CS 61C:
Great Ideas in Computer Architecture

Lecture 4: Memory Management

Krste Asanović & Randy Katz

http://inst.eecs.berkeley.edu/~cs61c

Agenda

- **Pointers to Pointers**
- Strings in C
- C Memory Management
- Stack
- Heap
- Implementations of malloc/free
- Common memory problems & how to avoid/find them
- And in Conclusion, …

Pointers to Pointers

```c
#include <stdio.h>

// changes value of pointer
void next_el(int **h) {
    *h = *h + 1;
}

int main(void) {
    int A[] = { 10, 20, 30 };
    // p points to first element of A
    int *p = A;
    next_el(&p);
    // now p points to 2nd element of A
    printf("*p = %d\n", *p);
}
```

Your Turn …

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

Answer	
RED	2
GREEN	3
ORANGE	4
YELLOW	5

Name	Type	Addr	Value
		…	
		106	
		105	
		104	
		103	
		102	
		101	
		100	
		…	

Agenda

- Pointers to Pointers
- **Strings in C**
- C Memory Management
- Stack
- Heap
- Implementations of **`malloc/free`**
- Common memory problems & how to avoid/find them
- And in Conclusion, …

C Strings

- C strings are null-terminated character arrays
  - `char s[] = "abc";`

Type	Name	Byte Addr	Value
		…	
		108	
		107	
		106	
		105	
		104	
		103	
		102	
		101	
		100	
		…	

String Example

```
#include <stdio.h>

int slen(char s[]) {
    int n = 0;
    while (s[n] != 0) n++;
    return n;
}

int main(void) {
    char str[] = "abc";
    printf("str = %s,  length = %d\n", str, slen(str));
}
```

Output: str = abc, length = 3

Concise strlen()

```
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?

Arguments in `main()`

- To get arguments to the main function, use:
  - `int main(int argc, char *argv[])`
  - `argc` is the *number* of strings on the command line
  - `argv` is a *pointer* to an array containing the arguments as strings

```
#include <stdio.h>

int main(int argc, char *argv[]) {
    for (int i=0;  i<argc;  i++)
        printf("arg[%d] = %s\n", i, argv[i]);
}
```

Example

```c
#include <stdio.h>

int main(int argc, char *argv[]) {
    for (int i=0;  i<argc;  i++)
        printf("arg[%d] = %s\n", i, argv[i]);
}
```

UNIX:

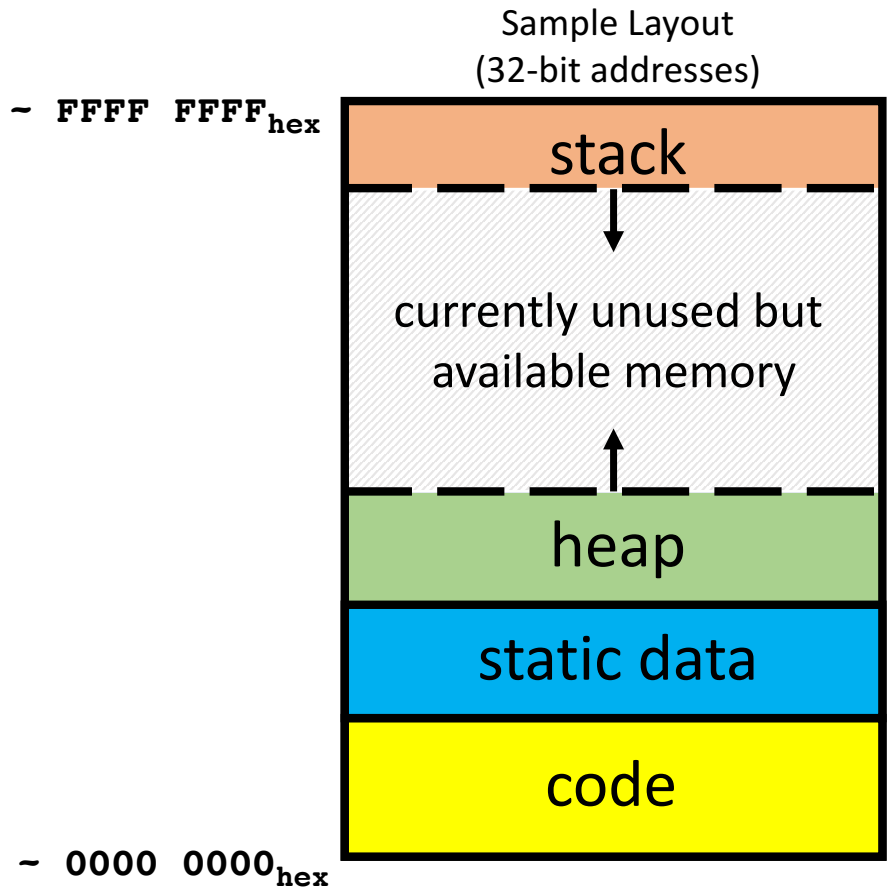
```
$ gcc -o ex Argc.c
$ ./ex -g a "d e f"
arg[0] = ./ex
arg[1] = -g
arg[2] = a
arg[3] = d e f
```

Agenda

- Pointers Wrap-up
- Strings in C
- **C Memory Management**
- Stack
- Heap
- Implementations of **`malloc/free`**
- Common memory problems & how to avoid/find them
- And in Conclusion, …

C Memory Management

- How does the C compiler determine where to put code and data in the machine's memory?
- How can we create dynamically sized objects?
  - E.g. array of variable size depending on requirements



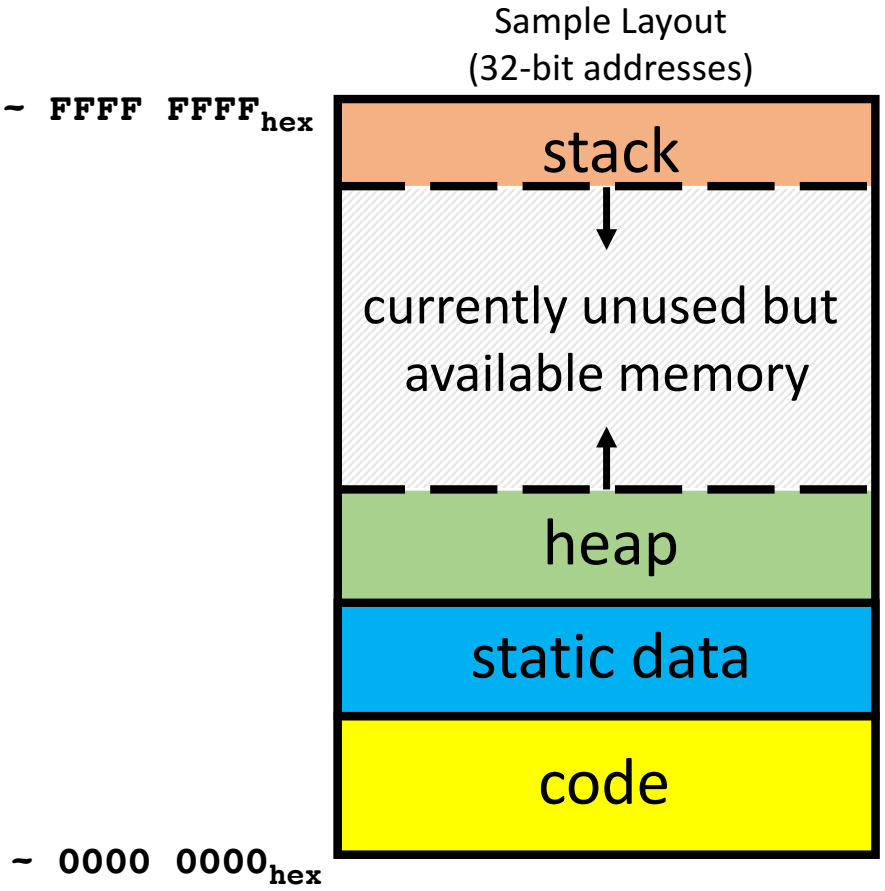
C Memory Management: Code

- Code
  - Loaded when program starts
  - Does not change

```
int static_data = 55;

void f(char c) {
    int local_data = 4;
    int *heap_data
        = malloc(50*sizeof(int));
}

int main(void) {
    int local_data;
    char more_local_data = 'X';
    f(more_local_data);
}
```



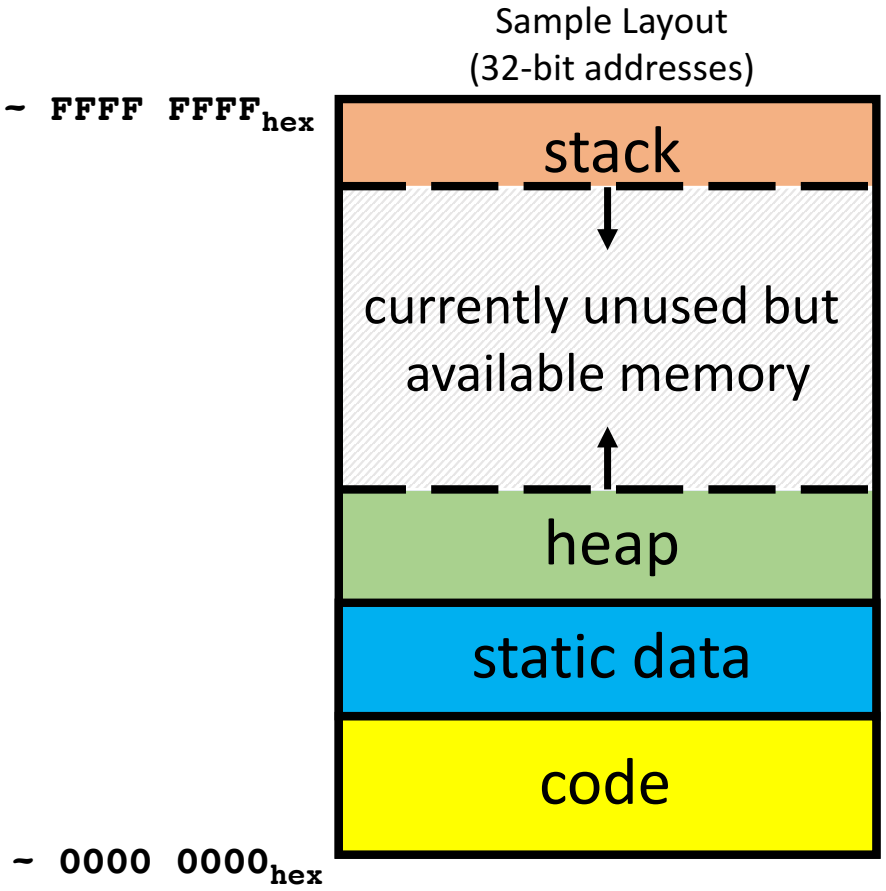
C Memory Management: Static Data

- Static Data
  - Loaded when program starts
  - Can be modified
  - Size is fixed

```
int static_data = 55;

void f(char c) {
    int local_data = 4;
    int *heap_data
        = malloc(50*sizeof(int));
}

int main(void) {
    int local_data;
    char more_local_data = 'X';
    f(more_local_data);
}
```



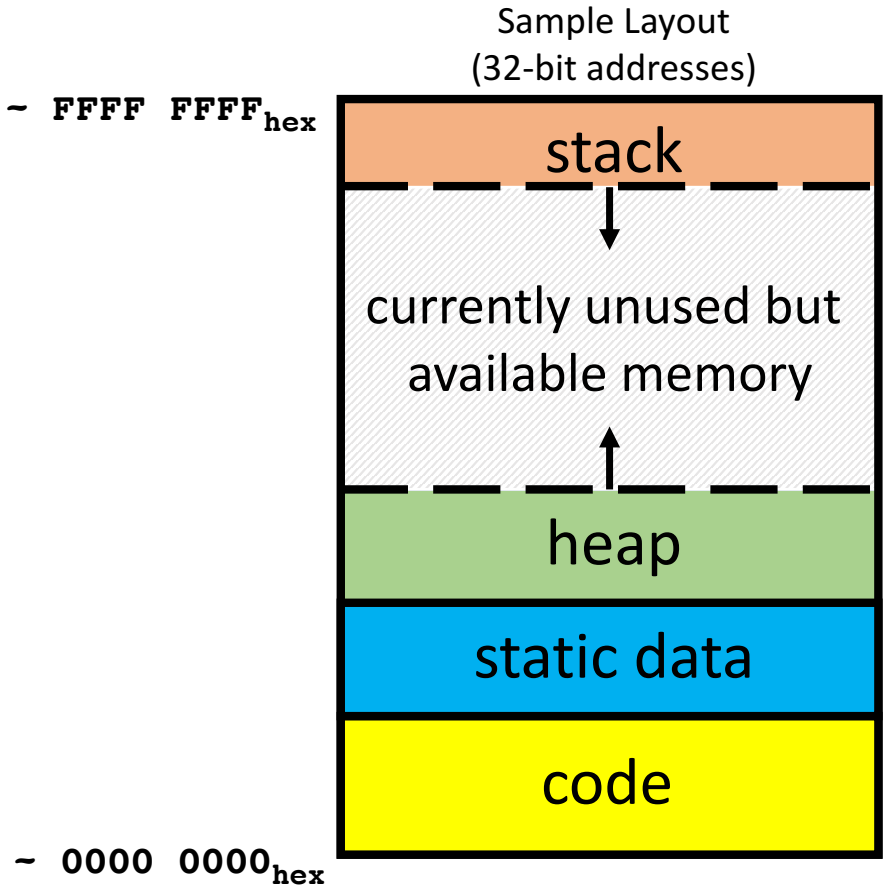
C Memory Management: Stack

- Stack
  - Local variables & arguments inside functions
  - Allocated when function is called
  - Stack usually grows downward

```
int static_data = 55;

void f(char c) {
    int local_data = 4;
    int *heap_data
        = malloc(50*sizeof(int));
}

int main(void) {
    int local_data;
    char more_local_data = 'X';
    f(more_local_data);
}
```



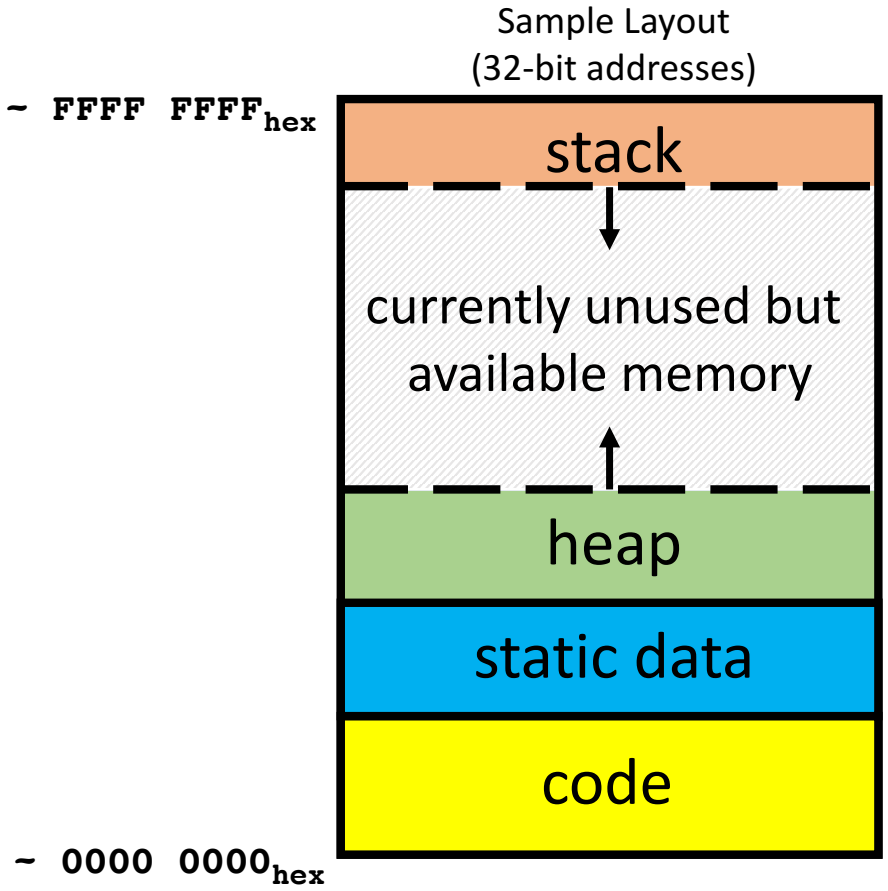
C Memory Management: Heap

- Heap
  - Space for dynamic data
  - Allocated and freed by program as needed

```
int static_data = 55;

void f(char c) {
    int local_data = 4;
    int *heap_data
        = malloc(50*sizeof(int));
}

int main(void) {
    int local_data;
    char more_local_data = 'X';
    f(more_local_data);
}
```



Agenda

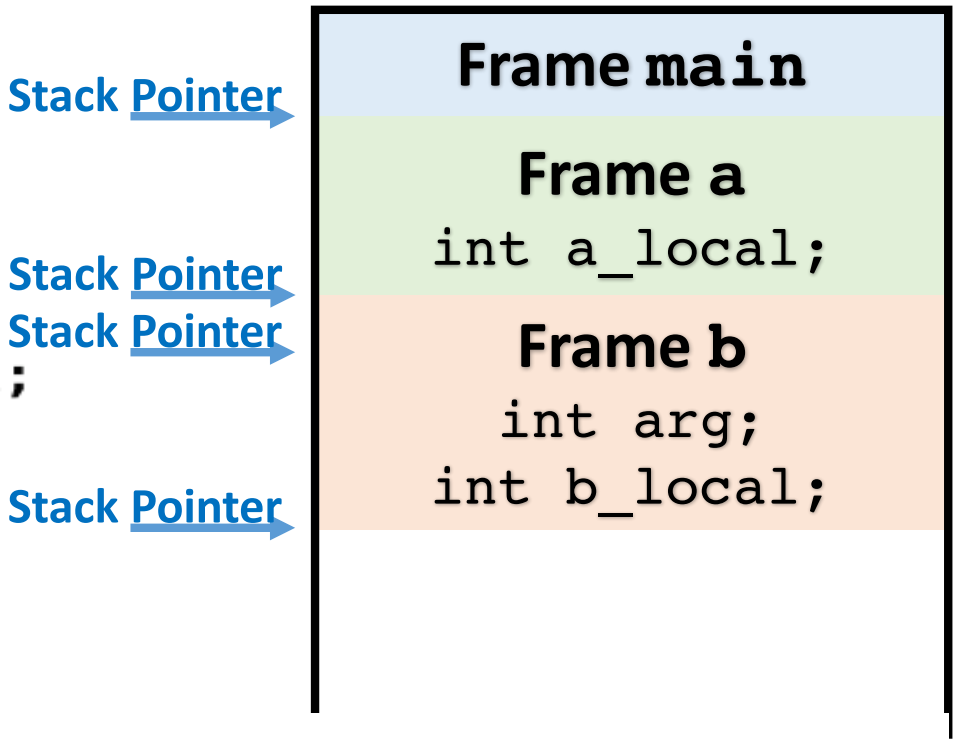
- Pointers Wrap-up
- Strings in C
- C Memory Management
- **Stack**
- Heap
- Implementations of **`malloc/free`**
- Common memory problems & how to avoid/find them
- And in Conclusion, …

Stack

```
void a() {
    int a_local = 0;
    b(a_local);
}

int b(int arg) {
    int b_local = 5;
    return 2*arg + b_local;
}

int main(void) {
    a();
    b(7);
}
```



Stack

- Every time a function is called, a new frame is allocated
- When the function returns, the frame is deallocated
- Stack frame contains
  - Function arguments
  - Local variables
  - Return address (who called me?)
- Stack uses contiguous blocks of memory
  - Stack pointer indicates current level of stack
- Stack management is transparent to C programmer
  - We'll see details when we program assembly language

Your Turn …

```c
#include <stdio.h>
#include <stdlib.h>

int x = 2;

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

int main(void) {
    foo(10);
}
```

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

Answer	#x	#y
RED	1	1
GREEN	1	6
ORANGE	1	5
YELLOW	6	6

What's wrong with this Code?

```
#include <stdio.h>
#include <math.h>

int *f() {
    int x = 5;
    return &x;
}

int main(void) {
    int *a = f();
    // ... some calculations ...
    double d = cos(1.57);
    // ... now use *a ...
    printf("a = %d\n", *a);
}
```

Output:

```
a = -1085663214
```

- `*a` is a pointer to a local variable
  - allocated on the stack
  - "deallocated" when `f()` returns
  - stack reused by other functions
    - e.g. `cos`
    - which overwrite whatever was there before
  - `*a` points to "garbage"
- Obscure errors
  - depend on what other functions are called after `f()` returns
  - assignments to `*a` corrupt the stack and can result in even more bizarre behavior than this example
  - errors can be difficult to reproduce

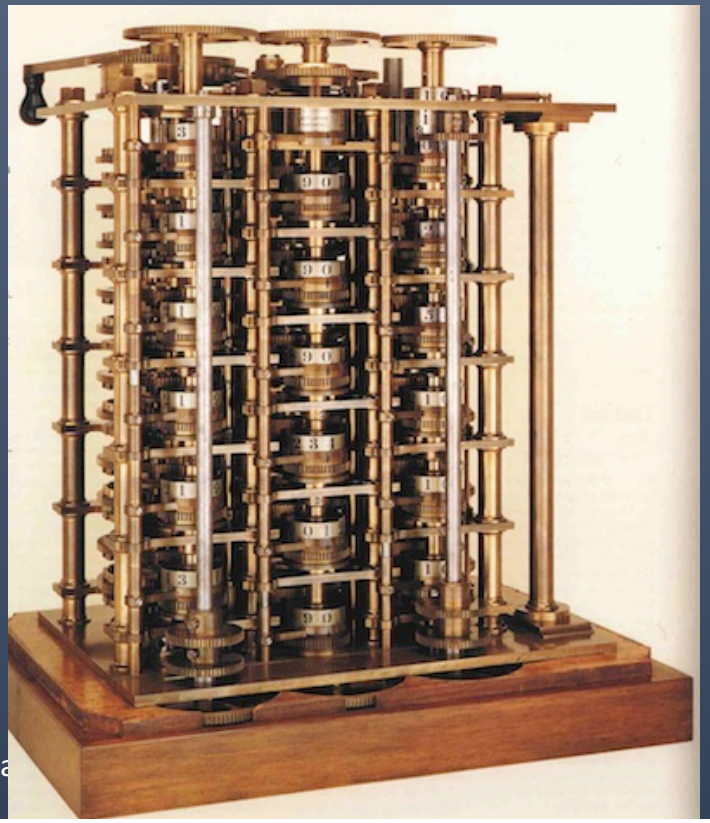
Agenda

- Pointers Wrap-up
- Strings in C
- C Memory Management
- Stack
- **Not on the test!**
- Heap
- Implementations of **`malloc/free`**
- Common memory problems & how to avoid/find them
- And in Conclusion, …

Early Memory Technology

Punched cards, From early 1700s through Jaquard Loom, Babbage, and then IBM

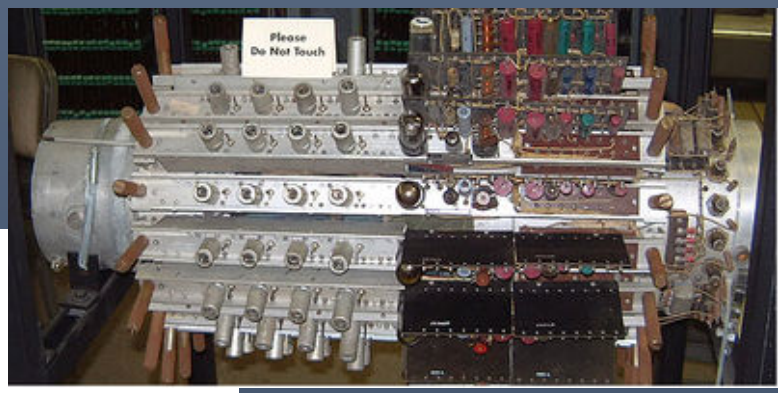
Babbage, 1800s: Digits stored on mechanical wheels

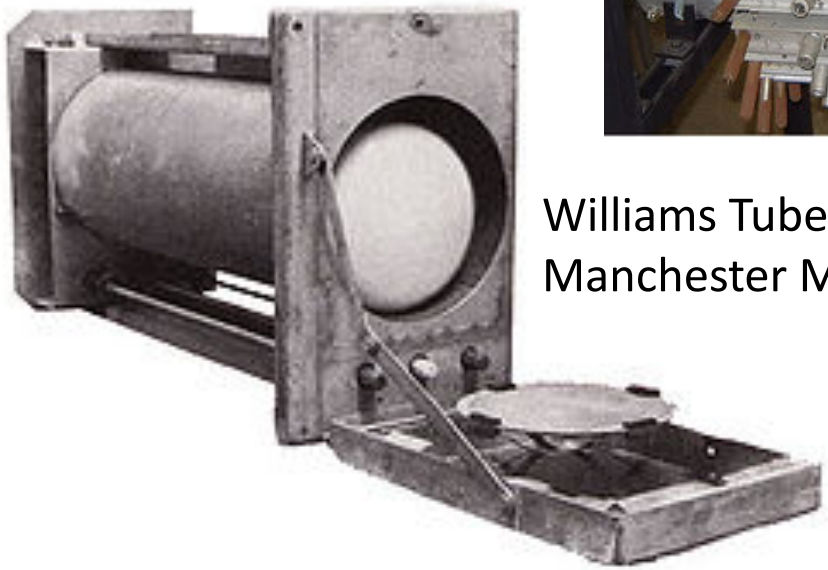


Early Memory technology

Mercury Delay Line, Univac 1, 1951

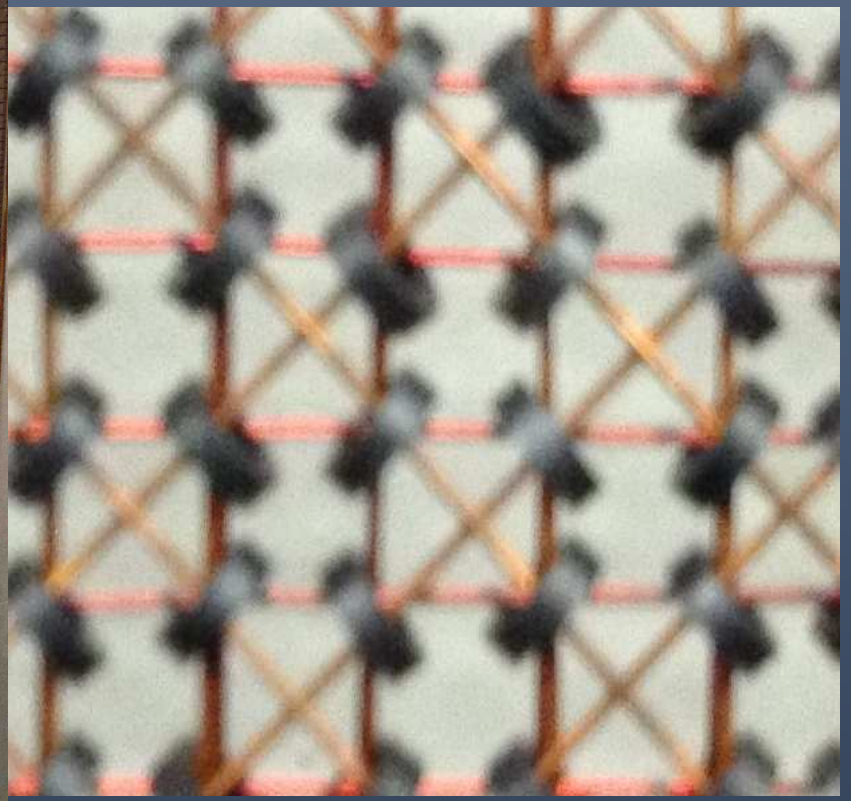
Punched paper tape, instruction stream in Harvard Mk 1



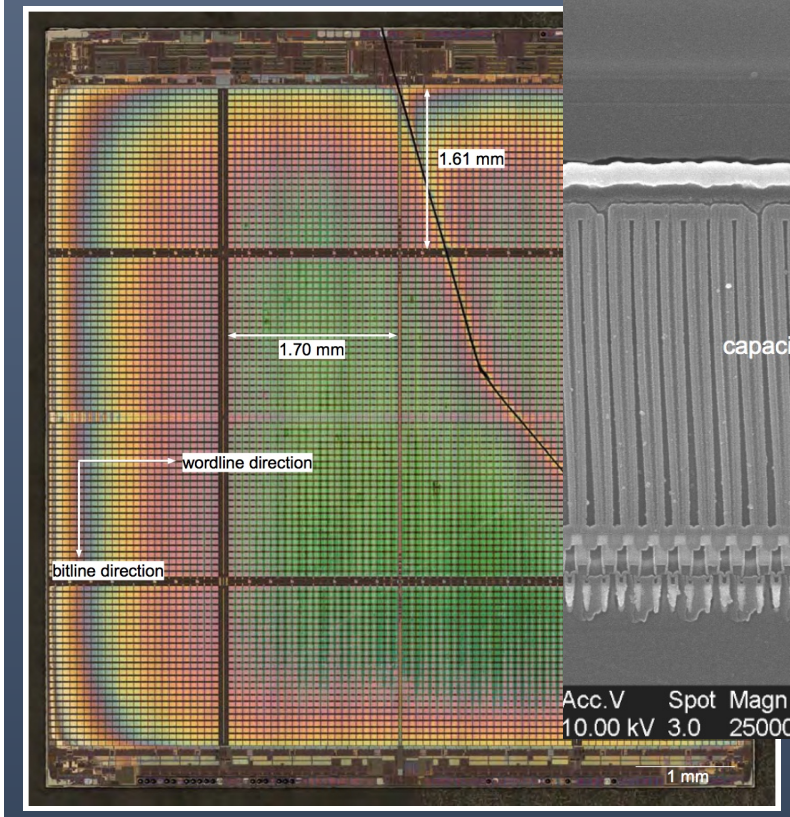


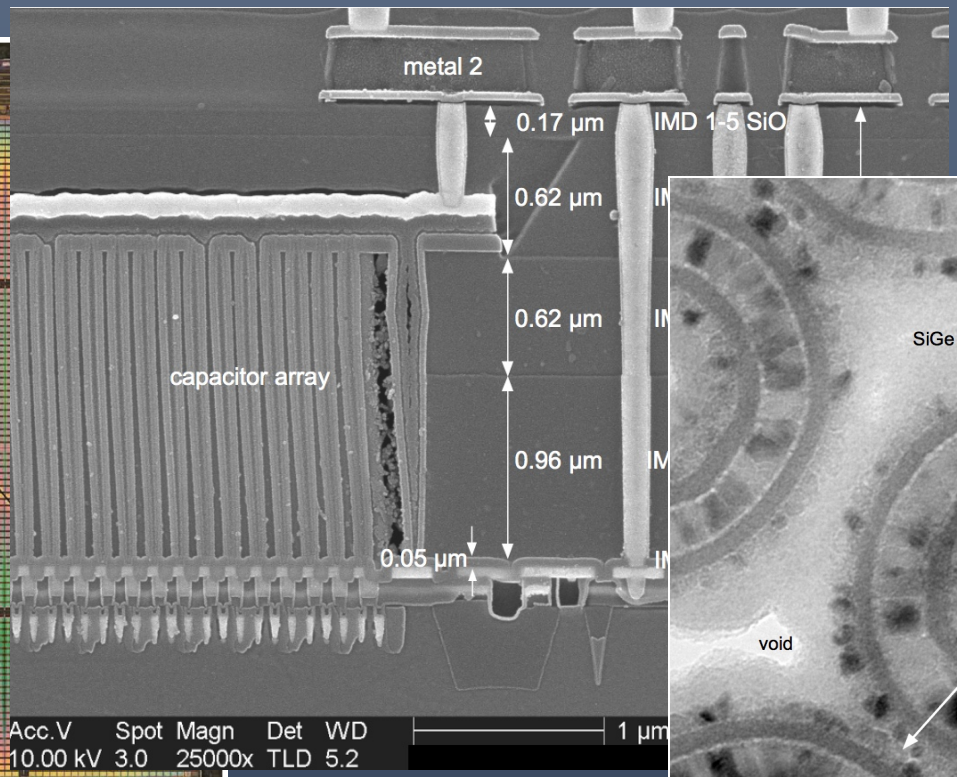
Williams Tube, Manchester Mark 1, 1947

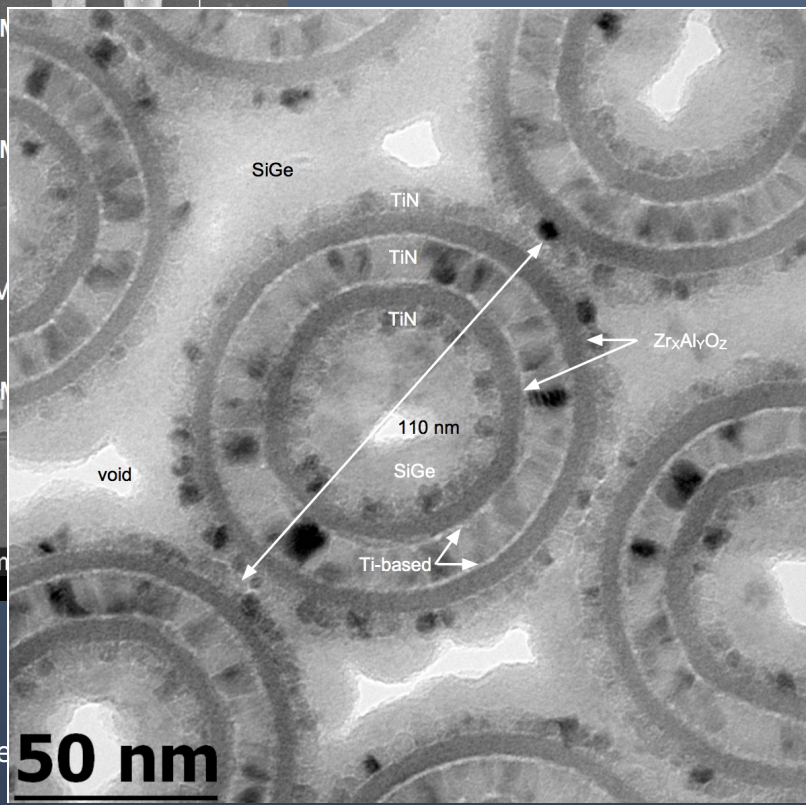
MIT Whirlwind Core Memory



Modern DRAM Technology







46nm DDR2 DRAM [© Chipworks]

Break!

Agenda

- Pointers Wrap-up
- Strings in C
- C Memory Management
- Stack
- **Heap**
- Implementations of **`malloc/free`**
- Common memory problems & how to avoid/find them
- And in Conclusion, …

Managing the Heap

C 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
  - **Beware**:
    previously allocated contents might move!

Malloc()

- **`void *malloc(size_t n)`**:
  - Allocate a block of uninitialized memory
  - **`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

- Example:

```
#include <stdlib.h>

int main(void) {
    // array of 50 ints ...
    int *ip = (int*)malloc(50*sizeof(int));

    // typecast is optional
    double *dp = malloc(1000*sizeof(double));
}
```

What's wrong with this Code?

```c
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    const int SZ = 10;
    int *p = malloc(SZ*sizeof(int));
    int *end = &p[SZ];
    while (p < end) *p++ = 0;
    free(p);
}
```

```
$ gcc FreeBug.c; ./a.out
a.out(23562,0x7fff78748000) malloc:
*** error for object 0x7fdcb3403168:
    pointer being freed was not allocated
*** set a breakpoint in malloc_error_break to debug
Abort trap: 6
```

free()

- **`void free(void *p)`**:
  - Release memory allocated by **`malloc()`**
  - **`p` *must* contain address originally returned by `malloc()`**

- Example:

```
#include <stdlib.h>
#include <stdio.h>

int main(void) {
    double *dp = malloc(100*sizeof(double));
    // ... do something with *dp ...
    // ...
    // Give memory back when done!
    free(dp);   // free knows it's 100 doubles
}
```

Fix

```
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    const int SZ = 10;
    int *array = malloc(SZ*sizeof(int));
    for (int *p = array;  p<&array[SZ]; )
        *p++ = 0;
    free(array);
}
```

- Do not modify return value from `malloc`
  - You'll need it to call `free`!

Why Call free()?

- Recycle no-longer-used memory to avoid running out
- Two common approaches:
  - malloc/free (Explicit memory management)
    - C, C++, …
    - ”manually” take care of releasing memory
    - Requires some planning: how do I know that memory is no longer used? What if I forget to release?
    - Drawback: potential bugs
      - memory leaks (forgot to call free)
      - corrupted memory (accessing memory that is now longer “owned”)
  - garbage collector (Automatic memory management)
    - Java, Python, …
    - No-longer-used memory is free’d automatically
    - Drawbacks:
      - performance hit
      - unpredictable:
        - what if garbage collector starts when a self-driving car enters a turn at 100mph?

Out of Memory

- Insufficient free memory: **`malloc()`** returns **`NULL`**

```
int main(void) {
    const int G = 1024*1024*1024;
    for (int n=0; ;n++) {
        char *p = malloc(G*sizeof(char));
        if (p == NULL) {
            fprintf(stderr,
                    "failed to allocate > %g TiBytes\n", n/1000.0);
            return 1;  // abort program
        }
        // no free, keep allocating until out of memory
    }
}
```

```
$ gcc OutOfMemory.c; ./a.out
failed to allocate > 131.064 TiBytes
```

Example: Dynamically Allocated Tree

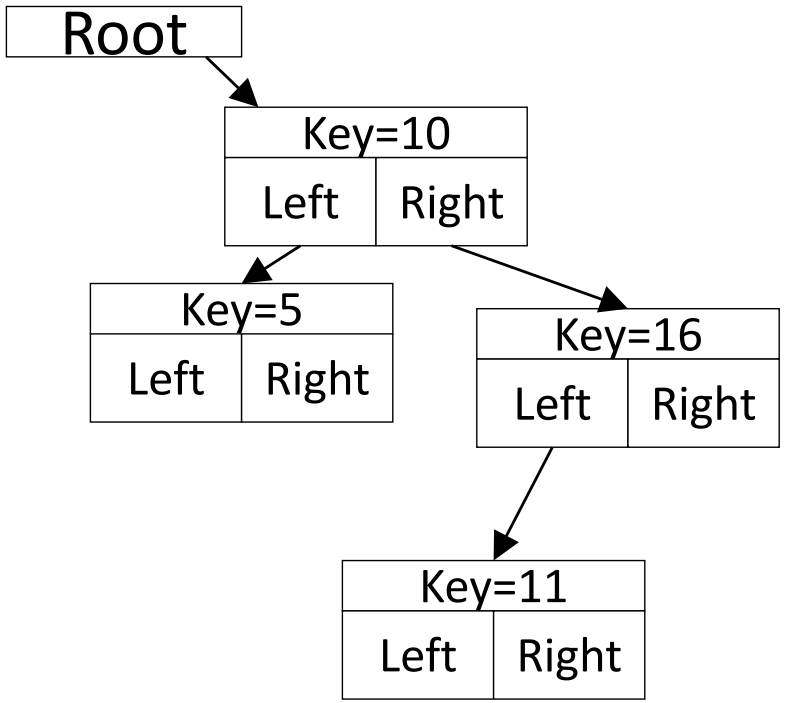
```c
#include <stdio.h>
#include <stdlib.h>

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

Node *create_node(int key, Node *left, Node *right) {
    Node *np;
    if ( (np = malloc(sizeof(Node))) == NULL) {
        printf("Out of Memory!\n"); 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));
}

int main(void) {
    Node *root = NULL;
    insert(10, &root);
    insert(16, &root);
    insert(5, &root);
    insert(11 , &root);
}
```



malloc and **free** are buddies!

malloc and **free**

- If you call **malloc** somewhere, you'll need to call **free** on the result (or accept having memory leak)
- E.g.

```
int *p = malloc(…);
// potentially very
// complicated code
// when done, call:
free(p);
```

no **malloc**, no **free**

- If you have not called **malloc**, do not call **free**!
- E.g.

```
int x;
int *p = &x;
// do whatever with p,
// but do not call free!
int a[5];
// do whatever with a, but
// do not call free(a)!
```

Observations

- Code, Static storage are easy:
  - they never grow or shrink
  - taken care of by OS
- Stack space is relatively easy:
  - stack frames are created and destroyed in last-in, first-out (LIFO) order
  - transparent to programmer
  - but don't hold onto it (with pointer) after function returns!
- *Managing the heap is the programmer's task*:
  - memory can be allocated / deallocated at any time
  - how tell / ensure that a block of memory is no longer used ***anywhere*** in the program?
  - requires planning before coding …

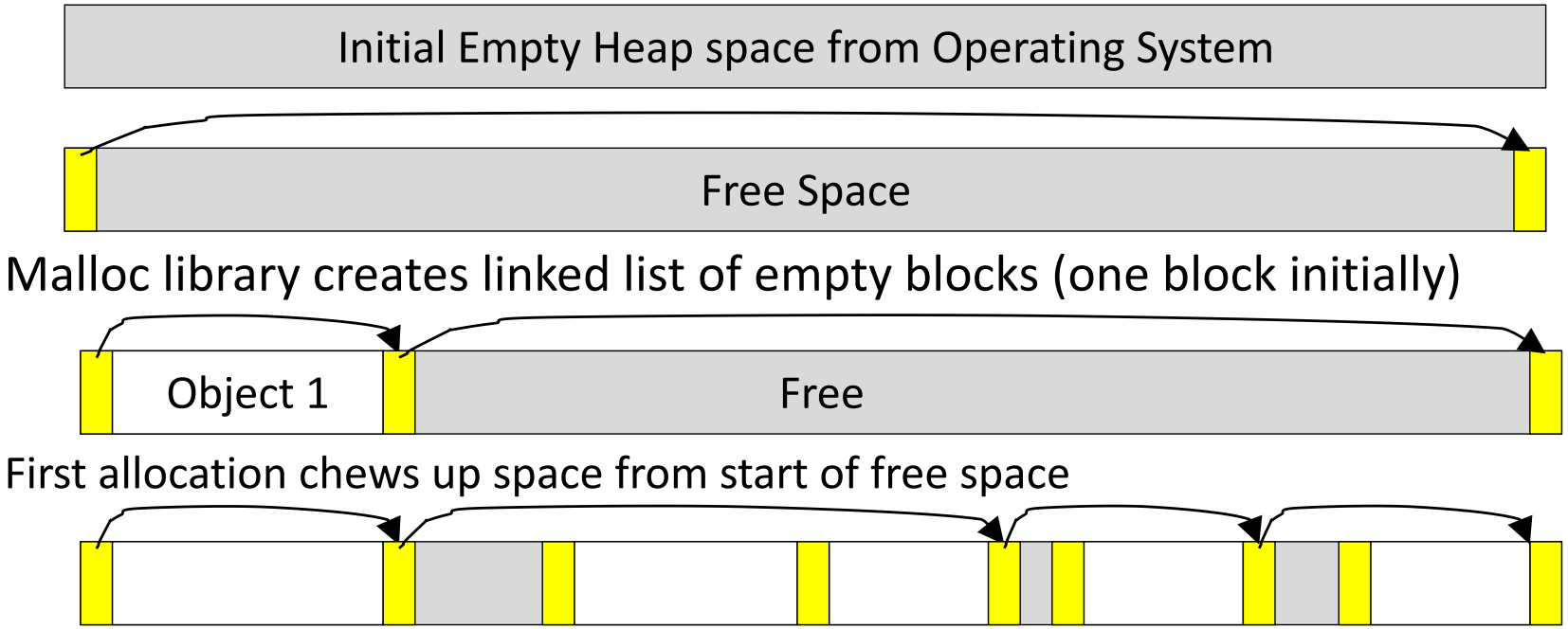
Agenda

- Pointers Wrap-up
- Strings in C
- C Memory Management
- Stack
- Heap
- **Implementations of `malloc/free`**
- Common memory problems & how to avoid/find them
- And in Conclusion, …

How are Malloc/Free implemented?

- Underlying operating system allows **`malloc`** library to ask for large blocks of memory to use in heap
  - E.g., using Unix **`sbrk()`** call
- C standard **`malloc`** library creates data structure inside unused portions of the heap to track free space
  - Writing to unallocated (or free'd) memory can corrupt this data structure.
  - Whose fault is it? The library's?

Simple **`malloc()`** Implementation



Problems after many malloc's and free's:

- Memory fragmentation (many small and no big free blocks, merge?)
- Long chain of blocks (slow to traverse)

Better `malloc` Implementations

- Keep separate pools of blocks for different sized objects
- E.g. “Buddy allocators” always round up to power-of-2 sized chunks to simplify finding correct size and merging neighboring blocks
  - https://en.wikipedia.org/wiki/Buddy_memory_allocation

Power-of-2 “Buddy Allocator”

Time

Step	64K	64K	64K	64K	64K	64K	64K	64K	64K	64K	64K	64K	64K	64K	64K	64K
1	2^4															
2.1	2^3								2^3							
2.2	2^2				2^2				2^3							
2.3	2^1		2^1		2^2				2^3							
2.4	2^0	2^0	2^1		2^2				2^3							
2.5	A: 2^0	2^0	2^1		2^2				2^3							
3	A: 2^0	2^0	B: 2^1		2^2				2^3							
4	A: 2^0	C: 2^0	B: 2^1		2^2				2^3							
5.1	A: 2^0	C: 2^0	B: 2^1		2^1		2^1		2^3							
5.2	A: 2^0	C: 2^0	B: 2^1		D: 2^1		2^1		2^3							
6	A: 2^0	C: 2^0	2^1		D: 2^1		2^1		2^3							
7.1	A: 2^0	C: 2^0	2^1		2^1		2^1		2^3							
7.2	A: 2^0	C: 2^0	2^1		2^2				2^3							
8	2^0	C: 2^0	2^1		2^2				2^3							
9.1	2^0	2^0	2^1		2^2				2^3							
9.2	2^1		2^1		2^2				2^3							
9.3	2^2				2^2				2^3							
9.4	2^3								2^3							
9.5	2^4															

https://en.wikipedia.org/wiki/Buddy_memory_allocation

Agenda

- Pointers Wrap-up
- Strings in C
- C Memory Management
- Stack
- Heap
- Implementations of **`malloc/free`**
- **Common memory problems & how to avoid/find them**
- And in Conclusion, …

Common Memory Problems

- Using uninitialized values
- Using memory that you don't own
  - De-allocated stack or heap variable
  - Out-of-bounds reference to array
  - Using **`NULL`** or garbage data as a pointer
- Improper use of **`free/realloc`** by messing with the pointer returned by **`malloc/calloc`**
- Memory leaks
  - you allocated something but forgot to free it later

Assignment to free'd Memory

Code

```
#include <stdlib.h>
#include <stdio.h>

int main(void) {
    int* a = malloc(sizeof(int));
    // ...
    free(a);  // a no longer exists!
    int* b = malloc(sizeof(int));
    *b = 128;
    *a = 55;  // ERROR!
    printf("a=%d,  b=%d (==128!)\n",
           *a, *b);
}
```

Output

```
$ gcc test.c
$ ./a.out
a=55, b=55 (==128!)
```

- Assignment to **a** corrupts **b**
  - or something else happens that is even more undesirable
- Error may go undetected!

“Defensive” Programming

Code

```
#include <stdlib.h>
#include <stdio.h>

int main(void) {
    int* a = malloc(sizeof(int));
    // ...
    free(a);
    a = NULL; // <-- point to illegal loc
    int* b = malloc(sizeof(int));
    *b = 128;
    *a = 55;
}
```

Output

```
$ ./a.out
Segmentation fault: 11
```

- Problem is evident
- But where is the error?
  - May not be obvious in a BIG program

Debugger (gdb)

```
$ gcc -g DefensiveB.c
$ gdb a.out
GNU gdb (GDB) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
Reading symbols from a.out...Reading symbols from
/a.out.dSYM/Contents/Resources/DWARF/a.out...done.

(gdb) run
Starting program: /defensiveB/a.out

Program received signal SIGSEGV, Segmentation fault.
0x0000000100000f76 in main () at DefensiveB.c:11
11              *a = 55;
(gdb)
```

What's wrong with this code?

```c
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    int *a = malloc(10*sizeof(int));
    int *b = a;  // b is an "alias" for a
    b[0] = 1;
    // work with a (or b) ... then increase size
    a = realloc(a, 1000*sizeof(int));
    // yet another array
    int *c = malloc(10*sizeof(int));
    c[0] = 3;
    b[0] = 2;
    printf("a[0]=%d, b[0]=%d, c[0]=%d\n", a[0], b[0], c[0]);

    printf("a = %p\n", a);
    printf("b = %p\n", b);
    printf("c = %p\n", c);
}
```

Warning:

- This particular result (with **`realloc`**) is a coincidence. If run again, or on a different computer, the result may differ.
- After realloc, **`b`** is no longer valid and points to memory it does not own.
- Using b after calling **`realloc`** is a BUG. Do not program like this!

Output:

```
a[0]=1, b[0]=2, c[0]=2
a = 0x7fc53b802600
b = 0x7fc53b403140
c = 0x7fc53b403140
```

Output: realloc commented out

```
a[0]=2, b[0]=2, c[0]=3
a = 0x7f9bdbc04be0
b = 0x7f9bdbc04be0
c = 0x7f9bdbc04c10
```

How many errors in this code?

```
#include <stdio.h>

char *upcase(char *str) {
    char dst;
    char *dp = &dst;
    while (*str) {
        char c = *str++;
        if (c >= 'a' && c <= 'z') c += 'A' - 'a';
        *dp-- = c;
    }
    printf("dp = %s\n", dp);
    return dp;
}

int main(void) {
    char str[] = "Nice weather today!";
    printf("str= %s\n", str);
    printf("up = %s\n", upcase(str));
}
```

Upcase Fixed

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

char *upcase(char *str) {
    char *res = malloc(strlen(str));  // free?
    char *dp = res;
    while (*str) {
        char c = *str++;
        if (c >= 'a' && c <= 'z') c += 'A' - 'a';
        *dp++ = c;
    }
    *dp = 0;
    return res;
}

int main(void) {
    char str[] = "Nice weather today!";
    printf("str= %s\n", str);
    printf("up = %s\n", upcase(str));
}
```

Still need strategy for freeing memory. Caller is responsible …

Output: str= Nice weather today!
up = NICE WEATHER TODAY!

What's wrong with this Code?

```
typedef struct node {
   struct node* next;
   int val;
} Node;

int findLastNodeValue(Node* head) {
   while (head->next != NULL) {
         head = head->next;
   }
   return head->val;
}
```

Administrivia

- Last call for instructional accounts: today at midnight
  - Fill out the Inst Acct Google Form on Piazza
- Homework 0 due this Friday
- Mini-Bio due before discussion section
  - You may choose to go to any discussion section but remember that sticking with 1 TA is better for EPA
- Weekly tutoring/Guerrilla sessions will begin next week!

Break!

Information

About
News
Tool Suite
Supported Platforms
The Developers

Source Code

Current Releases
Release Archive
Variants / Patches
Code Repository
Valkyrie / GUIs

Documentation

Table of Contents
Quick Start
FAQ
User Manual
Download Manual
Research Papers
Books

Valgrind is an instrumentation framework for building dynamic analysis tools. There are Valgrind tools that can automatically detect many memory management and threading bugs, and profile your programs in detail. You can

http://valgrind.org

Valgrind Example

```
1   #include <stdlib.h>
2
3   void f(void) {
4       int* x = malloc(10 * sizeof(int));
5       x[10] = 0;      // problem 1: heap block overrun
6   }                   // problem 2: memory leak -- x not freed
7
8   int main(void) {
9       f();
10      return 0;
11  }
```

Valgrind Output (abbreviated)

```
$ gcc -o test -g -O0 test.c
$ valgrind --leak-check=yes ./test
==8724== Memcheck, a memory error detector
==8724==
==8724== Invalid write of size 4
==8724==    at 0x100000F5C: f (test.c:5)
==8724==    by 0x100000F83: main (test.c:9)
==8724==
==8724== HEAP SUMMARY:
==8724==
==8724== 40 bytes in 1 blocks are definitely lost …
==8724==    at 0x100008EBB: malloc …
==8724==    by 0x100000F53: f (test.c:4)
==8724==    by 0x100000F83: main (test.c:9)
==8724==
==8724== LEAK SUMMARY:
==8724==    definitely lost: 40 bytes in 1 blocks
==8724==    indirectly lost: 0 bytes in 0 blocks
```

Classification of Bugs

Bohrbugs

- Executing faulty code produces error
  - syntax errors
  - algorithmic errors (e.g. sort)
  - dereferencing NULL
- "Easily" reproducible
- Diagnose with standard debugging tools, e.g. gdb

Heisenbugs

- Executing faulty code may not result in error
  - uninitialized variable
  - writing past array boundary
- Difficult to reproduce
- Hard to diagnose with standard tools
- Defensive programming & Valgrind attempt to convert Heisenbugs to Bohrbugs
  - crash occurs during testing, not $&^#!

Disclaimer: classification is controversial. Just do not write buggy programs …

Ref: J. Gray, Why do computers stop and what can be done about them?, Tandem TR 85.7

Agenda

- Pointers Wrap-up
- Strings in C
- C Memory Management
- Stack
- Heap
- Implementations of **`malloc/free`**
- Common memory problems & how to avoid/find them
- **And in Conclusion, …**

And in Conclusion …

- C has three main memory segments to allocate data:
  - Static Data: Variables outside functions (globals)
  - Stack: Variables local to function
  - Heap: Memory allocated explicitly with **`malloc/free`**
- Heap data is an exceptionally fertile ground for bugs
  - memory leaks & corruption
  - send me your best examples (EPA credit? - you paid for them!)
- ***Strategies:***
  - Planning:
    - Who "owns" **`malloc`**'d data?
    - Often more than one "owner" (pointer) to same data
    - Who can safely call **`free`**?
  - Defensive programming, e.g.
    - Assign **`NULL`** to free'd pointer
    - Use **`const`**'s for array size
  - Tools, e.g.
    - gdb, Valgrind

Additional Examples of

C Memory Errors

(to peruse at your leisure)

Using Memory You Don't Own

- What is wrong with this code?

```
int *ipr, *ipw;
void ReadMem() {
      int i, j;
      ipr = (int *) malloc(4 * sizeof(int));
                    i = *(ipr - 1000); j = *(ipr + 1000);
      free(ipr);
   }

   void WriteMem() {
      ipw = (int *) malloc(5 * sizeof(int));
      *(ipw - 1000) = 0; *(ipw + 1000) = 0;
      free(ipw);
   }
```

Using Memory You Don't Own

- Using pointers beyond the range that had been malloc'd
  - May look obvious, but what if mem refs had been result of pointer arithmetic that erroneously took them out of the allocated range?

```
int *ipr, *ipw;
void ReadMem() {
   int i, j;
   ipr = (int *) malloc(4 * sizeof(int));
         i = *(ipr - 1000); j = *(ipr + 1000);
   free(ipr);
 }

 void WriteMem() {
   ipw = (int *) malloc(5 * sizeof(int));
   *(ipw - 1000) = 0; *(ipw + 1000) = 0;
   free(ipw);
 }
```

Faulty Heap Management

- What is wrong with this code?

```
int *pi;
void foo() {
  pi = malloc(8*sizeof(int));
  …
  free(pi);
}

void main() {
  pi = malloc(4*sizeof(int));
  foo();
  …
}
```

Faulty Heap Management

- Memory leak: *more mallocs than frees*

```
int *pi;
void foo() {
  pi = malloc(8*sizeof(int));
  /* Allocate memory for pi */
  /* Oops, leaked the old memory pointed to by pi */
  …
  free(pi);
}

void main() {
  pi = malloc(4*sizeof(int));
  foo(); /* Memory leak: foo leaks it */
   …
}
```

Faulty Heap Management

- What is wrong with this code?

```
int *plk = NULL;
void genPLK() {
  plk = malloc(2 * sizeof(int));
   … … …
  plk++;
}
```

Faulty Heap Management

- Potential memory leak – handle (block pointer) has been changed, do you still have copy of it that can correctly be used in a later free?

```
int *plk = NULL;
void genPLK() {
  plk = malloc(2 * sizeof(int));
  … … …
  plk++;     /* Potential leak: pointer variable
  incremented past beginning of block! */
}
```

Faulty Heap Management

- What is wrong with this code?

```
void FreeMemX() {
  int fnh = 0;
  free(&fnh);
}

void FreeMemY() {
  int *fum = malloc(4 * sizeof(int));
  free(fum+1);
  free(fum);
  free(fum);
}
```

Faulty Heap Management

- Can't free non-heap memory; Can't free memory that hasn't been allocated

```
void FreeMemX() {
  int fnh = 0;
  free(&fnh); /* Oops! freeing stack memory */
}

void FreeMemY() {
  int *fum = malloc(4 * sizeof(int));
  free(fum+1);
  /* fum+1 is not a proper handle; points to middle
  of a block */
  free(fum);
  free(fum);
  /* Oops! Attempt to free already freed memory */
}
```

Using Memory You Haven't Allocated

- What is wrong with this code?

```
void StringManipulate() {
  const char *name = "Safety Critical";
  char *str = malloc(10);
  strncpy(str, name, 10);
  str[10] = '\0';
  printf("%s\n", str);
}
```

Using Memory You Haven't Allocated

- Reference beyond array bounds

```
void StringManipulate() {
  const char *name = "Safety Critical";
  char *str = malloc(10);
  strncpy(str, name, 10);
  str[10] = '\0';
   /* Write Beyond Array Bounds */
  printf("%s\n", str);
   /* Read Beyond Array Bounds */
}
```

Using Memory You Don't Own

- What's wrong with this code?

```
char *append(const char* s1, const char *s2) {
  const int MAXSIZE = 128;
  char result[128];
  int i=0, j=0;
  for (j=0; i<MAXSIZE-1 && j<strlen(s1); i++,j++) {
        result[i] = s1[j];
  }
  for (j=0; i<MAXSIZE-1 && j<strlen(s2); i++,j++) {
        result[i] = s2[j];
  }
  result[++i] = '\0';
  return result;
}
```

Using Memory You Don't Own

- Beyond stack read/write

```
char *append(const char* s1, const char *s2) {
   const int MAXSIZE = 128;
   char result[128];
   int i=0, j=0;
   for (j=0; i<MAXSIZE-1 && j<strlen(s1); i++,j++) {
          result[i] = s1[j];
   }
   for (j=0; i<MAXSIZE-1 && j<strlen(s2); i++,j++) {
          result[i] = s2[j];
   }
   result[++i] = '\0';
   return result;
}
```

`result` is a local array name – stack memory allocated

Function returns pointer to stack memory – won't be valid after function returns

Managing the Heap

- `realloc(p,size)`:
  - Resize a previously allocated block at `p` to a new `size`
  - If `p` is `NULL`, then `realloc` behaves like `malloc`
  - If `size` is 0, then `realloc` behaves like `free`, deallocating the block from the heap
  - Returns new address of the memory block; NOTE: it is likely to have moved!

  E.g.: allocate an array of 10 elements, expand to 20 elements later

```
int *ip;
ip = (int *) malloc(10*sizeof(int));
/* always check for ip == NULL */
… … …
ip = (int *) realloc(ip,20*sizeof(int));
/* always check for ip == NULL */
/* contents of first 10 elements retained */
… … …
realloc(ip,0); /* identical to free(ip) */
```

Using Memory You Don't Own

- What is wrong with this code?

```
int* init_array(int *ptr, int new_size) {
    ptr = realloc(ptr, new_size*sizeof(int));
    memset(ptr, 0, new_size*sizeof(int));
    return ptr;
}

int* fill_fibonacci(int *fib, int size) {
    int i;
    init_array(fib, size);
    /* fib[0] = 0; */ fib[1] = 1;
    for (i=2; i<size; i++)
                fib[i] = fib[i-1] + fib[i-2];
    return fib;
}
```

Using Memory You Don't Own

- Improper matched usage of mem handles

```
int* init_array(int *ptr, int new_size) {
    ptr = realloc(ptr, new_size*sizeof(int));
    memset(ptr, 0, new_size*sizeof(int));
    return ptr;
}

int* fill_fibonacci(int *fib, int size) {
    int i;
    /* oops, forgot: fib = */ init_array(fib, size);
    /* fib[0] = 0; */ fib[1] = 1;
    for (i=2; i<size; i++)
                fib[i] = fib[i-1] + fib[i-2];
    return fib;
}
```

Remember: `realloc` may move entire block

What if array is moved to new location?

Where is my stuff?

```c
#include <stdio.h>
#include <stdlib.h>

int global = 1;

int twice(int i) { return 2*i; }

int main(void) {
    int stack = 2;
    int *heap = malloc(sizeof(int));
    *heap = 3;
    int (*code)(int);  // pointer to "int func(int) {}"
    code = twice;
    printf("global  = %d, &global= %p\n", global, &global);
    printf("stack   = %d, &stack = %p\n", stack, &stack);
    printf("*heap   = %d, heap   = %p\n", *heap, heap);
    printf("code(4) = %d, code   = %p\n", twice(4), twice);
}
```

```
global  = 1, &global= 0x10923f020
stack   = 2, &stack = 0x7fff569c1bec
*heap   = 3, heap   = 0x7fa7b8400020
code(4) = 8, code   = 0x10923ee40
```

Aside: Memory "Leaks" in Java

- Accidentally keeping a reference to an unused object prevents the garbage collector to reclaim it
- May eventually lead to "Out of Memory" error
- But many errors are eliminated:
  - Calling **`free()`** with invalid argument
  - Accessing free'd memory
  - Accessing outside array bounds
  - Accessing unallocated memory (forgot calling **`new`**) (get null-pointer exception – error, but at least no "silent" data corruption)
  - **All this *can* happen in a C program!**

Using the Heap … Example

```c
#include <stdio.h>
#include <stdlib.h>

typedef struct {
    int year_planted;
    // ... kind, cost, ...
} Tree;

Tree* plant_tree(int year) {
    Tree* tn = malloc(sizeof(Tree));
    tn->year_planted = year;
    return tn;
}

int main(void) {
    const int ORCHARD = 100;
    // lets grow some apple trees ...
    Tree* apples[ORCHARD];
    for (int i=0;  i<ORCHARD;  i++)
        apples[i]= plant_tree(2014);

    // apples don't sell ... let's try pears
    Tree* pears[ORCHARD];
    for (int i=0;  i<ORCHARD;  i++)
        pears[i]= plant_tree(2016);
}
```

- New problem:
  - how do we get rid of the apple trees?
- Need a way to free no longer used memory
  - or may eventually run out