1. Pointer Lesson 1 Outline
2. A Pointer Experiment
3. Point!
4. What is a Pointer?
5. NULL Pointer
6. Are Pointers Useful?
7. Pointers and Allocation
8. What Does \texttt{malloc} Do?
9. Pointers and Deallocation
10. Function Arguments
11. Pass by Copy Example
11. Pass by Copy or Pass by Reference
12. Pass by Reference
13. Pass by Reference Example
14. The Address Operator $\&$
15. Address Operator and \texttt{scanf}
16. Pass by Copy vs Pass by Reference \#1
17. Pass by Copy vs Pass by Reference \#2
18. Pass by Copy vs Pass by Reference \#3
19. Pass by Reference Bad Example
20. Pass by Reference Good Example
21. Is Pass by Reference Really by Reference?
A Pointer Experiment

1. Take out a sheet of scrap paper.
2. Tear it in half.
3. Tear it in half again.
4. On one of the quarter sheets, write **legibly** either:
   - your full name (first and last), or
   - an integer from 1 to 100.
5. Fold it in half.
6. Fold it in half again.
7. When the hat comes around the first time, put your quarter sheet of paper into it.
8. When the hat comes around the second time, take a random quarter sheet of paper out of it. If you draw your own name, take out another one and put your name back in.
9. Let’s pick someone.
10. Have them stand up and read their piece of paper, then stay standing.
11. If they read a name, that person should also stand up, and the person who read their name should point at them.
12. Let’s do several of those around the room.
13. So the people pointing at other people are “pointers,” and the people who have a number are “values.”
What is a Pointer?

A **pointer** is a variable whose value is an address.

```c
float* float_pointer;
```

This means:

Grab a bunch of bytes in memory,

```c
ame them float_pointer, and think of them as storing an address, which is a special kind of int.
```

**How many bytes?**

On most platforms that you can buy today, a pointer is 8 bytes.
NULL Pointer

A NULL pointer is a pointer that points to nowhere.

```c
float* float_pointer = (float*)NULL;
```

This initialization statement means that the float pointer named `float_pointer` should initially point to nowhere.

Note that NULL is a C named constant. On most platforms, its value is zero, but that doesn’t have to be the case.
Are Pointers Useful?

We’ve already seen a context where pointers are useful: dynamic allocation of arrays.

A dynamically allocated array is really just a pointer to the first byte of the first element of that array:

```c
float* list1_input_value =
    (float*)NULL;
```

```c
list1_input_value =
    (float*)malloc(sizeof(float) * number_of_elements);
```

```c
list1_input_value
```

```c
list1_input_value
```
Pointers and Allocation

When you allocate an array

```c
list1_input_value = (float*)malloc(sizeof(float) * number_of_elements);
```

you’re setting a pointer variable’s value to:

- the address of
- the first byte of
- the first element of
- the array.
What Does `malloc` Do?

The `malloc` function finds a block of memory that is otherwise not being used, claims it, and returns its address (that is, a pointer to the block’s first byte).

```c
list1_input_value = (float*)malloc(sizeof(float) * number_of_elements);
```

In this case, `malloc` finds an unclaimed block of `sizeof(float) * number_of_elements` bytes, lays claim to that block, and returns its address to be assigned to `list1_input_value` (which is a pointer, which is to say its value is an address in main memory).
Pointers and Deallocation

When you deallocate an array
   free(list1_input_value);
you’re releasing the block of memory that contains the array;
that is, you’re no longer claiming it.
But, that doesn’t change the value of the pointer variable.
The pointer’s value is still the address of the block of memory – which no longer is the array.
This is BAD BAD BAD!
So, you have to assign NULL to (nullify) the pointer
IMMEDIATELY:
   list1_input_value = (float*)NULL;
Function Arguments

When you call a function in C and you pass it some arguments, those arguments are *passed by copy*.

This means that the formal arguments in the function definition are actually copies of the actual arguments in the function call. They live at different addresses than the originals.

*Pass by copy* is also known as:

- pass by value;
- call by copy;
- call by value.
Pass by Copy Example

```c
#include <stdio.h>
int main ()
{ /* main */
    int x = 5;
    void my_increment(int var);

    printf("main: before call, x = %d\n", x);
    my_increment(x);
    printf("main: after call, x = %d\n", x);
} /* main */

void my_increment (int var)
{ /* my_increment */
    printf("my_increment: before inc, var = %d\n", var);
    var += 1;
    printf("my_increment: after inc, var = %d\n", var);
} /* my_increment */
```

```
cat my_bad_increment.c
gcc -o my_bad_increment my_bad_increment.c
my_bad_increment
main: before call, x = 5
my_increment: before inc, var = 5
my_increment: after inc, var = 6
main: after call, x = 5
```
Pass by Copy or Pass by Reference

Okay, so *pass by copy* means that changing the value of the copy doesn’t change the value of the original.

Is there a way to pass an argument so that, in the function, we can change the value of the formal argument, and that’ll change the value of the actual argument in the call?

Yes: *pass by reference.*
Pass by Reference

*Pass by reference* means that, instead of passing a copy of the actual argument, you pass the address of the actual argument.

If we can pass the address, then we can modify the value of the variable that lives at that address.
Pass by Reference Example

```c
#include <stdio.h>

int main ()
{
    int x = 5;
    void my_increment(int* varptr);
    printf("main: before call, x = %d\n", x);
    my_increment(&x);
    printf("main: after call, x = %d\n", x);
}

void my_increment (int* varptr)
{
    printf("my_increment: before inc, *varptr = %d\n", *varptr);
    *varptr += 1;
    printf("my_increment: after inc, *varptr = %d\n", *varptr);
}
```

```
gcc -o my_good_increment my_good_increment.c
my_good_increment
main: before call, x = 5
my_increment: before inc, *varptr = 5
my_increment: after inc, *varptr = 6
main: after call, x = 6
```
The Address Operator &

The *address operator* & is an operator that means “the address of:”

```c
#include <stdio.h>
int main ()
{
    int* ip = (int*)NULL;
    int i;

    ip = &i;
    i = 5;
    printf("i=%d, *ip=%d\n", i, *ip);
    *ip = 6;
    printf("i=%d, *ip=%d\n", i, *ip);
}
```

```
% gcc -o addr_op addr_op.c
% addr_op
i=5, *ip=5
i=6, *ip=6
```
Address Operator and `scanf`

We already know a case where we use the address operator: `scanf`.

When we call `scanf`, we want to change the value of the argument(s) at the end of the call; for example:

```c
scanf("%d", &number_of_elements);
```

We want to modify the value of `number_of_elements`. So we have to pass the **address** of this variable, so that `scanf` can change its value.
Pass by Copy vs Pass by Reference #1

In C, when an argument is passed to a function, the program grabs a new location in memory and copies the value of the actual argument into this new location, which is then used as the formal argument.

This approach is known by several names:

- *pass by value*
- *call by value*
- *pass by copy*
- *call by copy*

By contrast, if we use pointers – and possibly the address operator & in the actual argument(s) – then this accomplishes *pass by reference* (even though the pointer itself is passed by copy).
Pass by Copy vs Pass by Reference #2

We can visualize *pass by reference* by imagining Henry’s house, which has the address 123 Any Street.

We can refer to Henry’s house this way:

Henry’s house

But we can also refer to Henry’s house this way:

Dr. Neeman’s house

So, “Henry’s house” and “Dr. Neeman’s house” are two different names for the same location; they are *aliases*. 
Pass by Copy vs Pass by Reference #3

We can *refer* to Henry’s house this way:

Henry’s house

But we can also *refer* to Henry’s house this way:

Dr. Neeman’s house

So, “Henry’s house” and “Dr. Neeman’s house” are *aliases*: two different names for the same location.

With **pass by reference**, when we call a function, each actual argument and its corresponding formal argument are *aliases* of the same location in memory.
Pass by Reference Bad Example

```c
#include <stdio.h>
int main ()
{ /* main */
    int henrys_house;
    void who(int dr_neemans_house);

    who( henrys_house);
    printf("%d people live in Henry's house.\n", henrys_house);
} /* main */

void who (int dr_neemans_house)
{ /* who */
    printf("How many people live in Dr Neeman's house?\n");
    scanf("%d", &dr_neemans_house);
} /* who */
```

% gcc -o henrys_house_bad henrys_house_bad.c
% henrys_house_bad
How many people live in Dr Neeman's house?
4
134513624 people live in Henry's house.
Pass by Reference Good Example

```c
#include <stdio.h>

int main ()
{ /* main */
    int henrys_house;
    void who(int* dr_neemans_house);

    who(&henrys_house);
    printf("%d people live in Henry’s house.\n", henrys_house);
} /* main */

void who (int* dr_neemans_house)
{ /* who */
    printf("How many people live in Dr Neeman’s house?\n");
    scanf("%d", &dr_neemans_house);
} /* who */
```

```
% cat henrys_house_good.c
% gcc -o henrys_house_good henrys_house_good.c
% henrys_house_good
How many people live in Dr Neeman's house?
4
4 people live in Henry's house.
```
Is Pass by Reference Really by Reference?

In C, the **only** passing strategy is pass by copy. To pass by reference, we have to piggyback on top of pass by copy – because in C, **everything** is pass by copy. So, the **value** that we have to pass by copy is the **address** of the argument we want to be able to change, which we achieve using the **address operator** &.

In other words, in C pass by reference is actually pass by copy: you copy the address.