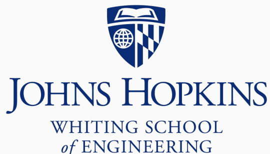
Pointers

Ben Langmead

ben.langmead@gmail.com

www.langmead-lab.org



Source markdown available at github.com/BenLangmead/c-cpp-notes

Pointers

All problems in computer science can be solved by another level of indirection...

Pointers

...except of course for the problem of too many indirections.

– David Wheeler

Pointers

A pointer is a variable that holds *the address of* (*pointer to*) a value

- `int *counter_ptr` is a *pointer to* an integer
- `char *welcome_message` is a *pointer to* a character

Pointers

Pointer can be assigned *the address of* a non-pointer (using &)

```
int counter = 0;              // regular variable
int *counter_ptr = &counter; // pointer variable
```

`counter_ptr` gets the address of `counter`

Pointers

`&` adds a layer of *indirection*

- `&a` is the address of (pointer to) `a`

`*` *removes* a layer of indirection

- `b` is a pointer, `*b` is the variable it points to

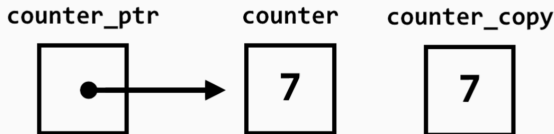
```c
int counter = 7;                 // variable
int *counter_ptr = &counter;     // = counter's address
int counter_copy = *counter_ptr; // = copy of counter
```

Pointers

We can represent code like this:

```
int counter = 7;                  // variable
int *counter_ptr = &counter;      // = counter's address
int counter_copy = *counter_ptr;  // = copy of counter
```

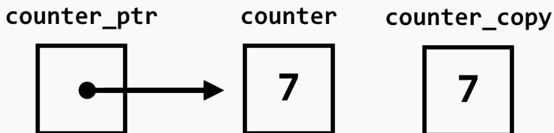
By drawing a diagram like this:



Pointers

For normal variables, we write their value in a box labled with their name

For pointers, we draw an arrow to the variable pointed to

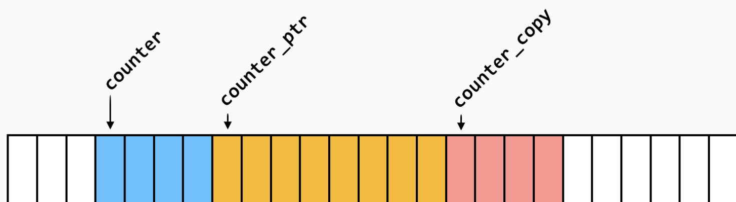


Pointers

Pointers live at addresses in memory just like normal variables

On modern (64-bit) computers, pointers occupy 8 bytes each

Memory layout for previous program might look like this:



Pointers

```c
#include <stdio.h>

int main() {
    int a = 40;
    int b = *&a; // reference-then-dereference
    printf("%d\n", b);
    return 0;
}
```

```
$ gcc -c ptr_eg0.c -std=c99 -pedantic -Wall -Wextra
$ gcc -o ptr_eg0 ptr_eg0.o
$ ./ptr_eg0
40
```

Pointers

A dereferenced pointer is something *you can assign to*

- Sometimes called an *lvalue*

```c
#include <stdio.h>

int main() {
    int counter = 7;             // variable
    int *counter_ptr = &counter; // = counter's address
    *counter_ptr = 10;           // this is OK!
    printf("counter=%d\n", counter);
    return 0;
}
```

```
$ gcc -o ptr_eg1 ptr_eg1.c -std=c99 -pedantic -Wall -Wextra
$ ./ptr_eg1
counter=10
```

Pointers

Pointers enable "pass by pointer", allowing us to modify variables in caller

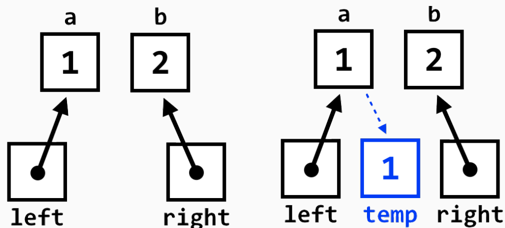
Pointers

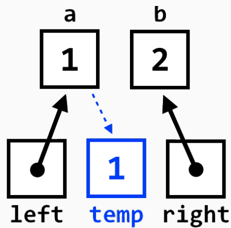
```c
#include <stdio.h>

void swap(int *left, int *right) {
    int tmp = *left;
    *left = *right;
    *right = tmp;
}

int main() {
    int a = 1, b = 2;
    swap(&a, &b);
    printf("a=%d, b=%d\n", a, b);
    return 0;
}
```

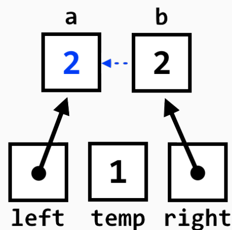
```
$ gcc -o swap1 swap1.c -std=c99 -pedantic -Wall -Wextra
$ ./swap1
a=2, b=1
```

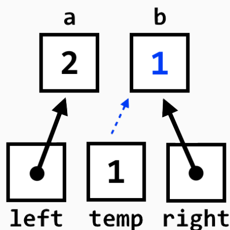
Pointers

When in doubt, draw a diagram:









Pointers

What's wrong here?

```
#include <stdio.h>

void swap(int left, int right) {
    int tmp = left;
    left = right;
    right = tmp;
}

int main() {
    int a = 1, b = 2;
    swap(a, b);
    printf("a=%d, b=%d\n", a, b);
    return 0;
}
```

Pointers

```
$ gcc -c swap2.c -std=c99 -pedantic -Wall -Wextra
$ gcc -o swap2 swap2.o
$ ./swap2
a=1, b=2
```

Forgot to make `left` and `right` be `int *`

Forgot to pass by pointer: `swap(&a, &b)`

Pointers

A value of `NULL` indicates an “empty” / “invalid” pointer

So what happens here?

```
char *null_ptr = NULL;
printf("address %p = %s\n", (void*)null_ptr, null_ptr);
```

Pointers

```
#include <stdio.h>

int main() {
    char *null_ptr = NULL;
    printf("address %p = %s\n", (void*)null_ptr, null_ptr);
    return 0;
}

$ gcc -c ptr_null_eg2.c -std=c99 -pedantic -Wall -Wextra
$ gcc -o ptr_null_eg2 ptr_null_eg2.o
$ ./ptr_null_eg2
address (nil) = (null)
```

Passing `NULL` as `%s` argument yields undefined behavior. `printf` is nice enough to print `(null)` and not crash, but we can't count on such forgiveness.

Pointers

What happens?

```
char *null_ptr = NULL;
printf("address %p = %c\n", (void*)null_ptr, *null_ptr);
//                   ^^                       ^^^^^^^^^
```

Pointers

```c
#include <stdio.h>

int main() {
    char *null_ptr = NULL;
    printf("address %p = %c\n", (void*)null_ptr, *null_ptr);
    //                   ^^                      ^^^^^^^^^
    return 0;
}
```

This straight up crashes

```
$ gcc -o ptr_null_eg3 ptr_null_eg3.c -std=c99 -pedantic -Wall -Wextra
$ ./ptr_null_eg3
Segmentation fault
```

Pointers

Dereferencing a pointer to memory that doesn't "belong" to you usually results in a segmentation fault or other crash

Dereferencing `NULL` is a particularly common mistake

- Always check return values for `NULL` errors (e.g. `fopen`)

Pointers

```c
int a = 7;
int *p = &a;
(*p)++;
```

Does this modify p, a, or both?

Pointers

```c
#include <stdio.h>

int main() {
    int a = 7;
    int *p = &a;
    printf("%p %d\n", (void*)p, a);
    (*p)++;
    printf("%p %d\n", (void*)p, a);
}
```

```
$ gcc -c ptr_eg4.c -std=c99 -pedantic -Wall -Wextra
$ gcc -o ptr_eg4 ptr_eg4.o
$ ./ptr_eg4
0x7ffdee08f6b4 7
0x7ffdee08f6b4 8
```

Answer: modifies a

Pointers

```
int a = 7;
int *p = &a;
p++; // used to be (*p)++
```

Does this modify p, a, or both?

Pointers

```
#include <stdio.h>

int main() {
    int a = 7;
    int *p = &a;
    printf("%p %d\n", (void*)p, a);
    p++;
    printf("%p %d\n", (void*)p, a);
}

$ gcc -c ptr_eg5.c -std=c99 -pedantic -Wall -Wextra
$ gcc -o ptr_eg5 ptr_eg5.o
$ ./ptr_eg5
0x7fff4e188844 7
0x7fff4e188848 7
```

Answer: modifies p

Pointers

Pointer *arithmetic* is possible

- Pointer can be operand in addition and subtraction, causing pointer to "seek" forward and backward across memory slots

```
$ ./ptr_eg5
0x7fffb585f094 7
0x7fffb585f098 7
```

Here the pointer advanced by 4 bytes

- It's an `int *` and `int` is 4 bytes long
- We advanced by 1 slot (p++), so that's 4 bytes

Pointers

`ptr1 = ptr2` - assignment between same-type pointers works

`ptr1 == ptr2` - true if `ptr1` and `ptr2` point to same place

`ptr1 == NULL` - asks if `ptr1` is `NULL` (equals 0)

Pointers

- You can print a pointer, e.g. `printf("%p", ptr)`
- A pointer is just an (unsigned) integer
- `NULL` equals 0; usually indicates "empty" or "invalid" pointer
- Dereferencing a pointer to memory that doesn't "belong" to you will result in a crash
- Pointers can be operated on by +, -, +=, ++, =, ==

Pointers

We saw that these differ in terms of whether p or a is changed:

```
int a = 7;
int *p = &a;
p++; // changes p
```

```
int a = 7;
int *p = &a;
(*p)++; // changes a
```

Related question: what does it mean for a pointer to be `const`?

Pointers

Putting const before the pointer type means the variable *pointed to* can't be modified

```
#include <stdio.h>

int main() {
    int a = 7;
    const int *p = &a;
    printf("%p %d\n", (void*)p, a);
    (*p)++;
    printf("%p %d\n", (void*)p, a);
}
```

```
$ gcc -c ptr_const_eg1.c -std=c99 -pedantic -Wall -Wextra
ptr_const_eg1.c: In function 'main':
ptr_const_eg1.c:7:9: error: increment of read-only location '*p'
     (*p)++;
         ^~
```

Pointers

`const int *p` - variable *pointed to* can't be modified

`int * const p` - the *pointer* can't be modified

`const int * const p` - *neither* can be modified