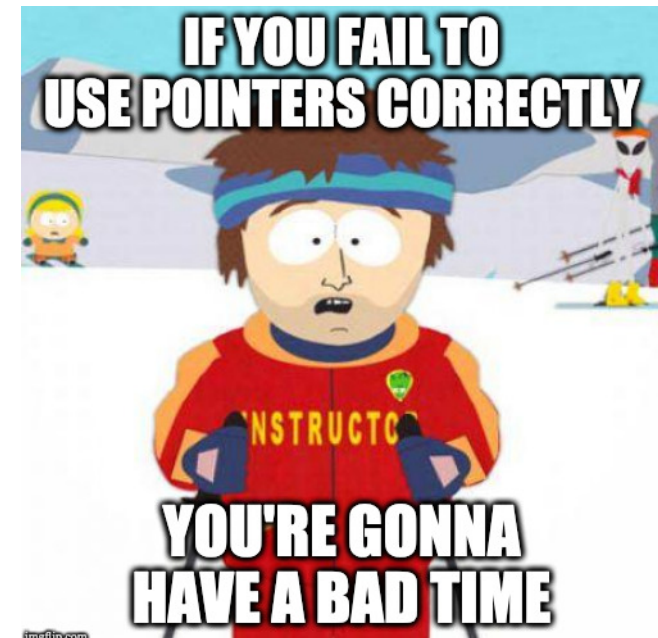


Pointers, Arrays, Memory: AKA the cause of those F@#)(#@*(Segfaults



Announcements!

- Lab 0 due date extended until next Friday
 - But do it this week if you can...
- Next week lecture will still be fully remote
 - As much as we love in-person teaching...
the ZoomCave is a better recording studio than 310 Soda
- Project 1 will be released Real Soon Now (RSN)
 - Start it early:
It covers a lot of tricky language issues
- No lecture & discussion on Monday
 - It is a holiday!

C Syntax: Variable Declarations

- Similar to Java, but with a few minor but important differences
 - All variable declarations must appear before they are used
 - All must be at the beginning of a block.
 - A variable may be initialized in its declaration;
if not, it holds garbage! (the contents are undefined)
- Examples of declarations:
 - Correct: `{ int a = 0, b = 10; ...`
 - Incorrect: `for (int i = 0; i < 10; i++) { ...`

An Important Note: Undefined Behavior...

- A lot of C has “Undefined Behavior”
 - This means it is often un**predictable** behavior
 - It will run one way on one compiler and computer...
 - But some other way on another
 - Or even just be different each time the program is executed!
- Often contributes to **Heisenbugs**
 - Bugs that seem random/hard to reproduce
 - (In contrast to **Bohrbugs** which are deterministic and therefore reproducible)

C Syntax : Control Flow (1/2)

- Within a function, remarkably close to Java constructs (shows Java's legacy) in terms of control flow
 - A statement can be a { } of code or just a standalone statement
- if-else
 - `if (expression) statement`
 - `if (x == 0) y++;`
 - `if (x == 0) {y++;}`
 - `if (x == 0) {y++; j = j + y;}`
 - `if (expression) statement1 else statement2`
 - There is an ambiguity in a series of if/else if/else if you don't use { }s, so **always** use { }s to block the code
 - In fact, it is a bad C habit to not always have the statement in { }s, it has resulted in some amusing errors...
- while
 - `while (expression) statement`
 - `do statement while (expression);`

C Syntax : Control Flow (2/2)

- **for**
 - `for (initialize; check; update) statement`
- **switch**
 - ```
switch (expression) {
 case const1: statements
 case const2: statements
 default: statements
}
```
  - `break; /* need to break out of case */`
  - Note: until you do a break statement things keep executing in the switch statement
- C also has **goto**
  - But it can result in spectacularly bad code if you use it, so don't! Makes your code hard to understand, debug, and modify.

# C Syntax: True or False

- What evaluates to *FALSE* in C?
  - 0 (integer)
  - NULL (a special kind of pointer that is also 0: more on this later)
  - ***No explicit Boolean type in old-school C***
    - Often you see `#define bool (int)`
    - Then `#define false 0`
  - Basically anything where all the bits are 0 is false
- What evaluates to *TRUE* in C?
  - ***Anything*** that isn't false is true
  - Same idea as in Python: only 0s or empty sequences are false, anything else is true!

# C and Java operators nearly identical

- arithmetic: `+`, `-`, `*`, `/`, `%`
- assignment: `=`
- augmented assignment: `+=`, `-=`, `*=`, `/=`, `%=`, `&=`, `|=`, `^=`, `<<=`, `>>=`
- bitwise logic: `~`, `&`, `|`, `^`
- bitwise shifts: `<<`, `>>`
- boolean logic: `!`, `&&`, `||`
- equality testing: `==`, `!=`
- subexpression grouping: `()`
- order relations: `<`, `<=`, `>`, `>=`
- increment and decrement: `++` and `--`
- member selection: `.`, `->`
  - This is slightly different than Java because there are both structures and pointers to structures, more later
- conditional evaluation: `?` `:`



# Our Tip of the Day... Valgrind

- Valgrind turns most unsafe "heisenbugs" into "bohrbugs"
  - It adds almost all the checks that Java does but C does not
  - The result is your program ***immediately*** crashes where you make a mistake
  - It is installed on the lab machines
    - You can install it on some home machines, but not currently supported on MacOS-11
- Nick's scars from his 60C experience:
  - First C project, spent an entire day tracing down a fault...
    - Program would crash ***in a print statement*** only when there was a lot of input
  - That turned out to be a  $\leq$  instead of a  $<$  in initializing an array!

# Remember What We Said Earlier About Buckets of Bits?

- C's memory model is that conceptually there is simply one **huge** 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**

|             |      |      |      |      |
|-------------|------|------|------|------|
| 0xFFFFFFFFC | xxxx | xxxx | xxxx | xxxx |
| 0xFFFFFFFF8 | xxxx | xxxx | xxxx | xxxx |
| 0xFFFFFFFF4 | xxxx | xxxx | xxxx | xxxx |
| 0xFFFFFFFF0 | xxxx | xxxx | xxxx | xxxx |
| 0xFFFFFFFEC | xxxx | xxxx | xxxx | xxxx |
| ...         | ...  | ...  | ...  | ...  |
| 0x14        | xxxx | xxxx | xxxx | xxxx |
| 0x10        | xxxx | xxxx | xxxx | xxxx |
| 0x0C        | xxxx | xxxx | xxxx | xxxx |
| 0x08        | xxxx | xxxx | xxxx | xxxx |
| 0x04        | xxxx | xxxx | xxxx | xxxx |
| 0x00        | xxxx | xxxx | xxxx | xxxx |

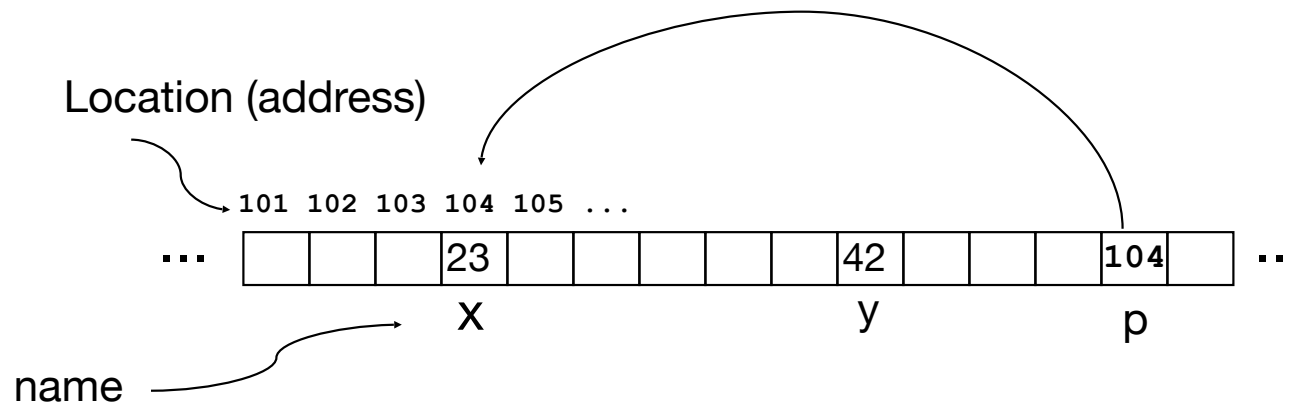
# Address vs. Value

- 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?!
  - Answer: Addresses are **unsigned**
- 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



# Pointer Syntax

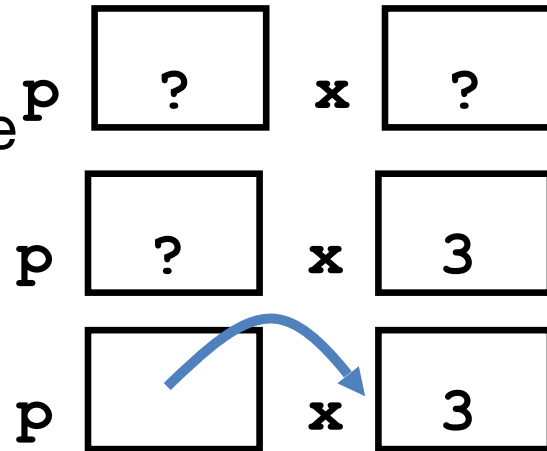
- `int *p;`
  - Tells compiler that `variable p` is address of an `int`
- `p = &y;`
  - Tells compiler to assign `address of y` to `p`
  - `&` called the “address operator” in this context
- `z = *p;`
  - Tells compiler to assign `value at address in p` to `z`
  - `*` called the “dereference operator” in this context

# Creating and Using Pointers

- How to create a pointer:  
& operator: get address of a variable

```
int *p, x; x = 3;
```

```
 p = &x;
```



Note the “\*” gets used two different ways in this example. In the declaration to indicate that `p` is going to be a pointer, and in the `printf` to get the value pointed to by `p`.

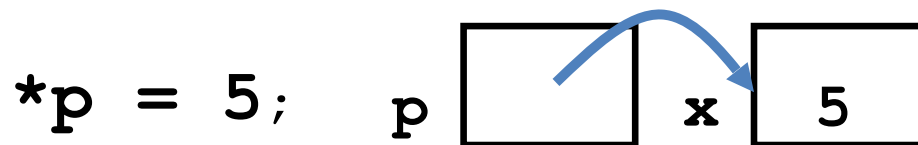
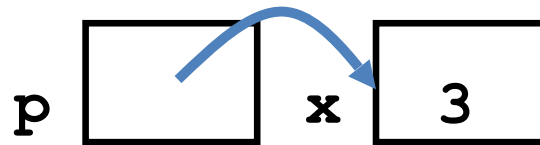
- How get a value pointed to?

“\*” (dereference operator): get the value that the pointer points to

```
printf("p points to %d\n", *p);
```

# Using Pointer for Writes

- How to change a variable pointed to?
  - Use the dereference operator `*` on left of assignment operator `=`



# Pointers and Parameter Passing

- Java and C pass basic parameters “by value”:  
Procedure/function/method gets a copy of the parameter, so *changing the copy cannot change the original*

```
void add_one (int x)
{
 x = x + 1;
}
int y = 3;
add_one(y);
```

*y remains equal to 3*



# Pointers and Parameter Passing

- How can we get a function to change the value held in a variable?

```
void add_one (int *p)
{
 *p = *p + 1;
}
int y = 3;
```

```
add_one(&y);
```

*y is now equal to 4*

# Types of Pointers

- Pointers are used to point to any kind of data (`int`, `char`, a `struct`, a pointer to a pointer to a pointer to a `char`, 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!
    - Can convert types (BUT BE CAREFUL):

```
void *a =
int *p = (int *) a; /* p now points to the same place as a,
 but is treated as a pointer to an int */
int **q = (int **) a; /* q now points to the same place as a,
 but is treated as a pointer to a pointer to an int */
```
- 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` and is initially pointing to the function `foo`.
    - `(*fn) (x, y)` will then call the function

# 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 immediately
  - The memory is set up so that this should never be valid
- 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 */ }`

# 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?

```
void f()
{
 int *ptr;
 *ptr = 5;
}
```

# Pointers and Structures

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

```
Point p1;
Point p2;
Point *paddr;
paddr = &p2;
```

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

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

```
/* This works too:
 copies all of p2 */
p1 = p2;
p1 = *paddr;
```

# Pointers in C

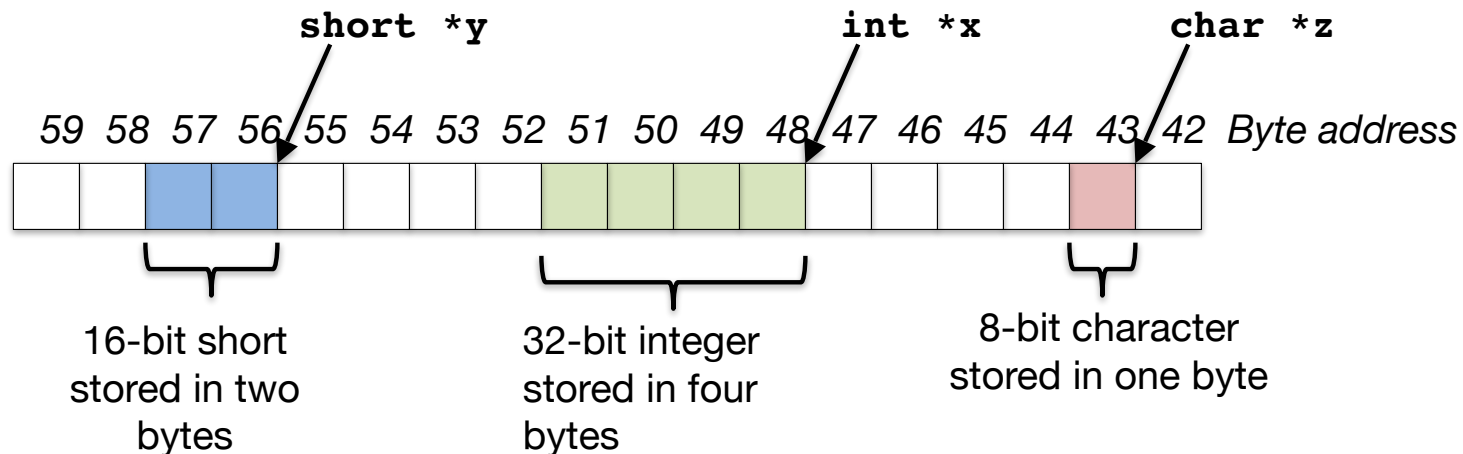
- Why use pointers?
  - If we want to pass a large struct or array, it's easier / faster / etc. to pass a pointer than the whole thing
    - Otherwise we'd need to copy a huge amount of data
    - You notice in Java that more complex objects are passed by reference....  
Under the hood this is a pointer
  - 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 time
    - 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 are *massively* 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 don't need to use raw pointers in application code
  - Most other languages use "pass by reference" for objects, which is semantically similar but with checks for misuse
- Low-level system code still needs low-level access via pointers
  - And compilers basically convert "pass by reference" into pointer-based code

# 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.





# 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

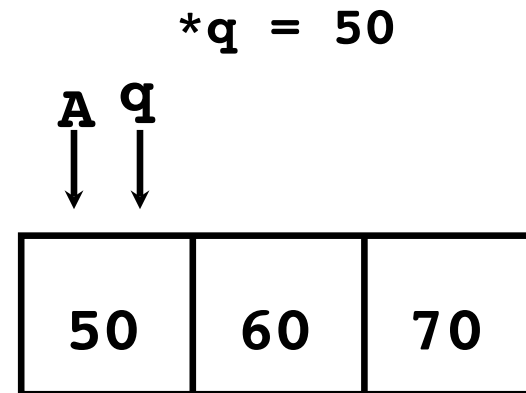
*Pointer arithmetic should be used cautiously*

# Changing a Pointer Argument?

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

```
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);
```

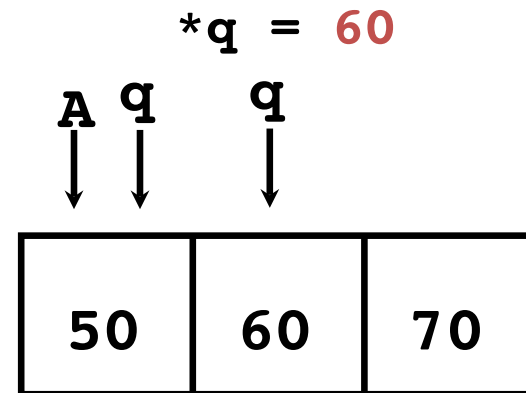


# Pointer to a Pointer

- 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);
```



# 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 */
```

C Arrays

- Declaration:

```
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

```
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[32]` is an array variable with 32 elements, 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 \approx pointer to the initial element
 - $a[i] \equiv *(a+i)$
- 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:

Really `int *array` Must explicitly pass the size

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

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

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
`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!

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

int strlen(char s[])
{
    int n = 0;
    while (*(s++) != 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

```
int i, ar[10];
for(i = 0; i < 10; i++){ ... }
```
 - Better pattern

```
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 $<=$:
When Nick took 60c, he lost a day to a "segfault in a malloc called by printf on large inputs":
Had a $<=$ rather than a $<$ in a single array initialization!

Arrays and Pointers

```
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));
}
```

What does this print? **4**

... because `array` is really a pointer (and a pointer is architecture dependent, but likely to be 4 or 8 on modern 32-64 bit machines!)

What does this print? **40**

Arrays and Pointers

```
int i;
int array[10];

for (i = 0; i < 10; i++)
{
    array[i] = ...;
}
```

```
int *p;
int array[10];

for (p = array; p < &array[10]; p++)
{
    *p = ...;
}
```

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...  
`*((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 `*(*(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 `*((char **)b) = ((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:Ou17;GET / HTTP/1.1\r\n
Host: www.mydomain.com\r\nCookie: login=1
17kf9012oeu\r\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...

# 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[])`
- 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
- ```
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?
- Its 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

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

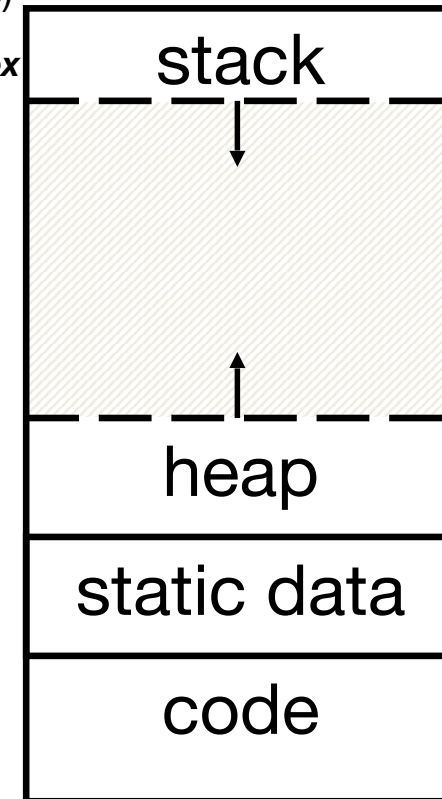
- Program's address space contains (32 bits assumed here)
4 regions:

- **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

Memory Address

$\sim FFFF\ FFFF_{hex}$

$\sim 0000\ 0000_{hex}$



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

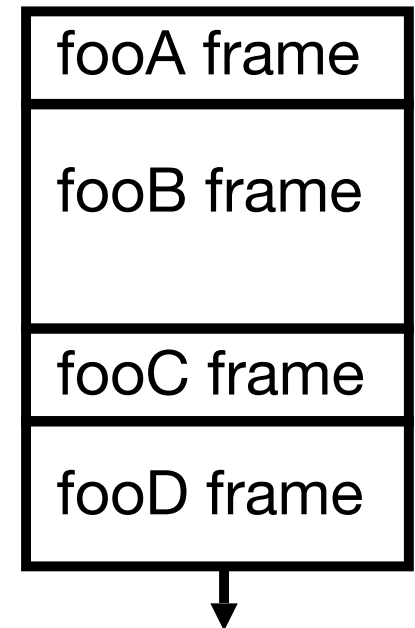
```
int myGlobal;  
main() {  
    int myTemp;  
}
```

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 uses 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

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

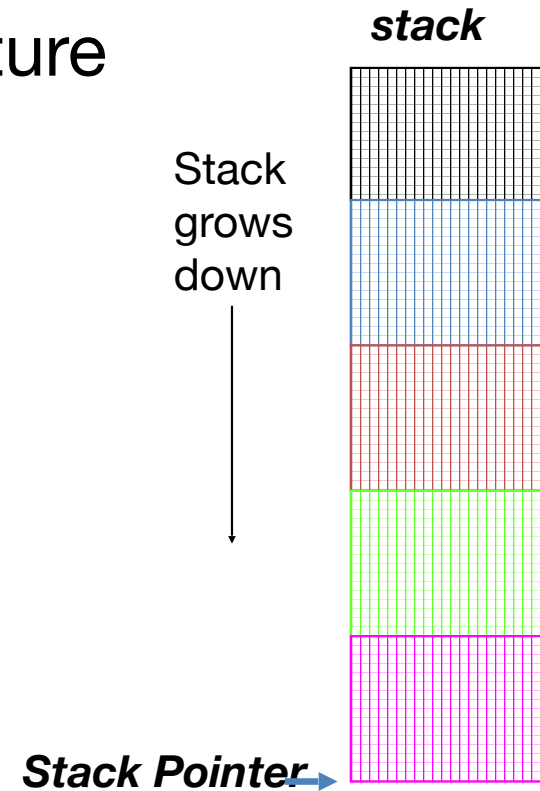
Stack Pointer →



Stack Animation

- Last In, First Out (LIFO) data structure

```
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)
{
}
```



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));
```

“Cast” operation, changes type of a variable. Here changes (**void \***) to (**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

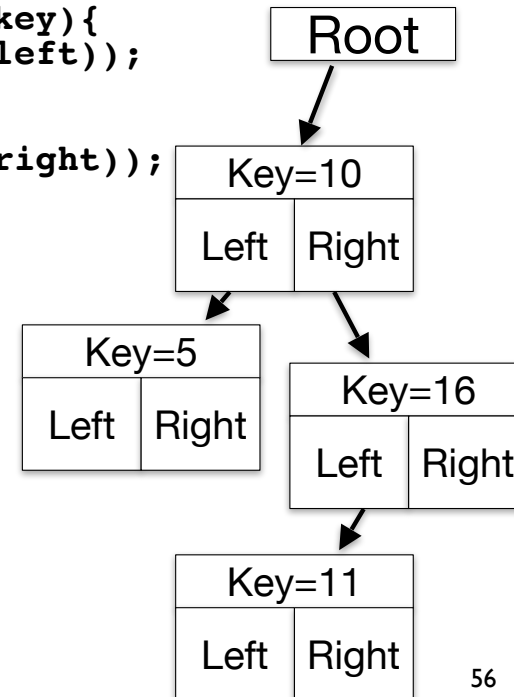
```
typedef struct node {
    int key;
    struct node *left; struct node
    *right;
} Node;

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 used by `malloc()`
  - 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:  

```
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  

```
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`  
....  
`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 address) 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