Pointers

2024 Winter APS 105: Computer Fundamentals
Jon Eyolfson

Lecture 12
1.0.1

Recall: Computers Just Store Numbers

Assuming we have a 64 bit (8 byte) binary number, we can represent it as a whole number using:

2^{63}	2^{62}	2^{61}	2^{60}	...	2^{2}	2^{1}	2^{0}
0	0	0	0	...	1	0	1

What decimal number would this be?

Binary Numbers are Too Long to Write Out and Read

Decimal numbers are for humans, but computers are based on powers of 2

Writing numbers using base 16 instead of 2 or 10 is more convenient

Decimal uses digits: 0 - 9
Binary uses bits: 0 - 1
Base 16 uses: 0 - 9, and 6 other characters

We Call the Base 16 Number System Hexadecimal

We borrow letters to represent the values: 10 through 15

- 10 is a
- 11 is b
- 12 is c
- 13 is d
- 14 is e
- 15 is f

We could call a hexadecimal digit (0-9 and a-f) a hexit (but no one does)

We Call the Base 16 Number System Hexadecimal

We borrow letters to represent the values: 10 through 15

- 10 is `a`
- 11 is `b`
- 12 is `c`
- 13 is `d`
- 14 is `e`
- 15 is `f`

We could call a hexadecimal digit (0-9 and a-f) a hexit (but no one does)

In C, a hex (short for hexadecimal number), starts with `0x`

Not testable, but will help you understand computers

The Same Rules Apply, Just with a New Base

This turns out to be convenient because each hex digit represents 4 bits
This works well with bytes: 2 hex digits represents 8 bits (or 1 byte)

16^{15}	16^{14}	16^{13}	16^{12}	...	16^{2}	16^{1}	16^{0}
0	0	0	0	...	0	f	4

What decimal number would this be?

The Same Rules Apply, Just with a New Base

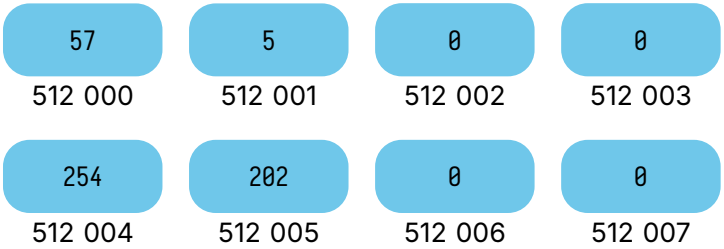
This turns out to be convenient because each hex digit represents 4 bits
This works well with bytes: 2 hex digits represents 8 bits (or 1 byte)

16^{15}	16^{14}	16^{13}	16^{12}	...	16^{2}	16^{1}	16^{0}
0	0	0	0	...	0	f	4

What decimal number would this be? $(15 \times 16) + (4 \times 1) = 244$

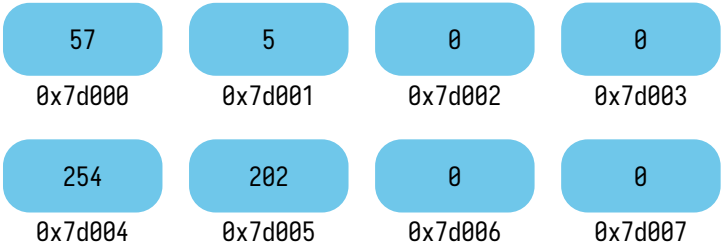
An Address Contains a Byte (Value in Blue, Address Below)

(Recall from Lecture 2)

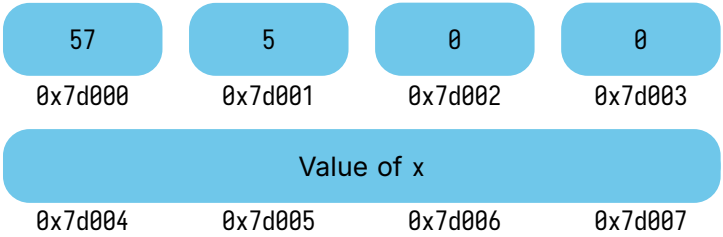


We Write Memory Addresses in Hex

(This is equivalent to the previous slide)



C Stores the Value of `int x` Somewhere in Memory



A Pointer is the Starting Address of a Value in Memory

The & operator is the address of, its result is the pointer to the value
  For values that take up multiple bytes, it's always the lowest address

In the previous example, `&x` would be `0x7d004`

Pointers Are a New Type

Assume we have:

```
int x = 1;
```

We can't do:

```
int z = &x;
```

The type of `&x` is `int *`

It's a pointer to an integer value

Each Time We Take the Address of a Variable, We Add * to its Type

Assume we have:

```
int x = 1;
```

We **can** do:

```
int *z = &x;
```

Each Time We Take the Address of a Variable, We Add * to its Type

Assume we have:

```
int x = 1;
```

We **can** do:

```
int *z = &x;
```

The type of `&z` is `int **`

It's a pointer to a pointer to an integer value

You Can Only Take the Address of a Variable

A variable stores a value in memory
(a value by itself may never be in memory)

You Can Only Take the Address of a Variable

A variable stores a value in memory
(a value by itself may never be in memory)

If you try to do something like: &4
You may get a very unhelpful message

```
error: cannot take the address of an rvalue of type 'int'
```

We Can Use the * Operator to Access the Value at an Address

Assume we have:

```
int x = 1;
int *z = &x;
```

We can do:

```
int y = *z;
```

After that statement, `y = 1`

Accessing a value through a pointer is called dereferencing
In the code above we'd say we dereference `z`

Each Use of the * Operator Removes a * from the Result Type

If we have the variable:

```
int **z;
```

The type of *z is `int *`

We Can Change Values of Variables Through Pointers

```
int main(void) {
    int x = 1;
    int y = 2;
    int *z = &x;
    *z = 3;
    return 0;
}
```

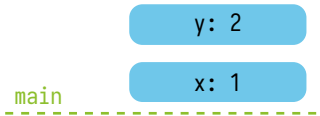
main

We Can Change Values of Variables Through Pointers

```
int main(void) {
    int x = 1;
    int y = 2;
    int *z = &x;
    *z = 3;
    return 0;
}
```

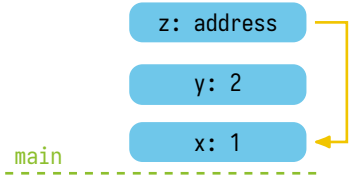
We Can Change Values of Variables Through Pointers

```
int main(void) {
    int x = 1;
    int y = 2;
    int *z = &x;
    *z = 3;
    return 0;
}
```



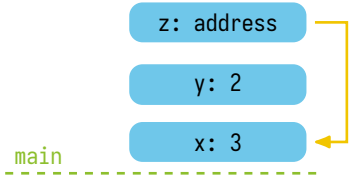
We Can Change Values of Variables Through Pointers

```
int main(void) {
    int x = 1;
    int y = 2;
    int *z = &x;
    *z = 3;
    return 0;
}
```



We Can Change Values of Variables Through Pointers

```
int main(void) {
    int x = 1;
    int y = 2;
    int *z = &x;
    *z = 3;
    return 0;
}
```



Functions Can Change Values of Variables Through Pointers

```
void setThree(int *p) {
    *p = 3;
}

int main(void) {
    int x = 1;
    int y = 2;
    int *z = &x;
    setThree(z);
    return 0;
}
```

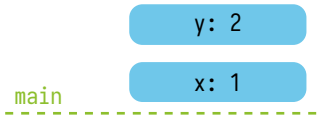
main

Functions Can Change Values of Variables Through Pointers

```
void setThree(int *p) {
    *p = 3;
}

int main(void) {
    int x = 1;
    int y = 2;
    int *z = &x;
    setThree(z);
    return 0;
}
```

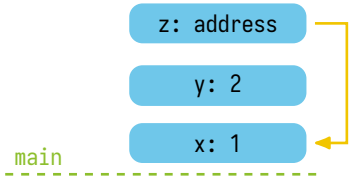
Functions Can Change Values of Variables Through Pointers

```
void setThree(int *p) {
    *p = 3;
}

int main(void) {
    int x = 1;
    int y = 2;
    int *z = &x;
    setThree(z);
    return 0;
}
```



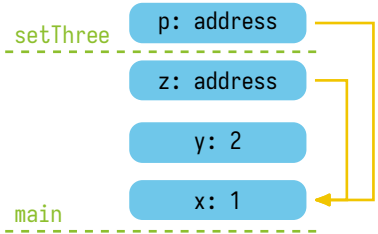
Functions Can Change Values of Variables Through Pointers

```
void setThree(int *p) {
    *p = 3;
}

int main(void) {
    int x = 1;
    int y = 2;
    int *z = &x;
    setThree(z);
    return 0;
}
```



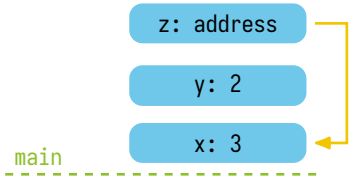
Functions Can Change Values of Variables Through Pointers

```
void setThree(int *p) {
    *p = 3;
}

int main(void) {
    int x = 1;
    int y = 2;
    int *z = &x;
    setThree(z);
    return 0;
}
```



Functions Can Change Values of Variables Through Pointers

```
void setThree(int *p) {
    *p = 3;
}

int main(void) {
    int x = 1;
    int y = 2;
    int *z = &x;
    setThree(z);
    return 0;
}
```



We Can Print the Address of a Pointer

The format specifier for pointers is: `%p`
  It expects a type of `void *`

A `void *` is basically C saying the type is a generic pointer
  We don't need to know the type of the value it's pointing to

You cannot dereference a `void *`
  We're allowed to cast a pointer to any type to a `void *`

We Can Add Print Statements to Verify

```c
#include <stdio.h>

void setThree(int *p) {
    printf("p [address is %p] = %p\n", (void *) &p, (void *) p);
    printf("  *p = %d\n", *p);
    *p = 3;
}

int main(void) {
    int x = 1; int y = 2; int *z = &x;
    printf("x [address is %p] = %d\n", (void *) &x, x);
    printf("y [address is %p] = %d\n", (void *) &y, y);
    setThree(z);
    setThree(&y);
    printf("x [address is %p] = %d\n", (void *) &x, x);
    printf("y [address is %p] = %d\n", (void *) &y, y);
    return 0;
}
```

Your Memory Addresses Will Very Likely be Different

The result of running the program (for me) is:

```
x [address is 0xffffd2c47f38] = 1
y [address is 0xffffd2c47f34] = 2
p [address is 0xffffd2c47ee8] = 0xffffd2c47f38
*p = 1
p [address is 0xffffd2c47ee8] = 0xffffd2c47f34
*p = 2
x [address is 0xffffd2c47f38] = 3
y [address is 0xffffd2c47f34] = 3
```

Note, the address of `p` may change between function calls

Now, We Should Understand the Swap Function

```c
#include <stdio.h>
#include <stdlib.h>

void swap(int* a, int* b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

int main(void) {
    int a = 1;
    int b = 2;
    printf("main (before swap) a: %d, b: %d\n", a, b);
    swap(&a, &b);
    printf("main (after swap)  a: %d, b: %d\n", a, b);
    return EXIT_SUCCESS;
}
```