Bugs, and Pointers
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 in ANSI C: `for (int i=0; ...`
  - Correct in C99 (and beyond): `for (int i=0; ...`
An Important Note: Undefined Behavior…

- A lot of C has “Undefined Behavior”
  - This means it is often unpredictable behavior
    - It will run one way on one computer…
    - But some other way on another
    - Or even just be different each time the program is executed!

- Often characterized as “Heisenbugs”
  - Bugs that seem random/hard to reproduce, and seem to disappear or change when debugging
  - Cf. “Bohrbugs” which are repeatable
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.
- Do you think they use signed or unsigned numbers? Negative address?!

Don’t confuse the **address** referring to a memory location with the **value** stored in that location.

For now, the abstraction lets us think we have access to $\infty$ memory, numbered from 0…
- An address refers to a particular memory location. In other words, 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
How to create a pointer:

& operator: get address of a variable

```c
int *p, x;
```

```c
x = 3;
```

```c
p = &x;
```

```
Note the "*" gets used 2 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 value pointed to

```c
printf("p points to %d\n", *p);
```
- How to change a variable pointed to?
  - Use dereference * operator on left of =

```c
*p = 5;
```

Diagram:
- Pointer `p` points to variable `x` which is initially 3.
- After `*p = 5;` executes, `x` becomes 5.
Pointers and Parameter Passing (1/2)

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

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

\[ y \text{ is still } = 3 \]
Pointers and Parameter Passing (2/2)

- How to get a function to change a value?

```c
void addOne (int *p) {
    *p = *p + 1;
}
int y = 3;
addOne(&y);

y is now = 4
```
More C Pointer Dangers

- Declaring a pointer just allocates space to hold the pointer – it does not allocate something to be pointed to!
- **Local variables in C are not initialized**, they may contain anything.
- What does the following code do?

```c
void f()
{
    int *ptr;
    *ptr = 5;
}
```
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 time
  - Dangling references and memory leaks
Using Pointers Effectively
Pointers

- Pointers are used to point to any data type (int, char, a struct, etc.).

- Normally a pointer can only point to one type (int, char, a struct, etc.).
  - void * is a type that can point to anything (generic pointer)
  - Use sparingly to help avoid program bugs... and security issues... and a lot of 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 and is initially pointing to the function foo.
    - (*fn) (x, y) will then call the function.
Points 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;
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 */ }
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
- Eg., 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.
Arrays
Arrays (1/5)

- **Declaration:**
  - `int ar[2];`
  - ...declares a 2-element integer array
  - An array is really just a block of memory

- **Declaration and initialization**
  - `int ar[] = {795, 635};`
  - declares and fills a 2-elt integer array

- **Accessing elements:**
  - `ar[num]`
  - returns the num\(^{th}\) element.
Arrays (2/5)

- Arrays are (almost) identical to pointers
  - `char *string` and `char string[]` are nearly identical declarations
  - They differ in very subtle ways: incrementing, declaration of filled arrays

- **Key Concept**: An array variable is a “pointer” to the first element.
Consequences:
- `ar` is an array variable but looks like a pointer in many respects (though not all)
- `ar[0]` is the same as `*ar`
- `ar[2]` is the same as `*(ar+2)`
- We can use pointer arithmetic to access arrays more conveniently.

Declared arrays are only allocated while the scope is valid

```c
char *foo() {
    char string[32]; ...;
    return string;
}
```

is incorrect
Arrays (4/5)

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

- Why? **SINGLE SOURCE OF TRUTH**
  - You’re utilizing *indirection* and *avoiding maintaining two copies* of the number 10
Arrays (5/5)

- **Pitfall:** An array in C does not know its own length, & bounds not checked!
  - Consequence: We can accidentally access off the end of an array.
  - Consequence: We must pass the array and its size to a procedure which is going to traverse it.

- **Segmentation faults and bus errors:**
  - These are VERY difficult to find; be careful!
  - You’ll learn how to debug these in lab…
### Pointer Arithmetic

- **pointer + n**
  - Adds $n \times \text{sizeof}($“whatever pointer is pointing to”) to the memory address

- **pointer – n**
  - Adds $n \times \text{sizeof}($“whatever pointer is pointing to”) to the memory address
- Java and C pass parameters “by value”
  - procedure/function/method gets a copy of the parameter, so changing the copy cannot change the original

```c
void addOne (int x) {
    x = x + 1;
}

int y = 3;
addOne(y);

y is still = 3
```
How to get a function to change a value?

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

int y = 3;
addOne(&y);

y is now = 4
```
Pointers (3/4)

- But what if you want to change a pointer?
  - What gets printed?

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

int A[3] = {50, 60, 70};
int *q = A;
IncrementPtr(q);
printf("*q = %d\n", *q);
```
Pointers (4/4)

- **Idea! Pass a pointer to a pointer!**
  - Declared as **h**
  - Now what gets printed?

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

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

A `q` `q` `q`

```
50 60 70
```

*q = 60

Garcia, Nikolić

Introduction to C (60)
Function Pointer Example
map (actually `mutate_map` easier)

```c
#include <stdio.h>

int x10(int), x2(int);
void mutate_map(int [], int n, int(*)(int));
void print_array(int [], int n);

int x2 (int n) { return 2*n; }
int x10(int n) { return 10*n; }

void mutate_map(int A[], int n, int(*)(int)) {
    for (int i = 0; i < n; i++)
}

void print_array(int A[], int n) {
    for (int i = 0; i < n; i++)
        printf("%d ",A[i]);
    printf("\n");
}

int main(void)
{
    int A[] = {3,1,4}, n = 3;
    print_array(A, n);
    mutate_map (A, n, &x2);
    print_array(A, n);
    mutate_map (A, n, &x10);
    print_array(A, n);
}
```

% ./map
3 1 4
6 2 8
60 20 80