed to a pointer

In array's declared scope, total array size.

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

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

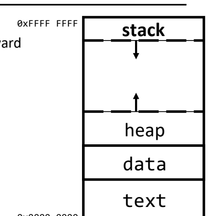
Agenda

Memory Locations

- Memory Locations
- The Stack
- The Heap
- Linked List Example
- When Memory Goes Bad
- Implementing Memory Management

C Program Address Space

- A program's address space contains 4 regions:
  - **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
- Programming in C requires knowing where objects are in memory, otherwise things don't work as expected.
  - By contrast, Java hides location of objects.

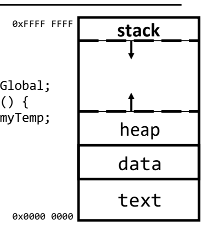


For now, OS somehow prevents accesses between stack and heap. (more later w/virtual memory)

Where are variables allocated?

- **Global**: If declared **outside** a function, allocated in **data (static)** storage.
- **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.
  - But a variable **does not exist anymore** once a function ends!

```
int myGlobal;
… main() {
  int myTemp;
…
}
```



Agenda

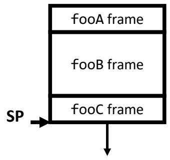
The Stack

- Memory Locations
- The Stack
- The Heap
- Linked List Example
- When Memory Goes Bad
- Implementing Memory Management

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.
- 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 processor architecture)

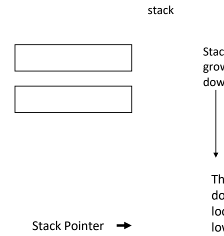
```
fooA() { fooB(); }
fooB() { fooC(); }
fooC() { … }
```



*see slide notes in pptx for technical elaboration

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)
       {
       }
```



The stack grows downward; a()'s local variables have lower byte addresses than main()'s, and so on.

Recall: Array Are Very Primitive

[From last time]

1. Array bounds are not checked during element access.
   - 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)

Passing Pointers into the Stack

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

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

```
void load_buf() { … };
int main() {
   …
   char buf[...];
   load_buf(buf, BUFLEN);
   …
}
```

stack
buf char array persistent through load_buf's execution

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!

```
char *make_buf() {
    char buf[50];
    return buf;
}
void foo() {…}
int main(){
   char *stackAddr = \
      make_buf();
   foo(stackAddr);
   …
}
```

stack
stackAddr points to overwritten memory
~~buf???~~

Agenda

The Heap

- Memory Locations
- The Stack
- The Heap
- Linked List Example
- When Memory Goes Bad
- Implementing Memory Management

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

`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 *`)

`sizeof(type)` gives size in bytes.

- To allocate an array of 20 `int`s:

```
int *ptr = (int *) malloc(20*sizeof(int));
if (ptr != NULL) { … // always check NULL after
```

  - Many years ago `int`s used to be 16b. Now, 32b or 64b…

Assuming size of objects can lead to misleading, unportable code. Use `sizeof()`!

`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!)

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 <u>may</u> 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); */
```


Agenda

Linked List Example

- Memory Locations
- The Stack
- The Heap
- Linked List Example
- When Memory Goes Bad
- Implementing Memory Management

Linked List Example

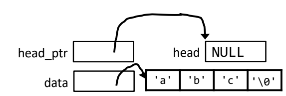
```
# include <string.h>
int main() {
  struct Node *head = NULL;
  add_to_front(&head, "abc");
  …  // free nodes, strings here…
}
void add_to_front(struct Node **head_ptr, char *data) {
  struct Node *node = (struct Node *) malloc(sizeof(struct Node));
  node->data = (char *) malloc(strlen(data) + 1); // extra byte!
  strcpy(node->data, data);  // strcpy also copies null terminator
  node->next = *head_ptr;
  *head_ptr = node;
}
```

```
struct Node {
  char *data;
  struct Node *next;
};
```


Linked List Example

```
# include <string.h>
int main() {
  struct Node *head = NULL;
  add_to_front(&head, "abc");
  …  // free nodes, strings here…
}
void add_to_front(struct Node **head_ptr, char *data) {
  struct Node *node = (struct Node *) malloc(sizeof(struct Node));
  node->data = (char *) malloc(strlen(data) + 1); // extra byte!
  strcpy(node->data, data);  // strcpy also copies null terminator
  node->next = *head_ptr;
  *head_ptr = node;
}
```

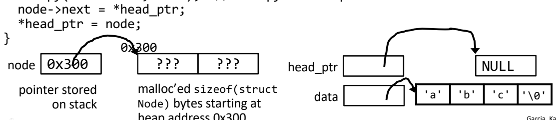
```
struct Node {
  char *data;
  struct Node *next;
};
```



Linked List Example

```
# include <string.h>
int main() {
  struct Node *head = NULL;
  add_to_front(&head, "abc");
  …  // free nodes, strings here…
}
void add_to_front(struct Node **head_ptr, char *data) {
  struct Node *node = (struct Node *) malloc(sizeof(struct Node));
  node->data = (char *) malloc(strlen(data) + 1); // extra byte!
  strcpy(node->data, data);  // strcpy also copies null terminator
  node->next = *head_ptr;
  *head_ptr = node;
}
```

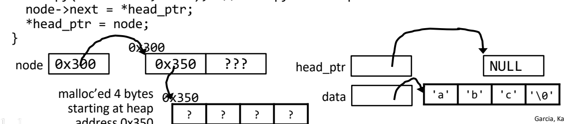
```
struct Node {
  char *data;
  struct Node *next;
};
```



Linked List Example

```
# include <string.h>
int main() {
  struct Node *head = NULL;
  add_to_front(&head, "abc");
  …  // free nodes, strings here…
}
void add_to_front(struct Node **head_ptr, char *data) {
  struct Node *node = (struct Node *) malloc(sizeof(struct Node));
  node->data = (char *) malloc(strlen(data) + 1); // extra byte!
  strcpy(node->data, data);  // strcpy also copies null terminator
  node->next = *head_ptr;
  *head_ptr = node;
}
```

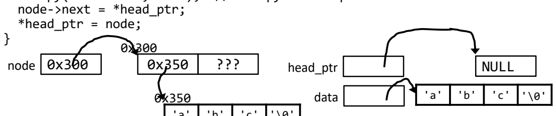
```
struct Node {
  char *data;
  struct Node *next;
};
```



Linked List Example

```
# include <string.h>
int main() {
  struct Node *head = NULL;
  add_to_front(&head, "abc");
  …  // free nodes, strings here…
}
void add_to_front(struct Node **head_ptr, char *data) {
  struct Node *node = (struct Node *) malloc(sizeof(struct Node));
  node->data = (char *) malloc(strlen(data) + 1); // extra byte!
  strcpy(node->data, data);  // strcpy also copies null terminator
  node->next = *head_ptr;
  *head_ptr = node;
}
```

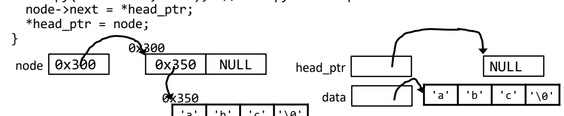
```
struct Node {
  char *data;
  struct Node *next;
};
```



Linked List Example

```
# include <string.h>
int main() {
  struct Node *head = NULL;
  add_to_front(&head, "abc");
  …  // free nodes, strings here…
}
void add_to_front(struct Node **head_ptr, char *data) {
  struct Node *node = (struct Node *) malloc(sizeof(struct Node));
  node->data = (char *) malloc(strlen(data) + 1); // extra byte!
  strcpy(node->data, data);  // strcpy also copies null terminator
  node->next = *head_ptr;
  *head_ptr = node;
}
```

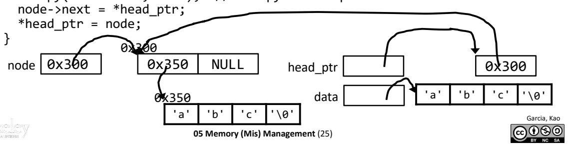
```
struct Node {
  char *data;
  struct Node *next;
};
```



Linked List Example

```
# include <string.h>
int main() {
  struct Node *head = NULL;
  add_to_front(&head, "abc");
  … // free nodes, strings here…
}
void add_to_front(struct Node **head_ptr, char *data) {
  struct Node *node = (struct Node *) malloc(sizeof(struct Node));
  node->data = (char *) malloc(strlen(data) + 1); // extra byte!
  strcpy(node->data, data);  // strcpy also copies null terminator
  node->next = *head_ptr;
  *head_ptr = node;
}
```

```
struct Node {
  char *data;
  struct Node *next;
};
```

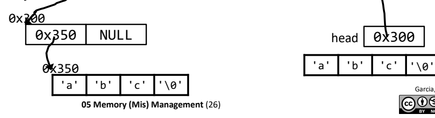


Linked List Example

```
# include <string.h>
int main() {
  struct Node *head = NULL;
  add_to_front(&head, "abc");
  … // free nodes, strings here…
}
void add_to_front(struct Node **head_ptr, char *data) {
  struct Node *node = (struct Node *) malloc(sizeof(struct Node));
  node->data = (char *) malloc(strlen(data) + 1); // extra byte!
  strcpy(node->data, data);  // strcpy also copies null terminator
  node->next = *head_ptr;
  *head_ptr = node;
}
```

```
struct Node {
  char *data;
  struct Node *next;
};
```

Check out the lecture code in Drive!



Agenda

When Memory Goes Bad

- Memory Locations
- The Stack
- The Heap
- Linked List Example
- When Memory Goes Bad
- Implementing Memory Management

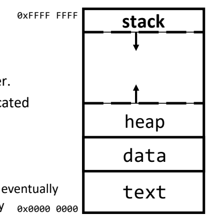
Working with Memory

Code, Static storage are easy:
- They never grow or shrink.

Stack space is also easy:
- Stack frames are created and destroyed in LIFO order.
- 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!
- **"Memory leak": If you forget to deallocate memory** } Your program will eventually run out of memory
- **"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



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

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); // !!!
```


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`)

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.

Faulty Heap Management

[Concept Check]

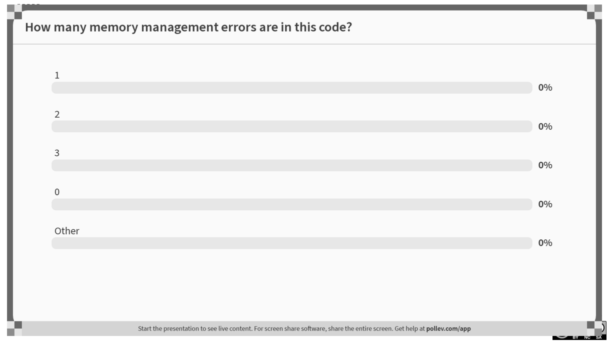
How many memory management errors are in this code?

[for next time]

```
void free_mem_x() {
  int fnh[3];
  ...
  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



Faulty Heap Management

[Concept Check]

How many memory management errors are in this code?

[for next time]

```
void free_mem_x() {
  int fnh[3];
  ...
  free(fnh);        } free() on stack-allocated memory
}

void free_mem_y() {
  int *fum = malloc(4*sizeof(int));
  free(fum+1);      } free() on memory that isn't the pointer from malloc
  ...
  free(fum);
  ...
  free(fum);        } Double free()
}
```

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

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!

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!

☑

Agenda

Implementing Memory Management

- Memory Locations
- The Stack
- The Heap
- Linked List Example
- When Memory Goes Bad
- Implementing Memory Management

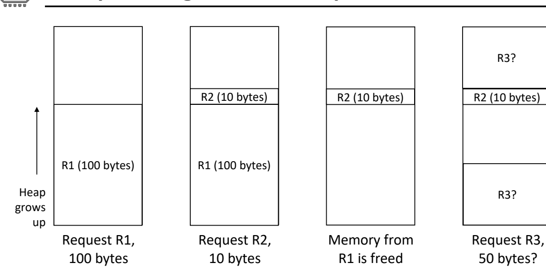
Material not tested. Recording:
https://www.youtube.com/watch?v=Sq5tSeWfnGY

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

Heap Management Example



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.

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.

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.