Intro to Pointers and Memory Addresses

Foundations to help understand Chapter 3.1

COMP2401 F25

Connor Hillen (Lecturer, He/Him)
connorhillen@cunet.carleton.ca

Learning Outcomes

Students who engage in this lecture and complete any required readings, assignments, and other practice materials should be prepared to demonstrate the following on exams and projects:

- **Understand** how data is stored at memory locations.
- **Apply** dereference operators to memory addresses.
- **Distinguish** the dereference operator and the address-of operator.

What is a Variable?

When we say something like:

```
int x = 42;
```

What are we really saying? What **is** x?

Variables: Where and What

It is helpful to divide our understanding of variables into a few parts:

1. **Identifier**: How will the programmer refer to the variable?
2. **Storage**: Where is the variable stored in memory?
3. **Value**: What is the current value of the variable?

Only the **value** actually exists in memory when the program is running.

Variables: Usage

Let’s provide clear definitions for the ways we work with variables, because they are treated differently by your compiler in C:

```c
int x; // Declaration: "Compiler, we need space for an integer, we'll call it x"
x = 42; // Assignment: "Store the value 42 where you reserved data for what we call x"
int y = 100; // Initialization: "Declare y and set it to 100"
printf("%d", x); // Access: "Get the value stored for x and use it"
```

Assembly

We will very briefly look at some assembly code.

You are **not** expected to understand assembly.

It is important to recognize that, in a way, the **goal of programming languages** is to output assembly code (which turns into machine code) that can be executed by the CPU to solve our problem.

Assembly: Low Level Languages

Machine Code

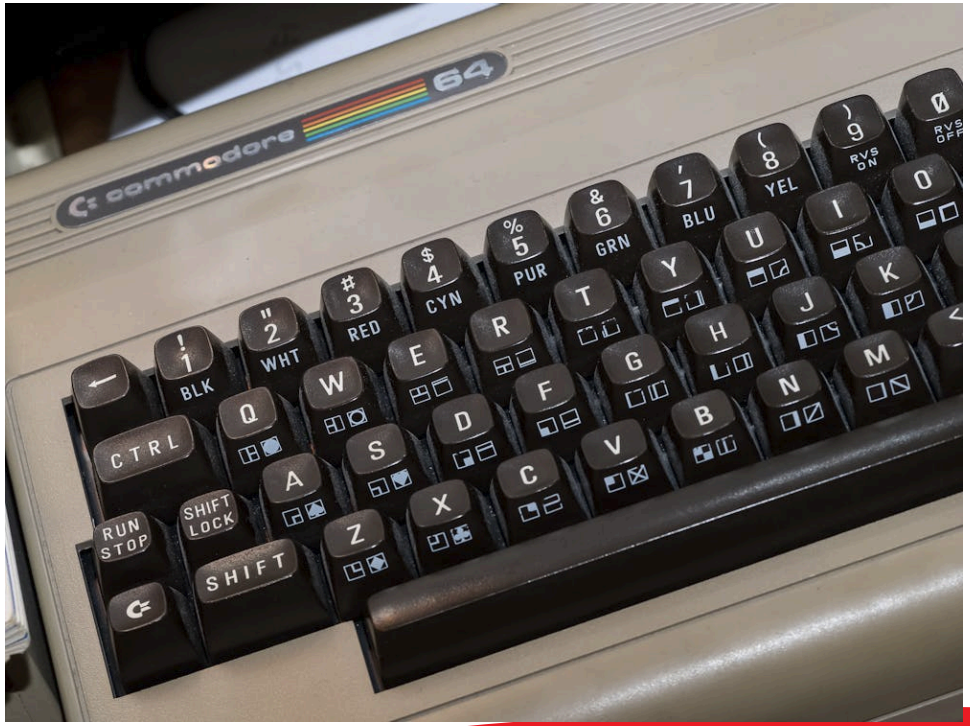
- Simple operations: copy data, add numbers, compare values.
- Directly uses addresses to move data in and out of the CPU

Assembly Code (Various languages)

- Human readable mappings to machine code

Commodore 64 Example

- Change the background colour?
- Specifically modify the RAM at address 53281.



C: “Mid” Level Language

Motivations

- **Addresses:** Let the compiler or operating system figure out the specific memory addresses for variables.
- **Abstraction:** Repeated patterns in assembly are simplified into keywords.
- **Efficiency:** Let the compiler work out the optimal assembly instructions for what your source code is trying to get across.

Mid-Level?

- Technically “high-level”, it is abstract.
- Not a *mapping* to assembly, but still has you thinking about similar concepts.
- Direct control; Like “driving stick” instead of automatic.

Abstraction: What IS a Variable?

```
1  int thisismyinteger = 421337;
2
3  int main() {
4      int anotherint = 12345678;
5      int my_small_int = thisismyinteger;
6      thisismyinteger += 42;
7  }
```

```
1   push   %rbp
2   mov    %rsp,%rbp
3   movl   $12345678,-4(%rbp)
4   mov    12025(%rip),%eax
5   mov    %eax,-0x8(%rbp)
6   mov    12016(%rip),%eax
7   add    $42,%eax
8   mov    %eax,12007(%rip)
9   mov    $0,%eax
10  pop    %rbp
11  ret
```

Abstraction: What IS a Variable?

```c
int thisismyinteger = 421337;

int main() {
    int anotherint = 12345678;
    int my_small_int = thisismyinteger;
    thisismyinteger += 42;
}
```

```
push   %rbp
mov    %rsp,%rbp
movl   $12345678,-4(%rbp)
mov    12025(%rip),%eax
mov    %eax,-0x8(%rbp)
mov    12016(%rip),%eax
add    $42,%eax
mov    %eax,12007(%rip)
mov    $0,%eax
pop    %rbp
ret
```

Where is the Variable?

In the assembly, the variable is just referred to by its address.

Addresses and Memory

Exercises to play with memory concepts

Memory Exercises

A series of exercises to learn about memory.

For now, forget about programming.

This is **not** C code, nor is it assembly.

Pen-and-paper exercises to **build intuition** about how your compiler and CPU wor with memory.

Memory Exercises

Some easy ones to start off…

1. $4 + 7 = ?$
2. $3 + 2 = ?$
3. $10 + 20 = ?$

These are considered **immediate values**.

To perform calculations, the CPU can use them directly.

Memory Table

Our memory can be considered a big array of data.

For our examples we will assume:

1. One location in memory can hold two decimal digits (00 - 99)
2. Memory is zero-indexed (the first location is 0)

So our memory might look like:

MEMORY = [00, 00, 00, 00, 00, 00, 00, 00, 00, 00, 00]

Memory Table

Let's use a table, though. It represents the same thing.

Label	Addr	Value
	00	00
	01	00
	02	00
	03	00
	04	00
	05	00
	06	00
	07	00
	08	00
	09	00
	10	00

Just remember that the address does **not** need to be stored anywhere. It's just the offset from the start of our memory.

Memory Table

Label	Addr	Value
	00	00
	01	09
	02	04
	03	40
	04	10
	05	00
	06	13
	07	02
	08	01
	09	00
	10	00

1. $01 + 02 =$
2. $03 + 04 =$
3. $80 + 20 =$

Memory Table

Label	Addr	Value
	00	00
	01	09
	02	04
	03	40
	04	10
	05	00
	06	13
	07	02
	08	01
	09	00
	10	00

1. $01 + 02 = 3$
2. $03 + 04 = 7$
3. $80 + 20 = 100$

Still just **immediate** values!

We need a way to say that we want the **value at** a specific address.

Dereference Operator, *

Asterisk / Star: *

`*x` is an operator that means “the value at address `x`”.

It is called the **dereference** operator, because `x` isn’t a value we necessarily care about, it **refers to** a value.

Memory Table

Label	Addr	Value
	00	00
	01	09
	02	04
	03	40
	04	10
	05	00
	06	13
	07	02
	08	01
	09	00
	10	00

`What is *01?` (Get **value-at** address 01)

Memory Table

Label	Addr	Value
	00	00
	01	09
	02	04
	03	40
	04	10
	05	00
	06	13
	07	02
	08	01
	09	00
	10	00

```
*01 == 09
```

Memory Table

Label	Addr	Value
	00	00
	01	09
	02	04
	03	40
	04	10
	05	00
	06	13
	07	02
	08	01
	09	00
	10	00

```
*02 + 03 == ?
```

Memory Table

Label	Addr	Value
	00	00
	01	09
	02	04
	03	40
	04	10
	05	00
	06	13
	07	02
	08	01
	09	00
	10	00

```
  *02 + 03
== 04 + 03
== 07
```

Memory Table

Label	Addr	Value
	00	00
	01	09
	02	04
	03	40
	04	10
	05	00
	06	13
	07	02
	08	01
	09	00
	10	00

1. `*01 == ??`
2. `01 + *02 == ??`
3. `*03 + 04 == ??`
4. `*05 + *06 == ??`
5. `*(02 + 07) == ??`

Memory Table

Label	Addr	Value
	00	00
	01	09
	02	04
	03	40
	04	10
	05	00
	06	13
	07	02
	08	01
	09	00
	10	00

What happens with `*15`?

Memory Table

Label	Addr	Value
	00	00
	01	09
	02	04
	03	40
	04	10
	05	00
	06	13
	07	02
	08	01
	09	00
	10	00

Segmentation Fault Error: We attempted to access memory outside of our allocated memory segment.

Memory Table: Double Dereference

Label	Addr	Value
	00	00
	01	09
	02	04
	03	40
	04	10
	05	00
	06	13
	07	02
	08	01
	09	00
	10	00

What would the result of `**07` be?

Memory Table

Label	Addr	Value
	00	00
	01	09
	02	04
	03	40
	04	10
	05	00
	06	13
	07	02
	08	01
	09	00
	10	00

We could also modify the memory using dereferencing.

```
*02 = 91
```

“The **value-at** address 02 is assigned the **value** 91”

Label	Addr	Value
	00	00
	01	09
	02	91
	03	40
	04	10
	05	00
	06	13
	07	02
	08	01
	09	00
	10	00

Variable Names

In higher level languages, we do not want to think about addresses directly.

We can **declare** a variable to name the value at an address and our compiler or operating system will work out the details.

Variable Names

If our source code has `int x;`, and the compiler decides that will be at address `02`, what might a fake “compiler” initialization look like, such that we would later do operations like `x = 5` and `x = 5 + 2` to work with values?

a) `x = 02`
b) `x = *02`
c) `x = &02`

The goal is that after declaring, we don’t have to consider the exact location again, we can just use the identifier `x`.

Variable Names

Label	Addr	Value
	00	00
	01	09
x	02	91
	03	40
	04	10
	05	00
	06	13
	07	02
	08	01
	09	00
	10	00

If we consider `x` to refer to `*02`, then we can use `x` and basically find-and-replace it for the remainder of operations.

```
Declare x as (*02)
x = x + 2                    (*02) = (*02) + 2
```

New Operator: Address-Of

Label	Addr	Value
	00	00
	01	09
x	02	91
	03	40
	04	10
	05	00
	06	13
	07	02
	08	01
	09	00
	10	00

So what if we **did** want the address of a variable?

The **address-of** operator is `&`. It basically removes the dereference operator.

So `&x` is like `*02`, just 02.

It makes no sense to say &02. There is no `*` to remove, nothing to get the address-of.

New Operator: Addresses and Values

Label	Addr	Value
a	00	00
b	01	00
c	02	00
	03	40
	04	10
	05	00
	06	13
	07	02
	08	01
	09	00
	10	00

Declare:

- `a = *00`
- `b = *01`
- `c = *02`

Initialize:

- `a = 5`
- `b = a`
- `c = &a`

Replace:

- `(*00) = 5`
- `(*01) = (*00)`
- `(*02) = &(*00) = 00`

Pointers and Addresses in C

Simple Application

Addresses in C

Remember: We don't really think in terms of "`*05`", that's just a helpful tool for better understanding C code.

Can you predict the output of the following code?

C Source Code

```
1  int a = 5;
2  int b = 10;
3  int c = a + b;
4  printf("%d\n", c);
```

Hypothetical Placeholder

```
1  a = (*40400)
2  b = (*40404)
3  c = (*40408)
4  (*40400) = 5
5  (*40404) = 10
6  (*40408) = (*40400) + (*40404)
7  printf("%d\n", (*40408));
```

Pointers in C

A **pointer** is like any other variable, except we use it to store memory locations.

- `int *x;` declares a variable named `x`.
- `*` doesn't dereference during **declaration**, it gives type information.
- `int *x`
  - Using `*x` will return you an `integer`.
  - `x` is a variable that holds an `int*` type.
- Nothing magical! Let's use our hypothetical notation…

C Source Code

```c
int a;
int *b;
a = 5;
b = &a;
printf("%d\n", *b);
```

Hypothetical Placeholder

```
a = (*40400)
b = (*40404)
(*40400) = 5
(*40404) = &(*40400)
printf("%d\n", *(*40404));
```

Label	Addr	Value
a	40400	00
	40401	00
	40402	00
	40403	00
b	40404	00
	40405	00
	40406	00
	40407	00

Why Pointers?

There are many reasons to use pointers. Make sure not to just memorize, but to think about what they are and where they end up being useful.

Most apparent: **Passing by reference** (pass by address).

- Sending arguments into a function **copies** the value into new memory
- Pointers let us copy a location into memory and then modify it!

Pass-by-Pointer

Frequently in C, we return an integer error code from the function.

If we do that, how can we get values returned?

```c
int calculate_area(int width, int height, int *area) {
  if (width < 0 || height < 0) { return -1; } // Error!
  *area = width * height; // Why does this work?
  return 0; // No error!
}
```

Pass-by-Pointer: Common Mistake!

```
1  int calc(int w, int h, int *a) {
2    if (w < 0 || h < 0) { return -1; }
3    *a = w * h;
4    return 0;
5  }
6
7  int main() {
8    int *area;
9    int error = calc(5, 10, area);
10
11   printf("Area is %d\n", *area); // Uh oh...
12 }
```

What's wrong here?

Pass-by-Pointer: Common Mistake!

```c
int calc(int w, int h, int *a) {
  if (w < 0 || h < 0) { return -1; }
  *a = w * h;
  return 0;
}

int main() {
  int *area;
  int error = calc(5, 10, area);

  printf("Area is %d\n", *area); // Uh oh...
}
```

```
main:area => (*100)
main:error => (*108)
calc:w => (*112)
calc:h => (*116)
calc:a => (*120)

-- Call Calc(5, 10, area) --
(*112) = 5
(*116) = 10
(*120) = ???

-- *a = w * h --
*(*120) = (*112) * (*116)
-- Becomes --
*(???) = 5 * 10
```

Segmentation Fault Error

If you try to access memory that your program doesn't "own", you have committed a segmentation fault, and your operating system will likely terminate the program with a "**Segmentation Fault Error**".

You will see **hundreds** of these.

Remember every time:

- Where am I dereferencing memory?
- Do I have memory on the other side of this?

Correct Pass-by-Pointer

How can we fix this?

```
1   int calc(int w, int h, int *a) {
2     if (w < 0 || h < 0) { return -1; }
3     *a = w * h;
4     return 0;
5   }
6
7   int main() {
8     int *area;
9     int error = calc(5, 10, area);
10
11    printf("Area is %d\n", *area); // Uh oh...
12  }
```

Correct Pass-by-Pointer

Store space for an integer.

Pass the address-of that variable.

```
1   int calc(int w, int h, int *a) {
2     if (w < 0 || h < 0) { return -1; }
3     *a = w * h;
4     return 0;
5   }
6
7   int main() {
8     int area;
9     int error = calc(5, 10, &area);
10
11    printf("Area is %d\n", *area); // Uh oh...
12  }
```

Course Consideration

In this course, try not to progress until you understand each piece of syntax.

If you are confused, read, review, ask questions!

Experiment! Run code, break it, modify it, experiment with exercises.

Write Intentionally

Know what each line of code your write does and why it does it.

Do not try to memorize quick rules, understand the concepts.

If you stumble and get confused while writing a line, pause, and learn what is confusing you. Rephrase it.

Learning Outcomes

Students who engage in this lecture and complete any required readings, assignments, and other practice materials should be prepared to demonstrate the following on exams and projects:

- **Understand** how data is stored at memory locations.
- **Apply** dereference operators to memory addresses.
- **Distinguish** the dereference operator and the address-of operator.