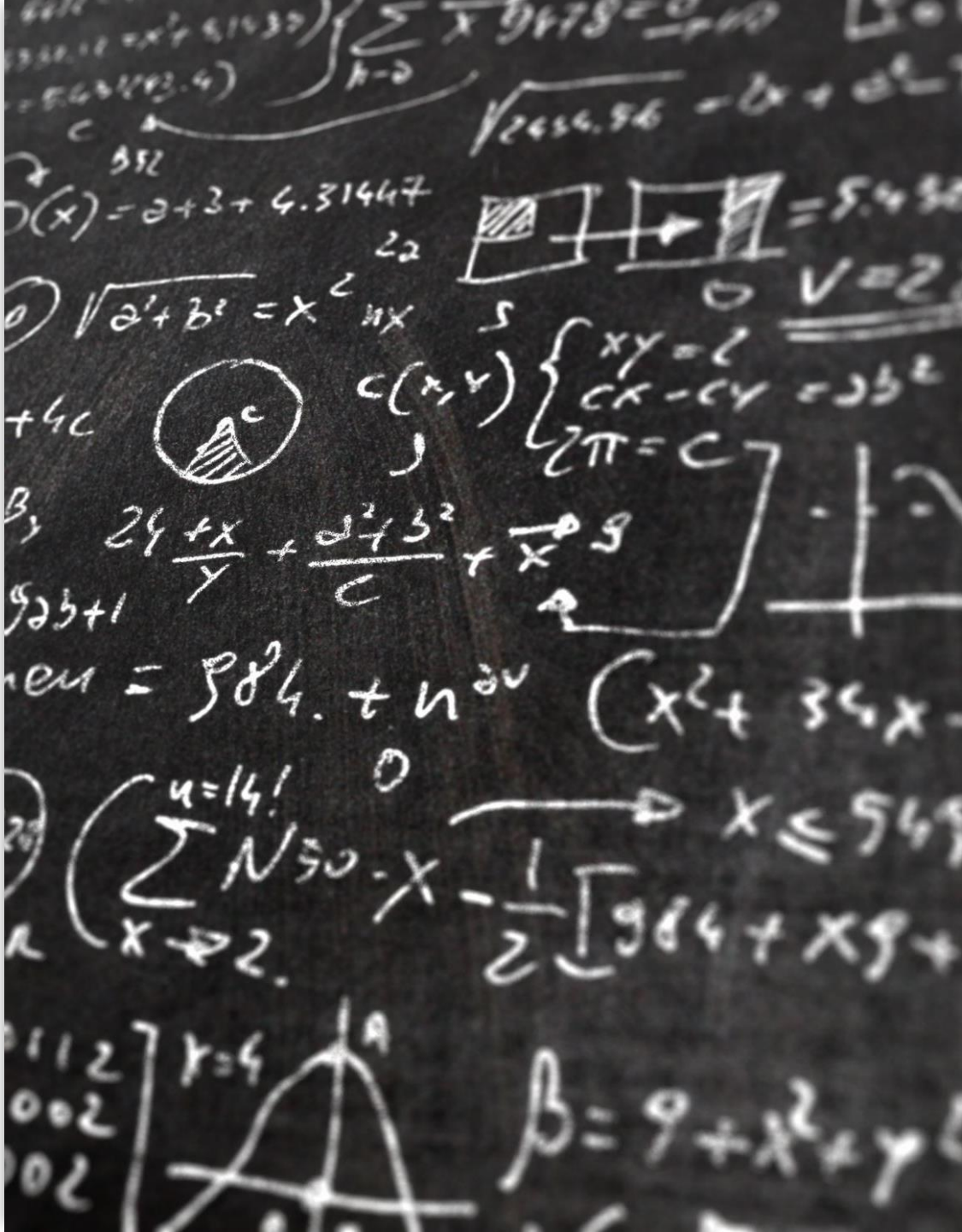
School of Computing & Communications | Lancaster University

# SCC.111 Software Development – Lecture 10: Pointers and Indirection

Adrian Friday, Nigel Davies, Hansi Hettiarachchi, Saad Ezzini

# This lecture

- What is **indirection**
- Why this is a valuable concept in programming
- How C does indirection (**pointers**!)
- Examples, and why we *should not be afraid!* ☺



# Indirection

- *Fundamental* concept in all fields of computing
- Thankfully
  - it's a relatively simple concept, and,
  - there are many real-world analogies!

# The Complaints Dept

- When you send a letter to the complaints department, c.f. a PO Box or contact a company on email or social media
- You don’t address an individual, rather you call ‘the contact number’ and it gets routed to a specific person within the company
- You have an indirect link to that person :)

# David Wheeler, FRS

- Mathematician by training
- Completed the world's first PhD in Computer Science in 1951
- Famously reported to have said "*All problems in computer science can be solved by another level of indirection... Except for the problem of too many layers of indirection.*"

# Indirection in C

- "**A pointer**" is a variable that contains '*the address of*' something else in memory such as another variable
- A pointer is not 'the thing itself', it's 'where to find it' – and this can then change at runtime
- It also has type so C knows what to expect, it's **type** is a pointer to something, such as *pointer to int*)
- *Pointers let you do great things!*

(K&R, ch. 5)

# Typical questions people ask about C

- How do I pass function parameters ‘by reference’ so I can modify them ?
- How can I return multiple values or arrays from a function ?
- Can I have data types of “variable” size instead of fixed sized arrays ?
- How do I represent strings ?
- *As we’ll see in a later lecture, we can also efficiently process arrays, strings, and create clever ‘linked’ data structures…*

The answer is **pointers**!

# First, variables (again)

Let’s check our understanding of how variables work

What actually happens when we say

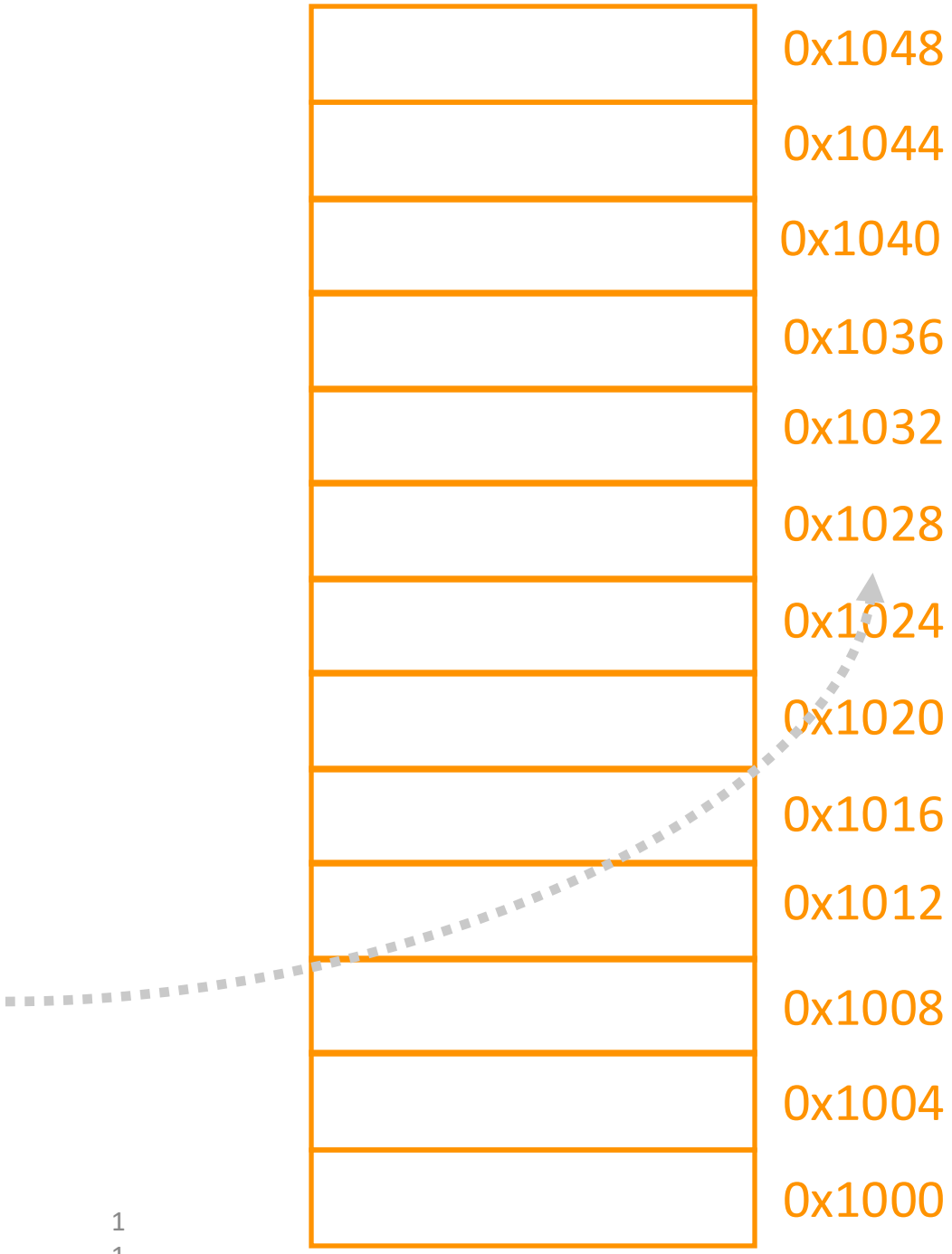
```
int x;

x = 65;
```

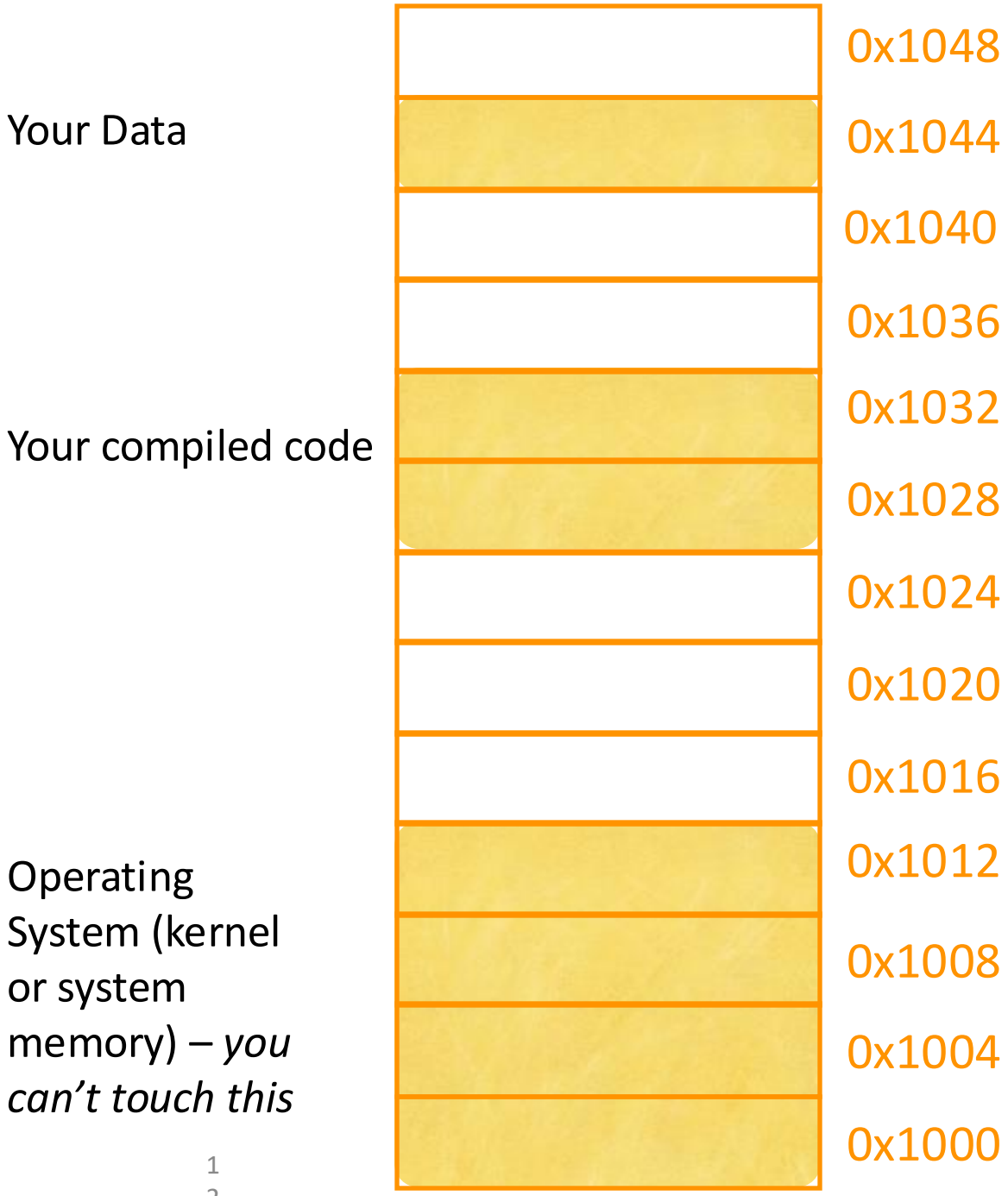
?

We can think of computer memory as a collection boxes *each with a unique location*

*N.B. Convention to write memory locations or addresses in hexadecimal*

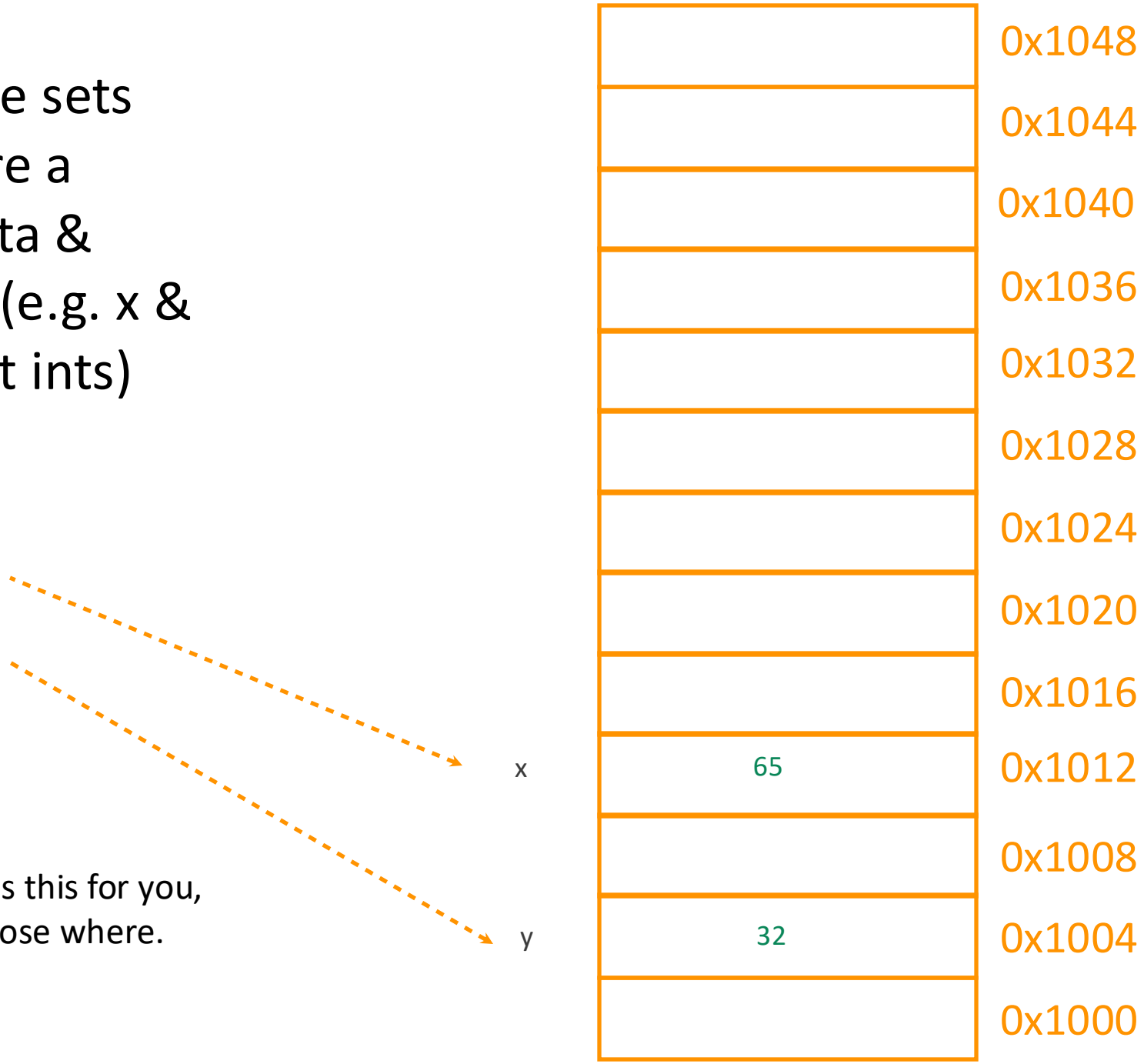


# Executing programs and their data get stored in this memory



Declaring a variable sets aside space to store a specific **type** of data & labels it's location (e.g. x & y are 4 byte/ 32-bit ints)

```
int x = 65;
int y = 32;
```

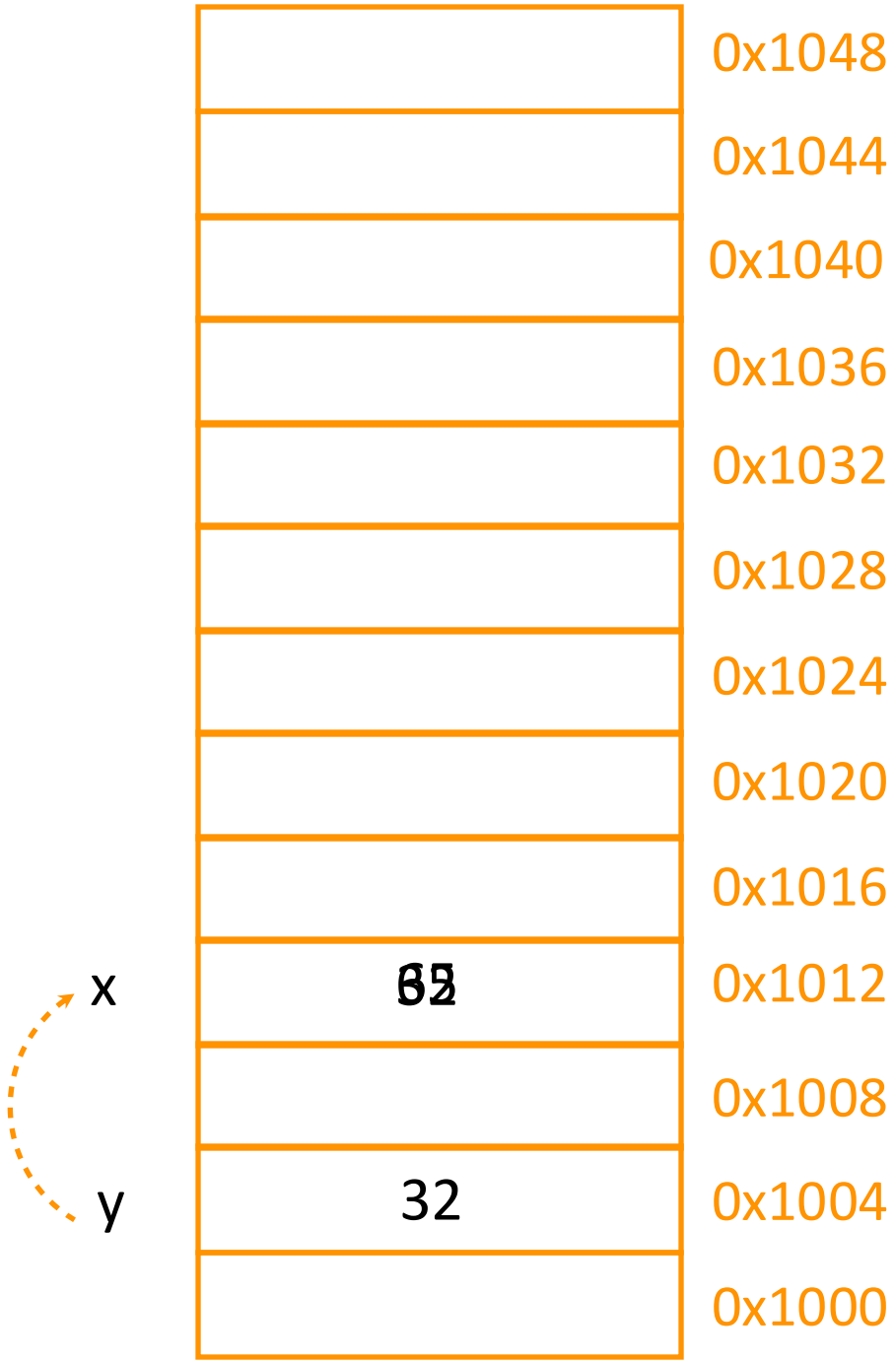



The compiler decides this for you, you *don't get* to choose where.

When we say something like x = y; we are taking a copy of the contents of the box **y** and storing it in the box **x**

When you declare a variable you are setting aside space to store information of a specific type and labelling that space

x = y;



# Pointers

- Imagine we declare a variable x that can contain an int
- How do we talk about the space labelled x (or where x is, rather than what x contains) ?
- Pointers give us a level of indirection, they *point to data's location*, they're variables so *we can change* which data 'they point to' at runtime

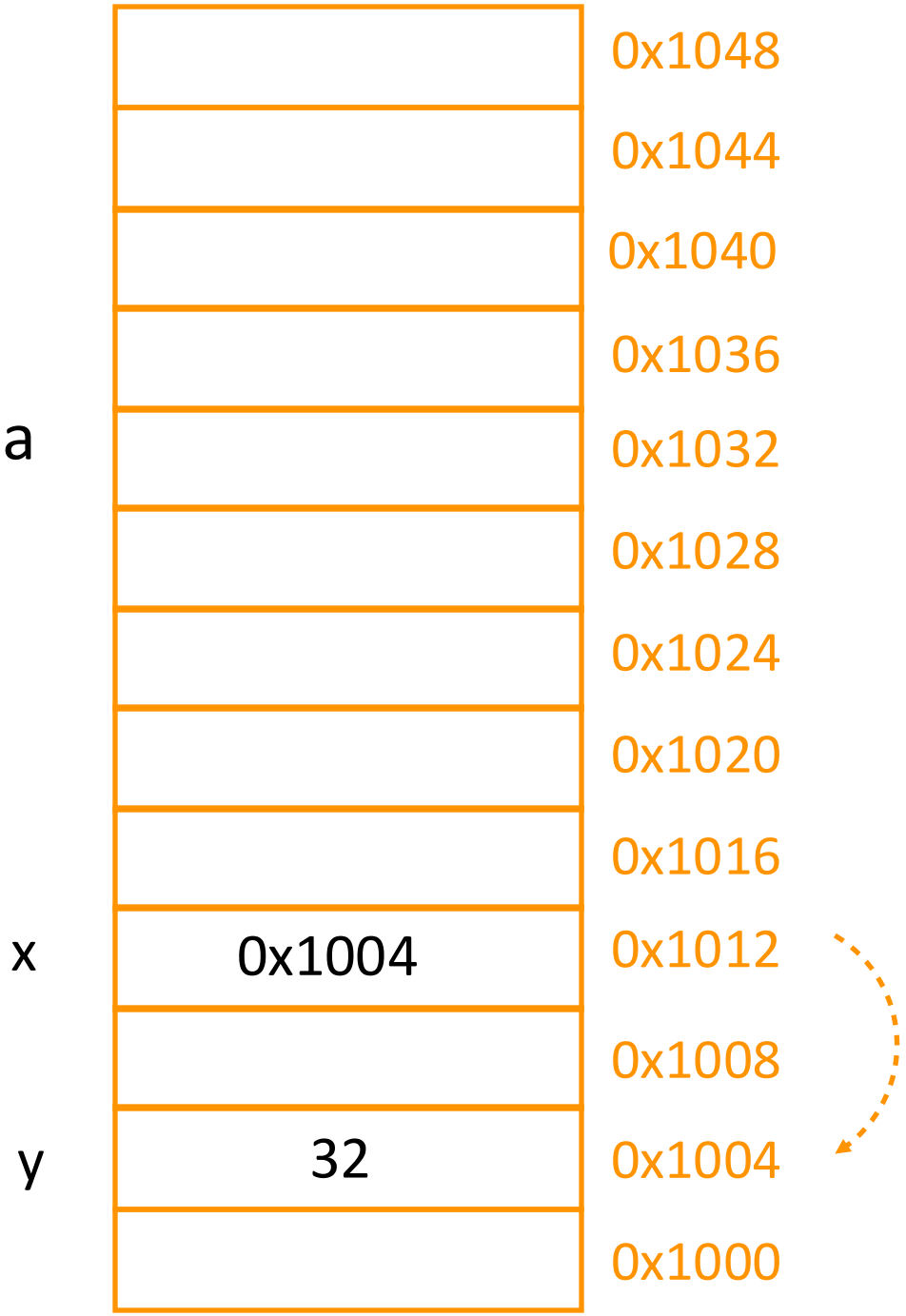
# Pointers

- A pointer is another type of variable which holds the location of a variable of a certain type
- Like all variables, we can change their value (what a pointer points to)

```
int *x = &y;
```

(a pointer to int, x)

(address of int y)



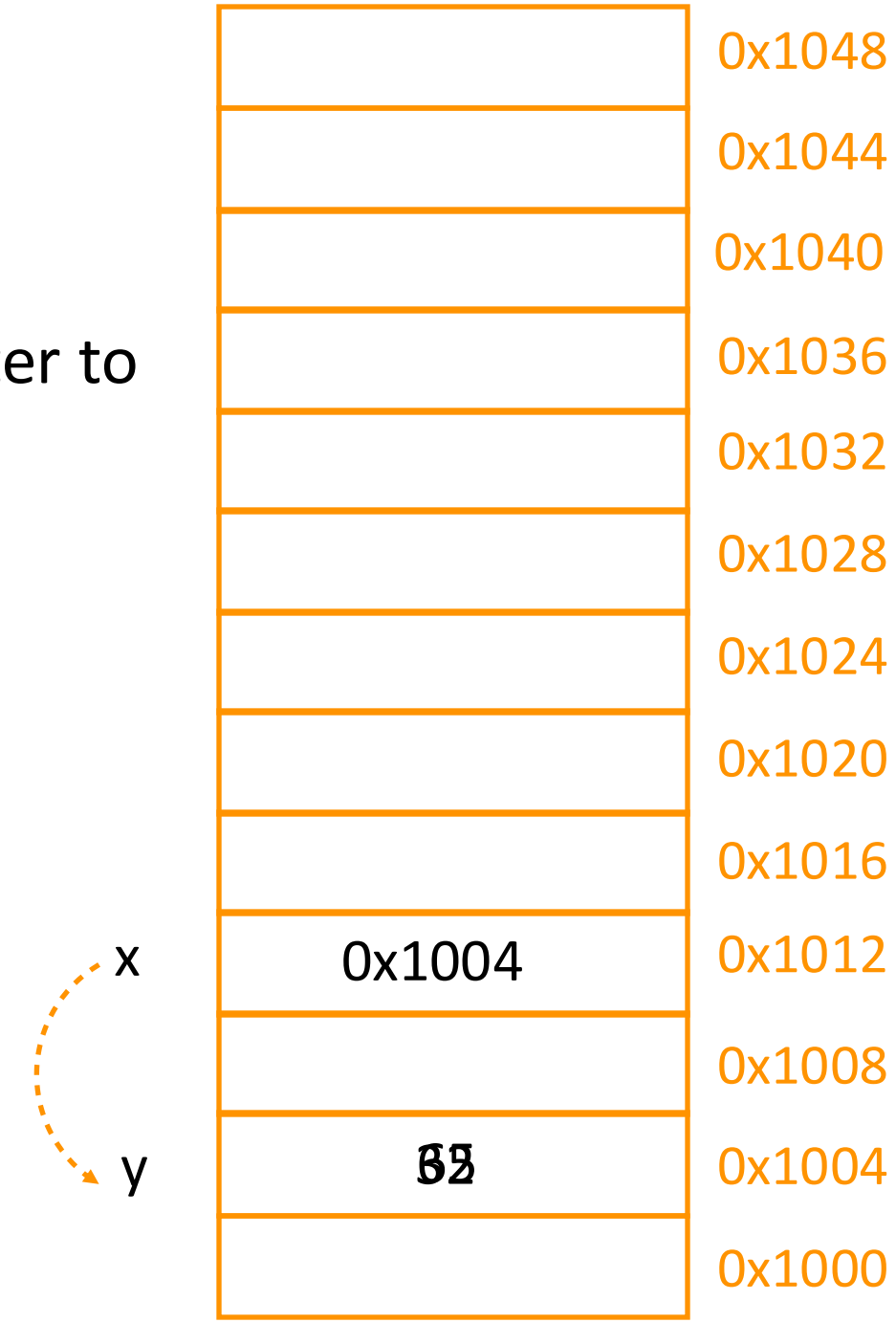

*Pointers are always the size of a memory address*

# Pointers

- We ‘indirect’ or ‘dereference’ a pointer to access the thing it points to…

*x = 65;

(dereference pointer x)



# Example, changing data passed to a function

Recall, when we call a function we *only get* a copy of the parameters' actual values

Nothing links the variables inside the function scope, *including* the parameters, with those outside (even if the names match)

```
void add_5(int value)
{
  // Actually want to modify the parameter ‘value’
  // rather than return it, this DOESN’T WORK

  value = value + 5;
}

int main()
{
  int amount = 10;

  add_5(amount);

  printf("Amount = %d\n", amount);

  return 0;
}
```

*this DOESN’T WORK*

*this DOES WORK*

```
void add_5(int *value)
{
  // 'value' is now where the data is that we modify

  *value = *value + 5;
}

int main()
{
  int amount = 10;

  add_5(&amount);

  printf("Amount = %d\n", amount);

  return 0;
}
```

We pass the **location** by value (where is the **location** of **amount** in main())

Note the ‘&’ takes the ‘address’ or a pointer to ‘amount’

# That’s why you need an & for most **scanf** parameters

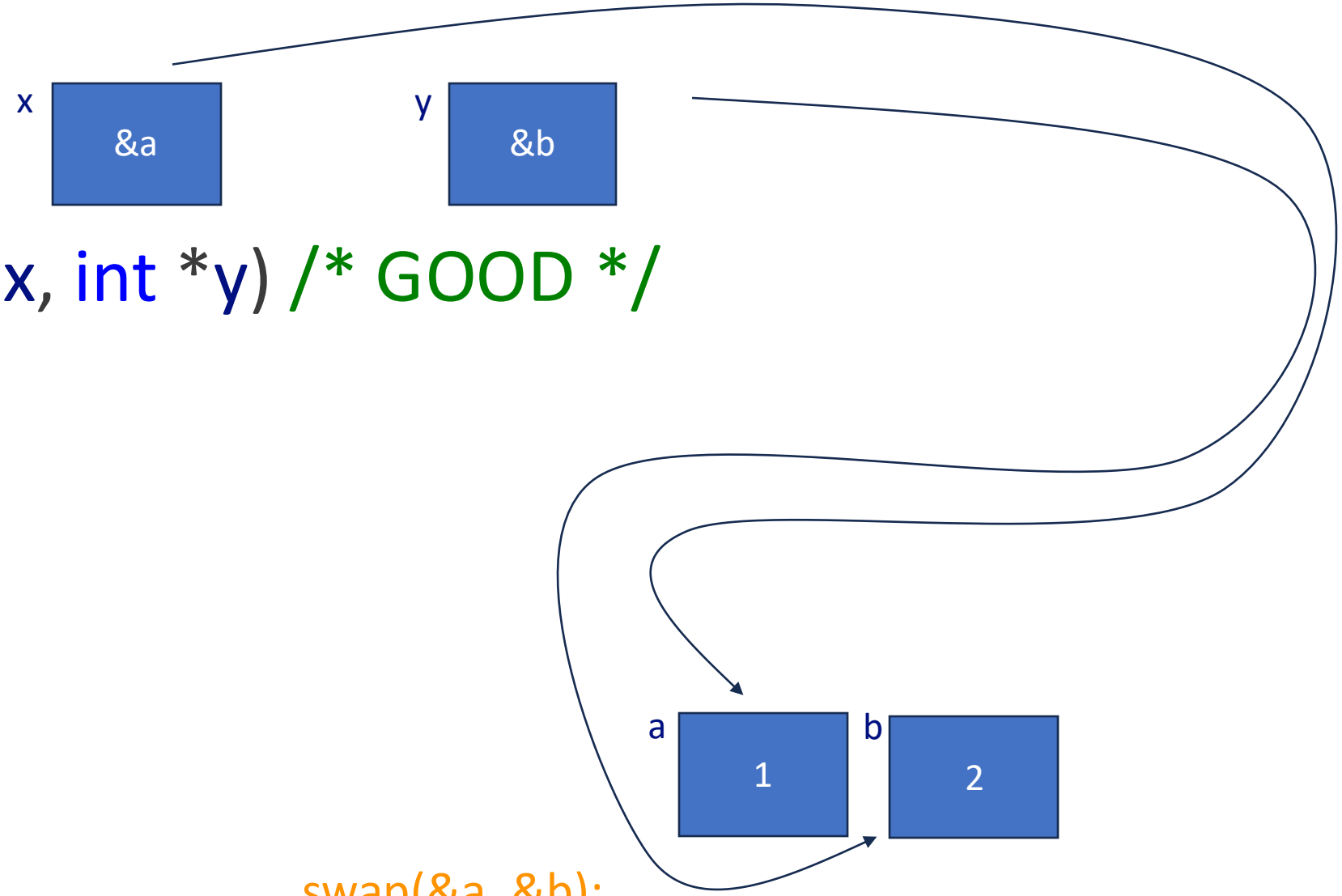
```
int main()
{
  int input;
  scanf("%d", &input); // address of ‘input’
  printf("You typed: %d\n", input);
}
```

we can use this same trick to change multiple parameters (swap function walkthrough)

```
void swap(int x, int y) /* WRONG */
{
  int temp = x;

  x = y;
  y = temp;
}
```

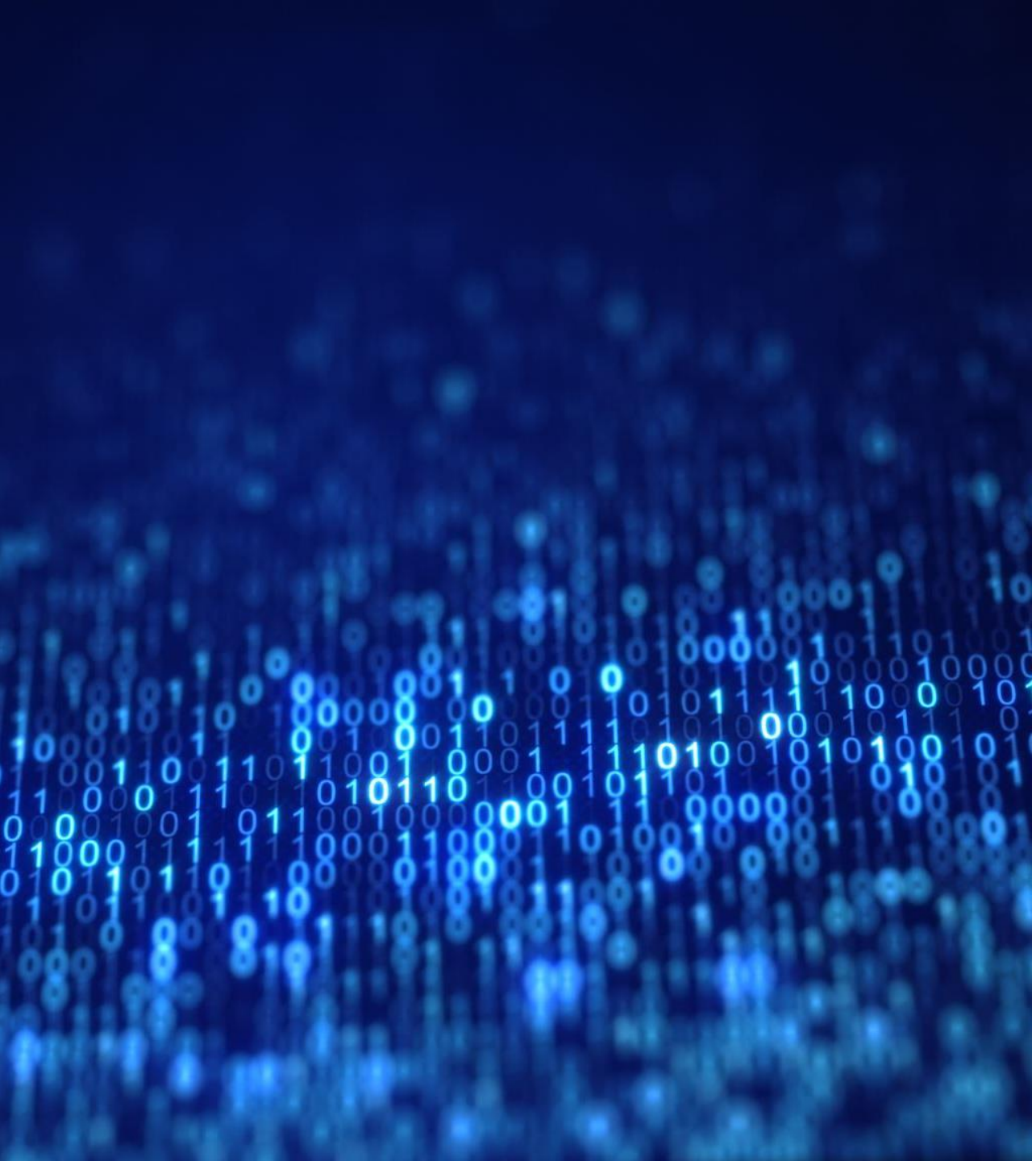
swap(a, b);
Just swaps copies of a and b inside the function!




```
void swap(int *x, int *y) /* GOOD */
{
  int temp = *x;

  *x = *y;
  *y = temp;
}
```

swap(&a, &b);
Swaps the content of a and b!



# Summary

- Introduced the concepts of indirection and pointers in C
- Using * and & notation to declare and dereference pointers
- How to use pointers with functions to *mimic* pass by reference
- *Next lecture: more powerful uses of pointers!*