

What is a pointer?

- A variable that holds a memory address.
 - The variable points to the location of an object in memory.
- Not all pointers contain an address.
 - Pointers that don't contain an address are set to the `NULL` pointer.
 - Value of the `NULL` pointer is 0.
 - `NULL` is a macro for either:
 - `((void *)0)`
 - `0`
 - `0L`
 - The definition depends completely on the compiler.

Review: Memory Addresses

- Memory is stored in registers that can be accessed by a specific number (address).
- Usually, each byte has a unique address.
- Bytes are grouped into words (timeshare uses word size 4).

```
int main(void) {  
    int fib[] = { 0, 1, 1, 2, 3, 5, 8, 13, 21, 34 };  
    return 0;  
}
```

Address	00	04	08	0C
A000_7FA0	00000000	00000000	00000000	00000000
A000_7FB0	00000000	00000001	00000001	00000002
A000_7FC0	00000003	00000005	00000008	0000000D
A000_7FD0	00000015	00000022	00000000	9D000350
A000_7FE0	00000000	00000000	00000000	00000000

Pointers and addresses

- Pointers are said to point at the address they are assigned.
- Can assign a pointer the address of a variable using the address-of operator (&).
- Multiple pointers can point to the same address.

```
#include <stdio.h>

int main(void) {
    int a = 42;
    int *ptr_a = &a;
    int *ptr_b = &a;

    // Two pointers can point to the same address.
    printf("The address of pointer A is %p\n", ptr_a);
    printf("The address of pointer B is %p\n", ptr_b);
    return 0;
}
```

Address	00	04	08	0C
A000_7FA0	00000000	00000000	00000000	00000000
A000_7FB0	00000000	00000000	00000000	00000000
A000_7FC0	00000000	00000000	A0007FD0	A0007FD0
A000_7FD0	0000002A	00000000	00000000	00000000
A000_7FE0	00000000	00000000	00000000	00000000

How to use the & operator

Example of using the & operator to access the address of a variable:

```
#include <stdio.h>

int main(void) {
    int foo = 0;

    // Print out address of foo.
    printf("Address of foo: %p\n", &foo);
    return 0;
}
```

foo is an int

printing a
pointer

address of
(or pointer to)
of foo

Dereferencing a Pointer

- The object a pointer points to can be accessed through dereference, or indirection.
- A pointer can be dereferenced using the dereferencing operator (*).
 - `a * b` /* multiplication */
 - `*b` /* pointer dereference*/
- Useful for manipulating the values of several variables through call-by-reference.

```
#include <stdio.h>

void increment_two_ints(int *a, int *b) {
    *a += 1;
    *b += 1;
    return;
}

int main(void) {
    int x = 3;
    int y = 4;
    increment_two_ints(&x, &y);

    // Now, x is 4 and y is 5.
    printf("The value of x is now: %d\n", x);
    printf("The value of y is now: %d\n", y);
    return 0;
}
```

1 May 2023

How to use the * operator

Example of using * to instantiate a pointer variable and using it to dereference a pointer.

Declare an int:
foo is an int

Declare a pointer bar:
*bar is an int

```
#include <stdio.h>

int main(void) {
    int foo = 13;

    // The * denotes that bar is a pointer variable.
    // The address of foo is stored in bar.
    int *bar = &foo;

    // Dereference bar using * to print out value at its stored address.
    printf("The value in the address stored in bar: %d\n", *bar);
    return 0;
}
```

printing an int

bar gets
address of
(or pointer to)
foo

*bar is an int



```

#include <assert.h>
#include <stdio.h>

int main(void) {
    int foo = 5;
    int *bar = &foo;
    int **baz = &bar;
    int ***qux = &baz;
    321
    ***qux += 1;
    assert(foo == 6);
    return 0;
}

```



Pointers
Have
Addresses,
Too

Benefits of Pointers

- Can be used when passing actual values is difficult.
- Can “return” more than one value from a function.
- Building dynamic data structures.
- Useful for passing large data structures around.
 - Pointers are efficient for this since copies of data structures don't need to be pushed into the stack.


```

void inc_by_ref(int *x) {
    *x = (*x) + 1;
    return;
}

int inc_by_val(int x) {
    return x + 1;
}

int main(void) {
    int x = 5;
    inc_by_ref(&x);
    x = inc_by_val(x);
    return 0;
}

```

Passing by value versus Passing by reference

- “Passing by value” duplicates passed values onto stack.
- “Passing by reference” duplicates a pointer onto the stack.



```
#include <stdio.h>

int main(void) {
    int age;
    double gpa;

    // Pass age and gpa variables by reference.
    printf("Please enter your age and gpa: ");
    scanf("%d %lf", &age, &gpa);
    return 0;
}
```

Passing by reference

- Allows “returning” multiple values.
 - As with `scanf()`
- Allows passing large amounts of data quickly.
 - You’re not copying the data, just telling where it is stored.

```
int main(void) {
    int a[2][2];
    int x = matrix_determinant_by_val(a[0][0], a[0][1], a[1][0], a[1][1]);
    int x = matrix_determinant_by_ref(a);
    return 0;
}
```

Summary

- A pointer is a variable that contains a memory address (just a value).
 - Pointers can point to functions as well.
- Pointers that don't contain a valid address should point to `NULL`.
- The address of a variable can be obtained with the address-of operator (`&`).
- The object a pointer points to can be accessed by dereferencing the pointer (`*`).
- Arbitrary levels of pointing: pointers can point to pointers which point to pointers, which point to pointers, which ...
- Useful for passing around large data structures.