Pointers, Arrays, Memory: AKA the cause of those F@#)(#@*(@ Segfaults
Agenda

- Pointers Redux
  - This is subtle and important, so going over again
- Arrays in C
- Memory Allocation
Address vs. Value

- Consider memory to be a **single** huge array
- Each cell of the array has an address associated with it
- Each cell also stores some value
- For addresses do we use signed or unsigned numbers? Negative address?!
- Don’t confuse the address referring to a memory location with the value stored there
Pointers

- An *address* refers to a particular memory location; e.g., it points to a memory location.

- *Pointer*: A variable that contains the address of a variable.
Types of Pointers

- Pointers are used to point to any kind of data (`int`, `char`, a `struct`, etc.)
- Normally a pointer only points to one type (`int`, `char`, a `struct`, etc.).
  - `void *` is a type that can point to anything (generic pointer)
  - Use `void *` sparingly to help avoid program bugs, and security issues, and other bad things!
- You can even have pointers to functions...
  - `int (*fn) (void *, void *) = &foo`
    - `fn` is a function that accepts two `void *` pointers and returns an `int`
    - is initially pointing to the function `foo`.
    - `(*fn) (x, y)` will then call the function
More C Pointer Dangers

- Declaring a pointer just allocates space to hold the pointer – it does not allocate the thing being pointed to!
- Local variables in C are not initialized, they may contain anything (aka “garbage”)
- What does the following code do?

```c
void f()
{
    int *ptr;
    *ptr = 5;
}
```
NULL pointers...

- The pointer of all 0s is special
  - The "NULL" pointer, like in Java, python, etc...
- If you write to or read a null pointer, your program should crash
- Since "0 is false", its very easy to do tests for null:
  - if(!p) { /* P is a null pointer */ }
  - if(q) { /* Q is not a null pointer */}
Pointers and Structures

typedef struct {
    int x;
    int y;
} Point;

Point p1;
Point p2;
Point *paddr;

/* dot notation */
int h = p1.x;
p2.y = p1.y;

/* arrow notation */
int h = paddr->x;
int h = (*paddr).x;

/* This works too */
p1 = p2;
Pointers in C

• Why use pointers?
  • If we want to pass a large struct or array, it’s easier / faster / etc. to pass a pointer than the whole thing
    • Otherwise we’d need to copy a huge amount of data
  • In general, pointers allow cleaner, more compact code

• So what are the drawbacks?
  • Pointers are probably the single largest source of bugs in C, so be careful anytime you deal with them
    • Most problematic with dynamic memory management—coming up next week
    • Dangling references and memory leaks
Why Pointers in C?

• At time C was invented (early 1970s), compilers often didn’t produce efficient code
  • Computers 100,000x times faster today, compilers better
• C designed to let programmer say what they want code to do without compiler getting in way
  • Even give compilers hints which registers to use!
• Today’s compilers produce much better code, so may not need to use pointers in application code
  • Low-level system code still needs low-level access via pointers
Pointing to Different Size Objects

- Modern machines are “byte-addressable”
  - Hardware’s memory composed of 8-bit storage cells, each has a unique address
- A C pointer is just abstracted memory address
- Type declaration tells compiler how many bytes to fetch on each access through pointer
  - E.g., 32-bit integer stored in 4 consecutive 8-bit bytes
- But we actually want “word alignment”
  - Some processors will not allow you to address 32b values without being on 4 byte boundaries
  - Others will just be very slow if you try to access “unaligned” memory.

![Diagram showing byte address and storage sizes for different types: 16-bit short stored in two bytes, 32-bit integer stored in four bytes, 8-bit character stored in one byte.](Image)
sizeof() operator

- **sizeof(type)** returns number of bytes in object
- But number of bits in a byte is not standardized
  - In olden times, when dragons roamed the earth, bytes could be 5, 6, 7, 9 bits long
  - Includes any padding needed for alignment
- By Standard C99 definition, `sizeof(char) == 1`
- Can take `sizeof(arg)`, or `sizeof(structtype)`
- We’ll see more of sizeof when we look at dynamic memory management
Pointer Arithmetic

`pointer + number`    `pointer – number`
e.g., `pointer + 1` adds 1 **something** to a pointer

char *p;
char a;
char b;

p = &a;
p += 1;

In each, p now points to b
(Assuming compiler doesn’t reorder variables in memory.)

`int *p;`
`int a;`
`int b;`
p = &a;
p += 1;

Never code like this!!!!

Adds 1*sizeof(char) to the memory address

Adds 1*sizeof(int) to the memory address

**Pointer arithmetic should be used cautiously**
Changing a Pointer Argument?

• What if want function to change a pointer?
• What gets printed?

```c
void inc_ptr(int *p)
{
    p = p + 1;
}

int A[3] = {50, 60, 70};
int* q = A;
inc_ptr(q);
printf("*q = %d\n", *q);
```
Pointer to a Pointer

• Solution! Pass a pointer to a pointer, declared as **h
• Now what gets printed?

```
void inc_ptr(int **h)
{
    *h = *h + 1;
}

int A[3] = {50, 60, 70};
int* q = A;
inc_ptr(&q);
printf("*q = %d\n", *q);
```

```prog
void inc_ptr(int **h)
{
    *h = *h + 1;
}

int A[3] = {50, 60, 70};
int* q = A;
inc_ptr(&q);
printf("*q = %d\n", *q);
```
Conclusion on Pointers...

- All data is in memory
  - Each memory location has an address to use to refer to it and a value stored in it
- Pointer is a C version (abstraction) of a data address
  - * “follows” a pointer to its value
  - & gets the address of a value
- C is an efficient language, but leaves safety to the programmer
  - Variables not automatically initialized
  - Use pointers with care: they are a common source of bugs in programs
Structures Revisited

- A "struct" is really just an instruction to C on how to arrange a bunch of bytes in a bucket...
- `struct foo {
    int a;
    char b;
    struct foo *c;
}
- Provides enough space and **aligns** the data with padding
  So actual layout on a 32b architecture will be:
  - 4-bytes for A
  - 1 byte for b
  - 3 unused bytes
  - 4 bytes for C
  - `sizeof(struct foo) == 12`
Plus also Unions

• A "union" is also instruction to C on how to arrange a bunch of bytes

• union foo {
    int a;
    char b;
    union foo *c;
}

• Provides enough space for the **largest element**

• union foo f;
f.a = 0xDEADB33F; /* treat f as an integer and store that value */
f.c = &f; /* treat f as a pointer of type "union foo *" and store the address of f in itself */
Administrivia:

• Project 1 is now live...
  • Yes, we are throwing you in the deep end right away
  • Designed to touch on a huge amount of C concepts
• As is homework 2
• And Lab 1 should be out RSN...
• Register your iClicker on Bcourses...
Clickety Click!

```c
void foo(int *x, int *y)
{
    int t;
    if ( *x > *y ) { t = *y; *y = *x; *x = t; }
}
int a=3, b=2, c=1;
foo(&a, &b);
foo(&b, &c);
foo(&a, &b);
printf("a=%d b=%d c=%d\n", a, b, c);
```

Result is:
A: a=3  b=2  c=1  
B: a=1  b=2  c=3  
C: a=1  b=3  c=2  
D: a=3  b=3  c=3  
E: a=1  b=1  c=1
C Arrays

- Declaration:
  ```c
  int ar[2];
  ```
  declares a 2-element integer array: just a block of memory which is uninitialized. The number of elements is static in the declaration, you can't do "int ar[x]" where x is a variable

  ```c
  int ar[] = {795, 635};
  ```
  declares and initializes a 2-element integer array
Array Name / Pointer Duality

• **Key Concept**: Array variable is simply a “pointer” to the first (0th) element

• So, array variables are *almost* identical to pointers
  • `char *string` and `char string[]` are nearly identical declarations
    • Differ in subtle ways: incrementing & declaration of filled arrays

• Consequences:
  • `ar` is an array variable, but works like a pointer
  • `ar[0]` is the same as `*ar`
  • `ar[2]` is the same as `*(ar+2)`
  • Can use pointer arithmetic to access arrays
Arrays and Pointers

- Array ≈ pointer to the initial element
  - `a[i] ≡ *(a+i)`

- An array is passed to a function as a pointer
  - The array size is **lost**!

- Usually bad style to interchange arrays and pointers
  - Avoid pointer arithmetic!
    - Especially avoid things like `ar++`;

Passing arrays:

```c
int foo(int array[], unsigned int size)
{
    ... array[size - 1] ...
}

int main(void)
{
    int a[10], b[5];
    ... foo(a, 10)... foo(b, 5) ...
}
```

Must explicitly pass the size.
C Arrays are Very Primitive

- An array in C does not know its own length, **and its bounds are not checked!**
  - Consequence: We can accidentally **access off the end of an array**
  - Consequence: We must pass the array **and its size** to any procedure that is going to manipulate it

- **Segmentation faults and bus errors:**
  - These are VERY difficult to find; be careful! (You’ll learn how to debug these in lab)
  - But also “fun” to exploit:
    - “Stack overflow exploit”, maliciously write off the end of an array on the stack
    - “Heap overflow exploit”, maliciously write off the end of an array on the heap
C Strings

• String in C is just an array of characters
  
  ```c
  char string[] = "abc";
  ```

• How do you tell how long a string is?
  
  • Last character is followed by a 0 byte (aka “null terminator”):
    written as 0 (the number) or '\0' as a character

• Important danger: string length operation does not include the null terminator when you ask for length of a string!

```c
int strlen(char s[]) {
    int n = 0;
    while (s[n] != 0){
        n++;
    }
    return n;
}
```
Use Defined Constants

- **Array size** \( n \); want to access from 0 to \( n-1 \), so you should use counter AND utilize a variable for declaration & incrementation
  - **Bad pattern**
    ```c
    int i, ar[10];
    for(i = 0; i < 10; i++) { ... }
    ```
  - **Better pattern**
    ```c
    const int ARRAY_SIZE = 10;
    int i, a[ARRAY_SIZE];
    for(i = 0; i < ARRAY_SIZE; i++) { ... }
    ```

- **SINGLE SOURCE OF TRUTH**
  - You’re utilizing indirection and avoiding maintaining two copies of the number 10
  - DRY: “Don’t Repeat Yourself”
  - And don’t forget the \(<\) rather than \(\leq\): When Nick took 60c, he lost a day to a “segfault in a malloc called by printf on large inputs”: Had a \(\leq\) rather than a \(<\) in a single array initialization!
int
foo(int array[],
    unsigned int size)
{
    ...
    printf("%d\n", sizeof(array));
}

int
main(void)
{
    int a[10], b[5];
    ... foo(a, 10) ... foo(b, 5) ...
    printf("%d\n", sizeof(a));
}
Arrays and Pointers

These code sequences have the same effect!

But the former is much more readable: Especially don't want to see code like ar++
Arrays And Structures And Pointers

• typedef struct bar {
    char *a;    /* A pointer to a character */
    char b[18]; /* A statically sized array
    of characters */
} Bar;

...  
Bar *b = (Bar*) malloc(sizeof(struct bar));
b->a = malloc(sizeof(char) * 24);

• Will require 24 bytes on a 32b architecture for the structure:
  • 4 bytes for a (its a pointer)
  • 18 bytes for b (it is 18 characters)
  • 2 bytes padding (needed to align things)
Some Code Examples

- \( b->b[5] = 'd' \)
  - Location written to is 10th byte pointed to by \( b \)...
    \[ *((\text{char}*) b + 4 + 5) = 'd' \]

- \( b->a[5] = 'c' \)
  - Location written to is the first word pointed to by \( b \), treat that as a pointer, add 5, and write 'c' there...
    aka \[ *(*(\text{char}**b) + 5) = 'c' \]

- \( b->a = b->b \)
  - Location written to is the first word pointed to by \( b \)
  - Value it is set to is \( b \)'s address + 4)...
    aka \[ *((\text{char}**)b) = ((\text{char}*) b) + 4 \]
When Arrays Go Bad: Heartbleed

- In TLS encryption, messages have a length...
  - And get copied into memory before being processed
- One message was “Echo Me back the following data, its this long...”
  - But the (different) echo length wasn’t checked to make sure it wasn’t too big...

```
M 5 HB L=5000 107:0u17;GET / HTTP/1.1\nHost: www.mydomain.com\nCookie: login=1
17kf9012oeu\nUser-Agent: Mozilla....
```

- So you send a small request that says “read back a lot of data”
  - And thus get web requests with auth cookies and other bits of data from random bits of memory...
Clickers!

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

Result is:
A: 2
B: 3
C: 4
D: 5
E: None of the above
```
int x[] = { 2, 4, 6, 8, 10 };
int *p = x;
int **pp = &p;
(*pp)++;
(*(*pp))++;
printf("%d\n", *p);
```

P points to the start of X (2)
PP points to P
Increments P point to 2\textsuperscript{nd} element (4)
Increments 2\textsuperscript{nd} element by 1 (5)

Result is:
A: 2
B: 3
C: 4
D: 5
E: None of the above
Concise strlen()

```c
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[])`

- 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
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: \texttt{0x12}
  • Little endian: The first character is the least significant byte: \texttt{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
- 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
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
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!

```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
  - Space for 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.
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.

- **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
When Memory Goes Bad...
Failure To Free

- **#1: Failure to free allocated memory**
  - "memory leak"

- **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 good just up until the moment it isn't...
    - There are actually a couple of cliffs that will hit

- **And then your program is killed off!**
  - Because the OS goes "Nah, not gonna do it" when you ask for more memory
When Memory Goes Bad: Writing off the end of arrays...

- EG...
  - `int *foo = (int *) malloc(sizeof(int) * 100);`
  - `int i;`
  - `....`
  - `for(i = 0; i <= 100; ++i){`
    - `foo[i] = 0;`
  }

- Corrupts other parts of the program...
  - Including internal C data

- May cause crashes later
When Memory Goes Bad: Returning Pointers into the Stack

• It is OK to pass a pointer to stack space down
  
  • EG:
    ```c
    char [40]foo;
    int bar;
    ...
    strncpy(foo, "102010", strlen("102010")+1);
    baz(&bar);
    ```

• It is catastrophically bad to return a pointer to something in the stack...

  • EG
    ```c
    char [50] foo;
    ....
    return foo;
    ```

• The memory will be overwritten when other functions are called!
  
  • So your data no longer exists... And writes can overwrite key pointers causing crashes!
When Memory Goes Bad: Use After Free

- When you keep using a pointer..
  - `struct foo *f`  
    
    ```c
    ....
    f = malloc(sizeof(struct foo));
    ....
    free(f)
    ....
    bar(f->a);
    ```

- Reads after the free may be corrupted
  - As something else takes over that memory. Your program will probably get wrong info!

- Writes *corrupt* other data!
  - Uh oh... Your program crashes later!
When Memory Goes Bad: Forgetting Realloc Can Move Data...

- When you realloc it can copy data...
  - `struct foo *f = malloc(sizeof(struct foo) * 10);`
  - `...`
  - `struct foo *g = f;`
  - `....`
  - `f = realloc(sizeof(struct foo) * 20);`

- Result is *g may* now point to invalid memory
  - So reads may be corrupted and writes may corrupt other pieces of memory
When Memory Goes Bad: Freeing the Wrong Stuff...

- If you `free()` something never `malloc'ed()
  - Including things like
    ```
    struct foo *f = malloc(sizeof(struct foo) * 10)
    ...
    f++;
    ...
    free(f)
    ```
  - Malloc/free may get confused..
    - Corrupt its internal storage or erase other data...
When Memory Goes Bad: Double-Free...

• EG...
  • struct foo *f = (struct foo *) malloc(sizeof(struct foo) * 10);
    ...
    free(f);
    ...
    free(f);

• May cause either a use after free (because something else called malloc() and got that data) or corrupt malloc's data (because you are no longer freeing a pointer called by malloc)
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
And In Conclusion, ...

- C has three main memory segments in which to allocate data:
  - Static Data: Variables outside functions
  - Stack: Variables local to function
  - Heap: Objects explicitly malloc-ed/free-d.
- Heap data is biggest source of bugs in C code