More Memory (Mis) Management
Administrivia...

- After this lecture you should be able to do lab2, HW2, and Project 1
  - Do the lab and homework first to get up to speed on C in practice
- Reminder on project 1: It is *subtle* and covers a lot of the C language
Reminder: Remember What We Said Earlier About Buckets of Bits?

- C's memory model is that conceptually there is simply one **yuge** bucket of bits
- Arranged in bytes
- Each byte has an **address**
  - Starting at 0 and going up to the maximum value (0xFFFFFFFF on a 32b architecture)
  - 32b architecture means the # of bits in the address
- We commonly think in terms of "words"
  - Least significant bits of the address are the offset within the word
  - Word size is 32b for a 32b architecture, 64b for a 64b architecture:
    A word is big enough to hold an **address**

<table>
<thead>
<tr>
<th>Address</th>
<th>Mask 1</th>
<th>Mask 2</th>
<th>Mask 3</th>
<th>Mask 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFFFFFFFEC</td>
<td>xxxx</td>
<td>xxxx</td>
<td>xxxx</td>
<td>xxxx</td>
</tr>
<tr>
<td>0xFFFFFFFF8</td>
<td>xxxx</td>
<td>xxxx</td>
<td>xxxx</td>
<td>xxxx</td>
</tr>
<tr>
<td>0xFFFFFFFF4</td>
<td>xxxx</td>
<td>xxxx</td>
<td>xxxx</td>
<td>xxxx</td>
</tr>
<tr>
<td>0xFFFFFFFF0</td>
<td>xxxx</td>
<td>xxxx</td>
<td>xxxx</td>
<td>xxxx</td>
</tr>
<tr>
<td>0xFFFFFFFFEC</td>
<td>xxxx</td>
<td>xxxx</td>
<td>xxxx</td>
<td>xxxx</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>0x14</td>
<td>xxxx</td>
<td>xxxx</td>
<td>xxxx</td>
<td>xxxx</td>
</tr>
<tr>
<td>0x10</td>
<td>xxxx</td>
<td>xxxx</td>
<td>xxxx</td>
<td>xxxx</td>
</tr>
<tr>
<td>0x0C</td>
<td>xxxx</td>
<td>xxxx</td>
<td>xxxx</td>
<td>xxxx</td>
</tr>
<tr>
<td>0x08</td>
<td>xxxx</td>
<td>xxxx</td>
<td>xxxx</td>
<td>xxxx</td>
</tr>
<tr>
<td>0x04</td>
<td>xxxx</td>
<td>xxxx</td>
<td>xxxx</td>
<td>xxxx</td>
</tr>
<tr>
<td>0x00</td>
<td>xxxx</td>
<td>xxxx</td>
<td>xxxx</td>
<td>xxxx</td>
</tr>
</tbody>
</table>
And so for pointers...

- Declaring pointers
  - `int a; /* An integer value */`
  - `int *p; /* A pointer to an integer */`
  - `char **q; /* A pointer to a pointer to a character */`

- Getting the address of a variable/value
  - `p = &a;`

- Getting or setting the value held at a pointer
  - `a = *p;`
  - `*p = a;`

- And pointer arithmetic & arrays:
  - `p[10];`
  - `*(p + 10); /* Since sizeof(int) == 4, the actual address is 40 + p */`
Arguments in `main()`

- To get arguments to the main function, use:
  - `int main(int argc, char *argv[])`

- What does this mean?
  - `argc` contains the number of strings on the command line (the executable counts as one, plus one for each argument). Here `argc` is 2:
    - `unix% sort myFile`
  - `argv` is a pointer to an array containing the arguments as strings
    - Since it is an array of pointers to character arrays
    - Sometimes written as `char **argv`
Example

• `foo hello 87 "bar baz"
• `argc = 4 /* number arguments */
• `argv[0] = "foo",
  `argv[1] = "hello",
  `argv[2] = "87",
  `argv[3] = "bar baz",
• Array of pointers to strings
C Memory Management

• How does the C compiler determine where to put all the variables in machine’s memory?
• How to create dynamically sized objects?
• To simplify discussion, we assume one program runs at a time, with access to all of memory.
• Later, we’ll discuss virtual memory, which lets multiple programs all run at same time, each thinking they own all of memory
  • The only real addition is the C runtime has to say "Hey operating system, gimme a big block of memory" when it needs more memory
C Memory Management

- Program’s address space contains 4 regions:
  - **stack**: local variables inside functions, grows downward
  - **heap**: space requested for dynamic data via `malloc()` resizes dynamically, grows upward
  - **static data**: variables declared outside functions, does not grow or shrink. Loaded when program starts, can be modified.
  - **code**: loaded when program starts, does not change
- 0x0000 0000 hunk is reserved and unwriteable/unreadable so you crash on null pointer access
Where are Variables Allocated?

- If declared outside a function, allocated in “static” storage
- If declared inside function, allocated on the “stack” and freed when function returns
- `main()` is treated like a function

For both of these types of memory, 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!

Big difference from Java

```c
int myGlobal;
main() {
    int myTemp;
}
```
The Stack

- Every time a function is called, a new "stack frame" is allocated on the stack.
- Stack frame includes:
  - Return address (who called me?)
  - Arguments
  - Local variables
- Stack frames use contiguous blocks of memory; stack pointer indicates start of stack frame.
- When function ends, stack pointer moves up; frees memory for future stack frames.
- We’ll cover details later for RISC-V processor.

```c
fooA() { fooB(); }
fooB() { fooC(); }
fooC() { fooD(); }
```
Stack Animation

- Last In, First Out (LIFO) data structure

```c
void a (int m)
{
    b(1);
}
void b (int n)
{
    c(2);
}
void c (int o)
{
    d(3);
}
void d (int p)
{
}
```
Managing the Heap

C supports 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
- Careful – it might move!
  - And it *will not update other pointers pointing to the same block of memory*
Malloc()

- `void *malloc(size_t n):`
  - Allocate a block of uninitialized memory
  - NOTE: Subsequent calls probably will not yield adjacent blocks
  - `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 (check for it!)
  - Additional control information (including size) stored in the heap for each allocated block.
  - Basically the analogy to "new" in Java

- **Examples:**
  - `int *ip;`  
    `ip = (int *) malloc(sizeof(int));`
  - `typedef struct { ... } TreeNode;`  
    `TreeNode *tp = (TreeNode *) malloc(sizeof(TreeNode));`

- `sizeof` returns size of given type in bytes, **necessary if you want portable code!**
And then free()

- **void free(void *p):**
  - p is a pointer containing the address originally returned by malloc()

- Examples:
  - int *ip;
    ip = (int *) malloc(sizeof(int));
    ...
    free((void*) ip); /* Can you free(ip) after ip++ ? */
  - typedef struct {... } TreeNode;
    TreeNode *tp = (TreeNode *) malloc(sizeof(TreeNode));
    ...
    free((void *) tp);

- When you free memory, you must be sure that you pass the original address returned from malloc() to free(); Otherwise, crash (or worse)!
Using Dynamic Memory

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

Node *root = NULL;

Node *create_node(int key, Node *left, Node *right) {
    Node *np;
    if (!np =
        (Node*) malloc(sizeof(Node))){
        printf("Memory exhausted!\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);
    }
    else if (key <= (*tree)->key){
        insert(key, &((*tree)->left));
    }
    else{
        insert(key, &((*tree)->right));
    }
}

int main(){
    insert(10, &root);
    insert(16, &root);
    insert(5, &root);
    insert(11, &root);
    return 0;
}
Observations

- Code, Static storage are easy: they never grow or shrink
- Stack space is relatively easy: stack frames are created and destroyed in last-in, first-out (LIFO) order
- Managing the heap is tricky: memory can be allocated / deallocated at any time
  - If you forget to deallocate memory: “Memory Leak”
    - Your program will eventually run out of memory
  - If you call free twice on the same memory: “Double Free”
    - Possible crash or exploitable vulnerability
  - If you use data after calling free: “Use after free”
    - Possible crash or exploitable vulnerability
Strings...

• Reminder: Strings are just like any other C array...
  • You have a pointer to the start and no way of knowing the length
  • But you have an in-band "end of string" signal with the '\0' (0-byte) character

• Since you can have multiple pointers point to the same thing...
  • `char *a, *b; ... 
    a = b; ... 
    b[4] = 'x'; /* This will update a as well, since they are pointing to the same thing */`

• So how do you copy a string?
  • Find the length (strlen), allocate a new array, and then call strcpy...
  • `a = malloc(sizeof(char) * (strlen(b) + 1) ); 
    /* Forget the +1 at your own peril */`
  • `strcpy(a, b) or strncpy(a, b, strlen(b) + 1);`
  • `strcpy` doesn't know the length of the destination, so it can be very unsafe
  • `strncpy` copies only n character for safety, but if its too short it **will not copy the null terminator!**
And Constant Strings…

- Anything you put explicitly in quotes becomes a `constant` string
  - `char *foo = "this is a constant";`
- For efficiency, these strings are stored as `read only` global variables
  - So if you also have `char *bar = "this is a constant";`
  - it is the same string
- It is, guess what, undefined behavior to write to a constant string
  - But fortunately it is usually an immediate crash.
String & Character Functions

- getc/getchar
  - Read single characters... Note return type!

- gets/fgets
  - Read strings up to a linefeed...
  - Note danger of gets(): it will write however much it wants to!

- printf/fprintf
  - Formatted printing functions

- scanf/fscanf
  - Formatted data input functions: Need to take pointers as argument
  - e.g.
    ```
    int i;
    scanf("%i", &i);
    ```
Pointer Ninjitsu: Pointers to Functions

- You have a function definition
  - `char *foo(char *a, int b){ ... }`

- Can create a pointer of that type...
  - `char *(*f)(char *, int);`
    - Declares f as a function taking a char * and an int and returning a char *

- Can assign to it
  - `f = &foo`
    - Create a reference to function foo

- And can then call it...
  - `printf(“%s\n”, (*f)(“cat”, 3))`

- Necessary if you want to write generic code in C:
  - E.g. a hashtable that can handle pointers of any type
C unions

• We’ve seen how structs can hold multiple elements addressed by name…
  • But what if you want to hold different types in the same location?

• union fubar {
  int a;
  char *b;
  void c;
} Fubar;

• Accessed just like a struct, but…
  • Fubar *f = (Fubar *) malloc(sizeof(union fubar))…
    f->a = 1312;
    f->b = “baz”

• They are actually the same memory! It is just treated differently by the compiler!
How to Use Unions…

• Well, you also have to know what the type is… Because C won't do it for you

• Common pattern

  • enum FieldType {a_type, b_type, c_type};
    union bar {
      char *a;
      int b;
      float c;};

  struct foo {
    FieldType type;
    union bar data; }

  ...

  struct foo *f;

  ...

  switch(f->type){
    case a_type:
      printf("%s\n", f->data.a); break;
Endianness...

• Consider the following

```c
union confuzzle { int a; char b[4]; };  
union confuzzle foo;  
foo.a = 0x12345678;
```

• In a 32b architecture, what would foo.b[0] be?  
  0x12? 0x78?

• It's actually dependent on the architecture's "endianness"
  • Big endian: The first character is the most significant byte: 0x12
  • Little endian: The first character is the least significant byte: 0x78
Endianness and You...

- It generally doesn't matter if you write portable C code running on one computer...
  - After all, you shouldn't be treating an integer as a series of raw bytes
  - Well, it matters when you take CS161: x86 is little endian and you may write an address as a string
- It does matter when you want to communicate across computers...
  - The "network byte order" is big-endian, but your computer may be little-endian
- Endian conversion functions:
  - `ntohs()`, `htons()`: Convert 16 bit values from your native architecture to network byte order and vice versa
  - `ntohl()`, `htonl()`: Convert 32 bit values from your native architecture to network byte order and vice versa
Structure Layout In Memory

• Everything in C is just buckets of bytes…
  • So how do we do structures? We lay out the structure starting at the 0th byte

• `struct foo {
    int a;
    char b;
    short c;
    char *d;
};`

• It depends on the compiler and underlying architecture…
Alignment, Packing, & Structures...

• If the architecture did not not force alignment:
  • Just squish everything together (Sometimes seen on old exams)
  • struct foo {
      int a;    /* At 0 */
      char b;   /* At 4 */
      short c;  /* At 5 */
      char *d;  /* At 7 */
      char e;}; /* At 11 */

• But we already mention that computers don’t actually like this!
  • They want things aligned
Default Alignment Rules…

• These are the **default** alignment rules for the class
  • Centered around a “32b architecture”: Integers and pointers are 32b values
  • char: 1 byte, no alignment needed when stored in memory
  • short: 2 bytes, 1/2 world aligned
    • So 0, 2, 4, 6…
  • int: 4 bytes, word aligned
  • pointers are the same size as ints
  • Need to allow multiple instances of the same structure to be aligned!
So with alignment

- `struct foo {
    int a;    /* At 0 */
    char b;   /* At 4 */
    short c;  /* At 6 */
    char *d;  /* At 8 */
    char e;   /* At 13 */
};`

- For the class we assume *no reordering of fields*
- But `sizeof(struct foo) == 16!`
- Need to add padding to the end as well, so that if we allocate two structures at the same time it is always aligned!
Pointer Ninjitsu: Pointers to arrays of structures

- `typedef struct foo_struct {
    int x;
    char *z;
    char y;)

- So how big is a foo?
  - assume an aligned architecture, `sizeof(int) == sizeof(void *) == 4`:
  - 12... It needs to be padded

- Dynamically allocated a single element:
  - `foo *f = (foo *) malloc(sizeof(foo))`

- Dynamically allocate a 10 entry array of foos:
  - `foo *f = (foo *) malloc(sizeof(foo) * 10);`
Pointer Ninjitsu Continued: Accessing that array...

- Accessing the 5th element's string pointer:
  - `f[4].z = "fubar";`  
  - `(f + 4)->z = "fubar"; /* Semantically equivalent but LESS READABLE! */
    - Assigns the `z` pointer to point to the static string `fubar`
    - It is undefined behavior to then do
      - `f[4].z[1] = 'X'`
    - If you want to modify the string pointed to by `z` you are going to have to do a string copy

- What does it look like "under the hood"?
  - The address written to in `f[4].z = "fubar"` is `(f + 4 * 12 + 4)`:
    - Note: This math is the 'under the hood' math: if you actually tried this in C it would not work right! But it is what the compiler produces in the assembly language
    - The 5th element of type `foo` is offset `(4*12)` from `f`
      - Since we want all elements in the array to have the same alignment this is why we had the padding
      - The field `z` is offset 4 from the start of a `foo` object
Pointer Ninjitsu Advanced: How C++ works...

- C++ is "Object Oriented C"
  - AKA "portable PDP8 assembly language with delusions of grandeur"
- C++ objects are C structures with an extra pointer at the beginning
  - The "vtable" pointer:
    Pointing to an array of pointers to functions
- For inherited ("virtual") functions...
  - To call that function, the compiler writes code that follows the vtable, gets the pointer to function, and calls that
Managing the Heap

- Recall that C supports 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
    - careful – it might move!
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)
- This is one reason why your C code, when compiled, is dependent on a particular operating system
- C standard malloc library creates data structure inside unused portions to track free space
- This class is about how computers work: How they allocate memory is a huge component
Simple Slow Malloc Implementation

Initial Empty Heap space from Operating System

Malloc library creates linked list of empty blocks (one block initially)

Object 1

First allocation chews up space from start of free space

After many mallocs and frees, have potentially long linked list of odd-sized blocks
Frees link block back onto linked list – might merge with neighboring free space
The Problem Here: Fragmentation

- That memory hierarchy we saw earlier likes things small...
  - And likes things contiguous
- Things start to work badly when stuff is scattered all over the place
  - Which will eventually happen with such a simple allocator
Faster malloc implementations

- Keep separate pools of blocks for different sized objects
- “Buddy allocators” always round up to power-of-2 sized chunks to simplify finding correct size and merging neighboring blocks:
  - Then can just use a simple bitmap to know what is free or occupied
# Power-of-2 “Buddy Allocator”

<table>
<thead>
<tr>
<th>Step</th>
<th>64K</th>
<th>64K</th>
<th>64K</th>
<th>64K</th>
<th>64K</th>
<th>64K</th>
<th>64K</th>
<th>64K</th>
<th>64K</th>
<th>64K</th>
<th>64K</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(2^4)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.1</td>
<td>(2^3)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(2^3)</td>
</tr>
<tr>
<td>2.2</td>
<td>(2^2)</td>
<td>(2^2)</td>
<td>(2^2)</td>
<td>(2^3)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.3</td>
<td>(2^1)</td>
<td>(2^1)</td>
<td>(2^2)</td>
<td>(2^3)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.4</td>
<td>(2^0)</td>
<td>(2^0)</td>
<td>(2^1)</td>
<td>(2^2)</td>
<td>(2^3)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.5</td>
<td>A: (2^0)</td>
<td>(2^0)</td>
<td>(2^1)</td>
<td>(2^2)</td>
<td>(2^3)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>A: (2^0)</td>
<td>(2^0)</td>
<td>B: (2^1)</td>
<td>(2^2)</td>
<td>(2^3)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>A: (2^0)</td>
<td>C: (2^0)</td>
<td>B: (2^1)</td>
<td>(2^2)</td>
<td>(2^3)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5.1</td>
<td>A: (2^0)</td>
<td>C: (2^0)</td>
<td>B: (2^1)</td>
<td>(2^1)</td>
<td>(2^2)</td>
<td>(2^3)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5.2</td>
<td>A: (2^0)</td>
<td>C: (2^0)</td>
<td>B: (2^1)</td>
<td>D: (2^1)</td>
<td>(2^1)</td>
<td>(2^3)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>A: (2^0)</td>
<td>C: (2^0)</td>
<td>B: (2^1)</td>
<td>D: (2^1)</td>
<td>(2^1)</td>
<td>(2^3)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7.1</td>
<td>A: (2^0)</td>
<td>C: (2^0)</td>
<td>B: (2^1)</td>
<td>(2^1)</td>
<td>(2^3)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7.2</td>
<td>A: (2^0)</td>
<td>C: (2^0)</td>
<td>B: (2^1)</td>
<td>(2^3)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>(2^0)</td>
<td>(2^0)</td>
<td>(2^1)</td>
<td>(2^3)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9.1</td>
<td>(2^0)</td>
<td>(2^0)</td>
<td>(2^1)</td>
<td>(2^2)</td>
<td>(2^3)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9.2</td>
<td>(2^1)</td>
<td>(2^1)</td>
<td>(2^2)</td>
<td>(2^3)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9.3</td>
<td>(2^2)</td>
<td>(2^2)</td>
<td>(2^3)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9.4</td>
<td>(2^3)</td>
<td>(2^3)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9.5</td>
<td>(2^4)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Malloc Implementations

• All provide the same library interface, but can have radically different implementations

• Uses headers at start of allocated blocks and/or space in unallocated memory to hold malloc’s internal data structures

• Rely on programmer remembering to free with same pointer returned by malloc

  • Alternative is a "conservative garbage collector"

  • Rely on programmer not messing with internal data structures accidentally!

• If you get a crash in malloc, it means that somewhere else you wrote off the end of an array
Conservative Mark/Sweep Garbage Collectors

- An alternative to malloc & free...
  - malloc works normally, but free just does nothing
- Instead, it starts with the stack & global variables as the "live" memory
  - But it doesn't know if those variables are pointers, integers, or whatevers...
  - So assume that every piece of memory in the starting set is a pointer...
    - If it points to something that was allocated by malloc, that entire allocation is now considered live, and "mark it" as live
    - Iterate until there is no more newly discovered live memory
  - Now any block of memory that isn't can be deallocated ("sweep")
The Problems: Fragmentation & Pauses...

- A conservative garbage collector can't move memory around
  - So it gets increasingly fragmented...
    When we get to both caches and virtual memory we will see how this causes problems

- A conservative collector needs to stop the program!
  - What would happen if things changed underneath it? Ruh Roh...
  - So the system needs to pause

- Java, Go, and Python don't have this problem
  - Java and Go are designed to understand garbage collection: Able to have incremental collectors that don't require a long halt but only short halts
  - Python doesn't do real garbage collection: Just uses "reference counting". Every python object has a counter for the number of pointers pointing to it. When it gets to 0, free the object
    - Reference counter can't free cycles
Common Memory Problems: aka Common "Anti-patterns"

• Using uninitialized values
  • Especially bad to use uninitialized pointers

• Using memory that you don’t own
  • Deallocated stack or heap variable
  • Out-of-bounds reference to stack or heap array
  • Using NULL or garbage data as a pointer
  • Writing to static strings

• Improper use of free/realloc by messing with the pointer handle returned by malloc/calloc

• Memory leaks (you allocated something you forgot to later free)

• Valgrind is designed to catch most of these
  • It runs the program extra-super-duper-slow in order to add checks for these problems that C doesn't otherwise do
Using Memory You Don’t Own

• What is wrong with this code?

```c
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);
}
```

Out of bounds reads

Out of bounds writes
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();
      ...
  }

The first `malloc` of `pi` leaks
Reflection on Memory Leaks

• Memory leaks are not a problem *if your program terminates quickly*
  • Memory leaks become a much bigger problem when your program keeps running
  • Or when you are running on a small embedded system

• Three solutions:
  • Be very diligent about making sure you `free` all memory
    • Use a tool that helps you find leaked memory
    • Perhaps implement your own reference counter
  • Use a "Conservative Garbage Collector" `malloc`
  • Just quit and restart your program a lot ("burn down the frat-house")
    • Design your server to crash!
    But memory leaks will *slow down your program* long before it actually crashes
So Why Do Memory Leaks Slow Things Down?

• Remember at the start we saw that pyramid of memory?
  • Small & fast -> cache
    Big & slow -> main memory

• Memory leaks lead to *fragmentation*
  • As a consequence you use more memory, and it's more scattered around

• Computers are designed to access *contiguous* memory
  • So things that cause your working memory to be spread out more and in smaller pieces slow things down

• There also may be nonlinearities:
  • Fine... Fine... Fine... Hit-A-Brick-Wall!
Memory Leaks & The Project...

• We have a test which **will** cause your program to crash if you leak in `processInput()`
  • How do we do this? We tell the OS to not give your program very much memory...
  • But we won't check for leaks in your dictionary/hashtable
  • After all, you have to have it in memory for the entire program lifetime
  • So keep that in mind when running valgrind...
    • "Leaked memory" allocated in `readDictionary()` 😵
    • "Leaked memory" allocated in `processInput()` 😞
Faulty Heap Management

• What is wrong with this code?

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

This MAY be a memory leak if we don't keep somewhere else a copy of the original malloc'ed pointer
Faulty Heap Management

• How many things are wrong with this code?

  - void FreeMemX() {
    int fnh[3] = 0;
    ...
    free(fnh);  \textcolor{red}{Can't \texttt{free} memory allocated on the stack}
  }

  - void FreeMemY() {
    int *fum = malloc(4 * sizeof(int));
    free(fum+1);  \textcolor{red}{Can't \texttt{free} memory that isn't the pointer from \texttt{malloc}}
    ...
    free(fum);
    ...
    free(fum);  \textcolor{red}{Can't \texttt{free} memory twice}
  }
Using Memory You Haven’t Allocated

• What is wrong with this code?

```c
void StringManipulate() {
    const char *name = "Safety Critical"; // sizeof(char) is 1 but should have sizeof as a good habit
    char *str = malloc(10);
    strncpy(str, name, 10);
    str[10] = '\0'; // Write off of the end of the array!
    printf("%s\n", str);
}
```
Using Memory You Don’t Own

• What’s wrong with this code?

```c
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;
}
```

Returning a pointer to stack-allocated memory!
Using Memory You Don’t Own

• What is 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;
}

What if head is null?
Always check arguments.
Your code may be good...
But you make mistakes!
PROGRAM DEFENSIVELY
Using Memory You Don’t Own

- What is wrong with this code?

```c
void muckString(char *str) {
    str[0] = 'b';
}
void main(void) {
    char *str = "abc";
    muckString(str);
    puts(str);
}
```

Pointing to a static string... Ruh Roh...
So Why Was That A Problem...

• When the compiler sees
  • `char *foo = "abc"
  • The compiler interprets it as 'have the constant string "abc" somewhere in static memory, and have foo point to this'
    • If you have the same string "abc" elsewhere, it will point to the same thing...
    • If you are lucky, the compiler makes sure that these string constants are set so you can't write
      • "Access violation", "bus error", "segfault"

• There is something safe however...
  • `char foo[] = "abc"
  • The compiler interprets this as 'create a 4 character array on the stack, and initialize it to "abc"
    • But of course we can't now say `return foo;`
    • Because that would be returning a pointer to something on the stack...
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!

```c
int *ip;
ip = (int *) malloc(10*sizeof(int)); /* always check for ip == NULL */
...
...
ip = (int *) realloc(ip,20*sizeof(int)); /* always check 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?

```c
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;
    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;
}
```

 realloc might move the block!

Which means this hasn't updated *fib!
And Now A Bit of Security: Overflow Attacks

- struct UnitedFlyer{
  ...
  char lastname[16];
  char status[32];
  /* C will almost certainly lay this out in memory so they are adjacent */
  ...
};

... void updateLastname(char *name, struct UnitedFlyer *f){
  strcpy(f->lastname, name);
}
So what...

- Well, United has my status as:
  - `name = "Weaver", status = "normal-person: hated"`
- So what I need to do is get United to update my name!!!
  - So I provide United with my new name as:
    - `name = "Weaver          super-elite: actually like"`
    - `status = "super-elite: actually like"`
  - And then update my name **again** back to just "Weaver"
    - `name = "Weaver", status = "super-elite: actually like"`
- Basic premise of a **buffer overflow** attack:
  - An input that overwrites past the end of the buffer and leaves the resulting memory in a state suitable to the attacker's goals
And Valgrind...

- 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
- You **must** run your program in Valgrind before you ask for debugging help from a TA!
  - Tools like Valgrind are absolutely essential for debugging C code